Interactive Programming In Java

Chapter Outlines

Lynn Andrea Stein

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Cambridge, MA 02139 USA
ipij@ai.mit.edu
http://www.ai.mit.edu/projects/cs101/

Front Matter

I. Table of Contents
2. Preface
   1. Why Interactive Programming?
   2. Ramifications for Later Curriculum
   3. A Short History of the Rethinking CS101 Project
      1. Research Roots
      2. Classroom Experience
   4. How to Use This Book
      1. Part By Part
      2. Pedagogical Elements and Supplementary Materials
   5. About the Author
   6. Acknowledgements
3. List of Figures
4. List of Tables
5. List of Sidebars
   1. Java Sidebars
   2. Style Sidebars

Part 1: Introduction To Interactive Program Design

Chapter 1: Introduction to Program Design
Chapter Overview

Objectives of this Chapter

1. Computers and Programs
2. Thinking Like a Programmer
3. Programming Primitives, Briefly
4. Ongoing Computational Activity
5. Coordinating a Computational Community
   1. What is the Desired Behavior of the Program?
   2. Who are the Members of the Community?
   3. What Goes Inside Each One?
   4. How Do They Interact?
6. The Development Cycle
7. The Interactive Control Loop

Chapter Summary

Exercises

**Chapter 2: The Programming Process**

Chapter Overview

Objectives of this Chapter

1. The Problem
   1. Problem Requirements
      1. Assumptions
      2. Promises/Guarantees
   2. The Community Around You
      1. Program Libraries
      2. Users
      3. Physical Environment
      4. Understand their interfaces (and assumptions)
   3. Requirements are a Moving Target
      1. Software Lifecycle
      2. Documenting your Design
2. Designing a solution
   1. Who are the members?
   2. How do they Interact
1. What Promises?
2. What Assumptions?
3. Who does what?
3. What goes inside?
4. Nouns and Verbs
5. Acting it out

3. The process
   1. Keep it simple
   2. Keep it working
   3. You'll still have to debug (debugging is normal)
      1. When it's not working
      2. make state manifest
      3. Explain it to someone
      4. Act it out
   4. Documentation

Chapter Summary

Exercises

**Interlude: A Community of Interacting Entities**

Overview

Objectives of this Interlude

1. Introduction: Word Games
2. Designing a Community
   1. A Uniform Community of Transformers
   2. The User and the System
   3. What Goes Inside
3. Building a Transformer
   1. Transformer Examples
   2. Strings
      1. String Concatenation
      2. String Methods
   3. Rules and Methods
   4. Classes and Instances
   5. Fields and Customized Parts
   6. Generality of the Approach
4. Summary

Suggested Exercises

- Sidebar: Selected String Methods

**Part 2: Entities and Interactions**

**Chapter 3: Things, Types, and Names**

Chapter Overview

Objectives of this Chapter

1. Things
   1. Primitive Things and Literals
      1. Numbers
      2. Characters and Strings
      3. Booleans
   2. Objects
2. Naming Things
   1. Referring to Things
   2. Assignment
3. Types
   1. Declarations and The type-of-thing name-of-thing Rule
   2. Definition = Declaration + Assignment
   3. Primitive Types
   4. Object Types
4. Types of Names
   1. Shoebox Names
   2. Label Names

Chapter Summary

Exercises

- Sidebar: Java Naming Syntax and Conventions
- Sidebar: Java Primitive Types

**Chapter 4: Specifying Behavior: Interfaces**

Chapter Overview
Objectives of this Chapter

1. Interfaces are Contracts
   1. Generalized Interfaces and Java Interfaces
   2. A Java Interface Example
2. Method Signatures
   1. Name
   2. Parameters and Parameter Types
   3. Return Type
   4. Putting It All Together: Abstract Method Declaration Syntax
   5. What a Signature Doesn't Say
3. Interface Declaration
   1. Syntax
   2. Method Footprints and Unique Names
   3. Interfaces are Types: Behavior Promises
   4. Interfaces are Not Implementations

Chapter Summary

Exercises

- Style Sidebar: Method Documentation
- Style Sidebar: Interface Documentation

See also Java Chart on Interfaces.

**Chapter 5: Expressions: Doing Things With Things**

Chapter Overview

Objectives of this Chapter

1. Simple Expressions
   1. Literals
   2. Names
2. Method Invocation
3. Combining Expressions
4. Assignments and Side-Effecting Expressions
5. Other Expressions that Use Objects
   1. Field Access
   2. Instance Creation
   3. Type Membership
6. Complex Expressions on Primitive Types: Operations
   1. Arithmetic Operator Expressions
   2. Explicit Cast Expressions
   3. Comparator Expressions
   4. Logical Operator Expressions
7. Parenthetical Expressions and Precedence

Chapter Summary

Exercises

- Style Sidebar: Don't Embed Side-Effecting Expressions
- Sidebar: Java Operators
- Sidebar: Arithmetic Expressions
- Sidebar: Coercion and Casting
- Sidebar: Java Operator Precedence
- Sidebar: Other Assignment Operators

See also Java Chart on Expressions

**Chapter 6: Statements and Rules**

Chapter Overview

Objectives of this Chapter

1. Statements and Instruction-Followers
2. Simple Statements
3. Declarations and Definitions
4. Sequence Statements
5. Flow of Control
   1. Simple Conditionals
   2. Simple Loops
6. Statements and Rules
   1. Method Invocation Execution Sequence
   2. Return

Chapter Summary

Exercises

- Style Sidebar: Formatting Declaration Statements
Interlude: **Entities and Aggregates/Rules and Roles**

Overview

Objectives of this Interlude

1. The Problem
2. Representation
3. Interacting with the Rules
4. Paying Attention to the World
5. Fancy Dot Tricks
6. Remembering State
   1. Fields
   2. Fields vs. Variables
7. Summary

Suggested Exercises

Chapter 7: Building New Things: **Classes and Objects**

Chapter Overview

Objectives of this Chapter

1. Classes are Object Factories
   1. Classes and Instances
   2. Recipes Don't Taste Good
   3. Classes are Types
2. Class Declaration
   1. Classes and Interfaces
      1. implements and type inclusion
      2. contract vs. implementation
3. Data Members, or Fields
   1. Fields are not Variables
      1. Hotel Rooms and Storage Rental
      2. Whose Data Member is it?
3. Scoping of Fields
4. Comparison of Kinds of Names

2. Static Members

4. Methods
   1. Method Declaration
   2. Method Body and Behavior
   3. A Method ALWAYS Belongs to an Object
      1. this.
      2. Static Methods
   4. Method Overloading

5. Constructors
   1. Constructors are Not Methods
   2. Syntax
   3. Execution Sequence
   4. Multiple Constructors and the Implicit No-Arg Constructor
   5. Constructor Functions

Chapter Summary

Exercises

- Style Sidebar: Class Declaration
- Sidebar: Java Types and Default Initialization
- Table: Comparison of Kinds of Names
- Style Sidebar: Field Documentation
- Style Sidebar: Method Implementation Documentation
- Sidebar: Method Invocation and Execution
- Style Sidebar: Constructor Documentation
- Style Sidebar: Capitalization Conventions

See also Java Charts on Classes, Methods, and Fields.

Part 3: Refining Designs

Chapter 8: Designing with Objects

Chapter Overview

Objectives of this Chapter

1. Object-Oriented Design
1. Objects are Nouns
2. Methods are Verbs
3. Interfaces are Adjectives
4. Classes are Object Factories
5. Some Counter Code (An Example)
6. Public and Private

2. Kinds of Objects
   1. Data Repositories
   2. Resource Libraries
   3. Traditional Objects

3. Types and Objects
   1. Declared Types and Actual Types
   2. Use Interface Types
   3. Use Contained Objects to Implement Behavior
   4. The Power of Interfaces

Chapter Summary

Exercises

- Style Sidebar: Class and Member Documentation
- Sidebar: Final
- Sidebar: class Math

Collections: An Extended Example

Overview

Objectives

1. Exercises

Chapter 9: Animate Objects

Chapter Overview

Objectives of this Chapter

1. Animate Objects
2. Animacies are Execution Sequences
3. Being Animate-able
   1. Implementing Animate
2. AnimatorThread
3. Creating the AnimatorThread in the Constructor
4. A Generic AnimateObject

4. More Details
   1. AnimatorThread Details
   2. Delayed Start and the init() Trick
   3. Threads and Runnables

5. Where do Threads come from?
   1. Starting a Program
   2. Why Constructors Need to Return

- Sidebar: class AnimatorThread
- Sidebar: Thread Methods
- Sidebar: class Main
- Style Sidebar: Using main()

Chapter Summary

Exercises

**Chapter 10: Reusing Implementation: Inheritance**

Chapter Overview

Objectives of this Chapter

1. Derived Factories
   1. Simple Inheritance
   2. java.lang.Object
   3. Superclass Membership

2. Overriding
   1. super.
   2. The Outside-In Rule
   3. Problems with Private

3. Constructors are Recipes
   1. this()
   2. super()
   3. Implicit super()

4. Interface Inheritance

5. Relationships Between Types
The class Object
Style Sidebar: Explicit Use of this. and super()
Sidebar: Abstract Classes

1. Chapter Summary

Exercises

Interlude: A System of Animate Objects

Objectives

Overview

Suggested Exercises

Chapter 11: When Things Go Wrong: Exceptions

Chapter Overview

Objectives of this Chapter

1. Exceptional Events
   1. When Things Go Wrong
   2. Expecting the Unexpected
   3. What's Important to Record
2. Throwing an Exception
3. Catching an Exception
4. Throw vs. Return
5. Designing Good Test Cases

Sidebar: Throw Statements and Throws Clauses
Sidebar: Try Statement Syntax
Sidebar: Exceptions, Errors, and RuntimeExceptions

1. Chapter Summary

Exercises

Part 4: Refining Interactions

Chapter 12: Dealing with Difference: Dispatch

Chapter Overview
Objectives of this Chapter

1. Conditional Behavior
2. If and else
   1. Basic Form
   2. Else
   3. Cascaded Ifs
   4. Many Alternatives
3. Limited Options: Switch
   1. Constant Values
      1. Symbolic Constants
      2. Using Constants
   2. Syntax
      1. Basic Form
      2. The Default Case
      3. Variations
      4. Switch Statement Pros and Cons
4. Arrays
   1. What is an Array?
      1. Array Declaration
      2. Array Construction
      3. Array Elements
   2. Manipulating Arrays
      1. Stepping Through an Array Using a For Statement
      2. Using Arrays for Dispatch
5. When to Use Which Construct

- Sidebar: if Statement Syntax
- Sidebar: final
- Style Sidebar: Use Named Constants
- Sidebar: break and continue statements
- Sidebar: switch Statement Syntax
- Sidebar: Array Syntax
- Sidebar: for Statement Syntax

1. Chapter Summary

Exercises

Chapter 13: Encapsulation
Chapter Overview

Objectives of this Chapter

1. Design, Abstraction, and Encapsulation
2. Procedural Abstraction
   1. The Description Rule of Thumb
   2. The Length Rule of Thumb
   3. The Repetition Rule of Thumb
   4. An Example
   5. The Benefits of Abstraction
3. Protecting Internal Structure
   1. private
   2. Packages
      1. Packages and Names
      2. Packages and Visibility
   3. Inheritance
   4. Clever Use of Interfaces
4. Inner Classes
   1. Static Classes
   2. Member Classes
   3. Local Classes and Anonymous Classes
5.

- Style Sidebar: Procedural Abstraction
- Sidebar: Package Naming Summary
- Sidebar: Package Visibility Summary
- Sidebar: Inner Classes

1. Chapter Summary

Exercises

Chapter 14: Intelligent Objects and Implicit Dispatch

Chapter Overview

Objectives of this Chapter

1. Procedural Encapsulation and Object Encapsulation
2. From Dispatch to Objects
   1. A Straightforward Dispatch
2. Procedural Encapsulation
3. Variations
4. Pushing Methods Into Objects
5. What Happens to the Central Loop?

3. The Use of Interfaces
4. Runnables as First Class Procedures
5. Callbacks
6. Recursion

   1. Structural Recursion
      1. A Recursive Class Definition
      2. Methods and Recursive Structure
      3. The Power of Recursive Structure

   2. Functional Recursion

Chapter Summary

Exercises

Chapter 15: Event-Driven Programming

Chapter Overview

Objectives of this Chapter

1. Control Loops and Handler Methods
   1. Dispatch Revisited
2. Simple Event Handling
   1. A Handler Interface
   2. An Unrealistic Dispatcher
   3. Sharing the Interface
3. Real Event-Driven Programming
   1. Previous Examples
   2. The Idea of an Event Queue
   3. Properties of Event Queues
4. Graphical User Interfaces: An Extended Example
   1. java.awt
   2. Components
   3. Graphics
   4. The Story of paint
5. Events and Polymorphism
Chapter Summary

Exercises

See also the AWT Quick Reference.

**Interlude: Achieving Customized Behavior**

Objectives

Overview

Suggested Exercises

**Chapter 16: Event Delegation (and AWT)**

Chapter Overview

Objectives of this Chapter

1. Model/View: Separating GUI Behavior from Application Behavior
   1. The Event Queue, Revisited
2. Reading What the User Types: An Example
   1. Setting up a User Interaction
   2. Listening for the Event
   3. Registering Listeners
   4. Recap
3. Specialized Event Objects
4. Listeners and Adapters: A Pragmatic Detail
5. Inner Class Niceties

- Sidebar: cs101.awt.DefaultFrame

1. Chapter Summary

Exercises

See also the AWT Quick Reference.

**Interlude: An AWT Application**

Objectives

Overview
Suggested Exercises

Part 5: Systems of Objects

Chapter 17: Models of Communities

Chapter Overview

Objectives of this Chapter

1. State Machines
2. State Spaces
3. Organizational Behavior
4. Network Models
5. Patterns
6. UML
7. Metrics
   1. Static Complexity
   2. Throughput and Latency

Sidebar: FSM Rules

1. Chapter Summary

Exercises

Chapter 18: Interfaces and Protocols: Gluing Things Together

Chapter Overview

Objectives of this Chapter

1. Pacing
2. Procedure Calls
3. Callbacks
4. Explicit Communication Channel Objects
5. Protocols

Chapter Summary

Exercises

Chapter 19: Communication Patterns
Chapter Overview

Objectives of this Chapter

1. What is a Client-Server Interaction?
2. Implementing Client-Server Interactions
   1. Client Pull
   2. Server Push
3. The Nature of Duals
4. Pushing and Pulling Together
   1. Passive Repository
   2. Active Constraint

Chapter Summary

Exercises

Interlude: Combining Events and Interactive Control Loops

Objectives

Overview

Suggested Exercises

Chapter 20: Synchronization

Chapter Overview

Objectives of this Chapter

1. Reads and Writes
2. An Example of Conflict
3. Synchronization
4. Java synchronized
   1. methods
   2. (blocks)
5. What synchronization buys you
6. Safety rules
7. Deadlock
8. Obscure Details

Chapter Summary
Chapter 21: Network Programming

Chapter Overview

Objectives of this Chapter

1. A Readable Writeable Channel
   1. Tin Can Telephones
   2. Streams
2. Using A Channel
   1. For Writing
      1. Flushing Out the Stream
      2. A Scribe Example
   2. For Reading
      1. Reading and Blocking
      2. A Lector Example
   3. Encapsulating Communications
3. Real Streams
   1. Abstract Stream Classes
   2. Decorator Streams
   3. Stream Sources
   4. Decoration in Action
4. Network Streams: An Example
   1. Starting from Streams
   2. Decorating Streams
   3. Sockets and Ports
   4. Using A Socket
   5. Opening a Client-Side Socket
   6. Opening a Single Server-Side Socket
   7. A Multi-Connection Server
   8. Server Bottlenecks

Chapter Summary

Exercises

Interlude: Client/Server Chat

Objectives
Overview

Suggested Exercises

**Chapter 22: Conventional Architectures**

Chapter Overview

Objectives of this Chapter

1. Server Architectures
   1. Dumb broadcast server
   2. Routing server
   3. DNS
2. RPC
3. Peer Architectures
   1. Ring
   2. Round Robin
   3. Cubes
4. Arbitration
5. Blackboard
6. Tuple-space

Chapter Summary

Exercises

**Appendices**

1. [Applets](#)
2. [AWT Quick Reference](#)
   1. AWT Components
   2. Component
   3. Canvas
   4. Widgets and their Event Types
   5. Basic Widgets
   6. ItemSelectable Widgets
   7. Text Widgets
   8. Container
   9. Panel and Frame
   10. Dimension, Point, and Rectangle
11. Graphics
12. AWT Events
13. ActionEvent and ActionListener
14. AWT Listeners and Adapters

3. **IO Quick Reference**
   1. InputStream and Reader
   2. OutputStream and Writer
   3. Sources of Streams
   4. InputStreamReader and OutputStreamWriter
   5. Files
   6. Pipes
   7. Streams that Add Features
   8. Buffering
   9. Primitive Data
   10. Object Streams and Serialization
   11. Other Useful Streams
   12. IOExceptions

4. **Java Charts**
   1. About Java Charts
   2. Program File
   3. Class Declaration
   4. Field Declaration
   5. Method Declaration
   6. Expression
   7. Statement
   8. Disclaimers, Notes, Amendments, etc.

5. **Glossary**

6. Indices
   1. Syntax Sidebars
   2. Style Sidebars
   3. Interludes
   4. Case Studies
   5. Terms

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's [Rethinking CS101](http://www.cs101.org/ipij) Project at the [Computers and Cognition Laboratory](http://www.cs101.org/ipij) of the [Franklin W. Olin College of Engineering](http://www.cs101.org/ipij).
formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
webmaster@cs101.org
Preface

*Interactive Programming* is an introduction to computer programming intended for students in standard CS1 courses (or interested professionals) with no prior programming experience. It is the first textbook to rethink the traditional curriculum in light of the current interaction-based computer revolution. *Interactive Programming* shifts the foundation on which the teaching of Computer Science is based, treating computation as *interaction* rather than *calculation*, thus providing students with a solid grounding in the thought that underlies modern software practice. Students still learn the basic and necessary elements of computer programming and the Java language, but the context in which they learn it is more consistent both with Java's tools and philosophy and with the prevailing practice from which it arises.

Why Interactive Programming?

Traditionally, introductory programming teaches algorithmic problem-solving. In this view, a program is a sequence of instructions that describe the steps necessary to achieve a desired result. The 'pieces' of this program are these steps. They are combined by sequencing. The program produced is evaluated by means of its end result. Students trained in this way often have difficulty moving beyond the notion that there is a single thread of control over which they have complete control.

In contrast, most programs of interest today are made up of implicitly or explicitly concurrent components that interact to provide ongoing services. Buzzwords such as "client/server" and "event-driven" are part of the descriptive language of this new generation of programs. Embedded systems and software agents typify their incarnations. User interface design, distributed programming, and the world-wide web are logical extensions of a way of thinking that has interaction at its core.

When programming is taught from a traditional perspective, important topics like these are treated as advanced and inaccessible to the introductory student. It is unsurprising that senior software engineers report that today's undergraduates are ill-equipped to handle the realities of embedded interactive software. Most require on-the-job retraining to "think concurrently." Students trained in the traditional curriculum are often so indoctrinated in the "sequence of steps" mentality that they can no longer rely on the intuition common to every child coordinating a group of friends or trying to sneak a cookie behind her parent's back.

*Interactive Programming* provides an alternate entry into the computer science curriculum. It teaches problem decomposition, program design, construction, and evaluation, beginning with the following premises: A program is a community of interacting entities. Its "pieces" are these implicitly or explicitly concurrent entities: user interfaces, databases, network services, etc. They are combined by virtue of ongoing interactions which are constrained by interfaces and by protocols. A program is evaluated by its adherence to a set of invariants, constraints, and service guarantees -- timely response, no memory leaks, etc.

Because it begins from this alternate notion of what programming is about, *Interactive Programming* tells a rather different story from the traditional introductory programming book. By its end, students are empowered to write and read code for client-server chat programs, networked video games, web servers,
user interfaces, and remote interaction protocols. They build event-driven graphical user interfaces and spawn cooperating threads. Each of these programs -- all of which are beyond the scope of traditionally taught introductory courses -- is a natural extension of the community metaphor for computation.

Many computer science departments are contemplating a change to the Java programming language for introductory computer science courses. While it is possible to make this change without transforming the introductory curriculum, adopting Java without a corresponding curricular change amounts to sweeping more and more of what is important in today's computational world under the rug. Java embodies much of modern programming practice. Insisting on traditional approaches actually makes certain aspects of the language less accessible. Shifting to a curriculum in which concurrent interacting entities play a central role makes far more of modern computation theory, practice, and tools accessible to today's introductory student.

A more complete argument for rewriting the introductory computer science curriculum in this way is contained in "What We've Swept Under the Rug: Radically Rethinking CS1" (Computer Science Education Journal, to appear). See also http://www.ai.mit.edu/projects/cs101/.

### Ramifications for Later Curriculum

*Interactive Programming* includes a number of topics not often taught to introductory students: networks, user interfaces, client/server architecture, and event-driven programming. At the same time, students will develop a basic facility for programming and for problem decomposition, the most crucial skills taught in most existing CS1 courses.

In all respects, this course is still an introductory programming course. Its thematic lesson concerns a model of computation as interaction, rather than calculation. But its pragmatic goals include most of the skills that are learned in standard introductory CS. The fundamental lesson of this course remains how to take a description of a problem and construct a program whose behavior solves that problem. It differs from traditional courses in its underlying assumptions, the kinds of descriptions that can be considered, and the corresponding conceptualizations that are used to build a program. The computational constructs and modeling tools have changed; the problem still remains the programming.

As a result, this new CS1 course requires little revision of the rest of the computational course sequence. Upper level courses can continue as they are, but are likely to find their task simplified somewhat by the new perspective that students bring to them.

The remainder of the curriculum which begins with an introduction to computation on these terms may thus look much like the existing computer science undergraduate curriculum. Nonetheless, there are subtle but significant improvements. Several important topics that are currently covered only in advanced undergraduate or graduate level classes can be introduced earlier in the curriculum. For example, topics in distributed algorithms and parallel complexity -- such as the time/processor tradeoff -- can be taught in the first course in computer science theory if the model of parallel computation is already familiar. Since modern algorithms increasingly makes use of such approaches, it seems only natural to expose our undergraduates to the fundamental ideas in these areas.

Other topics, already present at the undergraduate level, become much easier to explain when students come equipped with this world view. Much of operating systems becomes an exploration of different methods for implementing and ensuring appropriate behavior multiprocessing, rather than focusing on the
concept of parallel execution itself. Students seeing these ideas for the second time, now in depth, are more likely to appreciate some of the subtleties of the problem rather than being confused by the many levels at which operating system code must operate. Synchronization and interprocess communication can be introduced along with scheduling. Transaction-safety, remote procedure call, and shared memory models similarly follow smoothly from this approach.

Further, a whole host of issues that now fit into our curriculum poorly, if at all, now become sensible parts of the model of computation that we teach our students. For example, the traditional curriculum has a tremendously difficult time introducing the topic of user interfaces. In many schools, this "special case" is tacked on to the curriculum as an afterthought (or altogether ignored), largely because it just doesn't fit. To readers of this book, however, accounting for the role of the user becomes straightforward. The user is another member of the community of interacting processes that together constitute our computation. The programmer's job is to develop an acceptable interface that gives each participant -- program or person -- an appropriate set of responsibilities and services. Of course, a human has different skills and needs from a computer program, but this, too, is a natural part of our larger way of thinking -- and teaching -- about computational systems.

Teaching computation this way also has the potential to harness our students' natural instincts. Traditional introductory courses tell their students, "Forget all of your intuitions about how the world works. This is computation; it is nothing like the world in which you live." Instead, Interactive Programming teaches that computation is very much like the world in which we live. It harnesses our intuitions about that world---about simultaneity and ordering constraints, about when it is more useful to partition a task and when it is simpler not to, and about what information must be available to whom at what time and how to get it there--and teaches readers to use that intuition to become better programmers.

A Short History of the Rethinking CS101 Project

This book is a part of a larger project to reshape the ways in which introductory computer science is taught (and, indeed, the ways in which the field itself is conceptualized). The Rethinking CS101 Project grew out of work in a variety of computational fields -- artificial intelligence, robotics, software agents, human-computer interaction, as well as programming languages -- and their common difficulties with the conventional wisdom concerning how computation is constituted. For example, introductory computer science teaches that a program's job is to calculate some desired result and then to stop. When a robot stops, however, this is generally a sign that it has broken. (Further, there's not really a "result" that the robot "calculates"; instead, it is supposed to continually exhibit appropriate behavior.)

Research Roots

In the early 1990s, the author worked to bring intuitions about computation into the classroom through the use of simple, inexpensive robotics. The use of robots enabled a focus on software life cycle, non-repeatability, and pragmatic software engineering uncommon in traditional introductory classrooms. The curriculum that developed from this experimentation marked a radical departure from the traditional single-threaded, sequentialist story.

The use of robotics clearly forced a shift in perspective in the introductory programming curriculum. In the first half of the decade, this shift was echoed, if more subtly, in the popular software market through approaches such as event-driven programming, client-server architectures, and enterprise computing.
Those techniques -- increasingly important to industry -- were still not deemed suitable for an introductory computing classroom. Nonetheless, they were inescapably changing the face of the computing sciences. Computing-in-the-raw is no longer calculate-and-stop. Instead, it is made up of agents and services, communities of ongoing interacting entities. Yet today's introductory classrooms shed little light on these now-prevalent industry practices.

Courses taught during this period included MIT freshmen, MIT graduate students, and international researchers in artificial intelligence. Spin-offs of these efforts include robotics classes at a variety of universities and colleges as well as the now-annual Robot-Building Laboratory at the National Conference on Artificial Intelligence and the establishment of the KISS Institute for Practical Robotics (of which the author is an Institute Fellow).

With the advent of the world-wide web and the popular adoption of Java, a new avenue towards teaching these approaches has been opened. The current Rethinking CS101 Project has shifted its focus away from physical robots and towards the underlying principles of interactive computation as illustrated by purely software systems. (A side effort within the project continues to pursue the robot hook, both in software simulations and in the interests of capitalizing on the newly emerging commodity robot market. Although robots are not central to the curricular shift represented by this project, they are easily integrated into its methods and models.) Interactive Programming represents the codification of the underlying approach to computation in a form suitable for adoption in otherwise-traditional university computer science curricula, thereby bringing them closer to state-of-the-art practice.

Classroom Experience

The curriculum presented in Interactive Programming has been taught in a variety of venues. The first course taught with the current set of materials was held in the summer of 1996, in a one-week intensive minicourse using the Java 1.0 API and Sun's JDK, the only Java available at the time. Its students were executives, managers, and a few software engineers enrolled in MIT's Summer Professional Programs. The majority had no substantial prior programming experience.

The course was subsequently taught twice in MIT's regular curriculum. Students were largely first-semester freshmen and others with no prior programming experience. (The course is also popular among advanced students in non-computational fields who want a single semester of computational coursework.) Student feedback has been resoundingly positive. The MIT course has been adopted by the EECS Department as a regular offering and is listed in the catalog as subject number 6.030, Introduction to Interactive Programming.

Precursors to this textbook were also used in teaching several other minicourses to professional audiences. These include the 1997 and 1998 Professional Institutes at MIT and a tutorial offered at the ACM SIGPLAN's Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '97). Students in these courses included software professionals, academics, and trainers. Generally versed in traditional programming, they attended the minicourses to learn a new way to think about computation.

Other instructors have used the beta release of the textbook. In the fall of 1998, the course materials was used at a handful of undergraduate institutions with student bodies substantially less sophisticated than MIT's, as well as an advanced class in a secondary school. Serious beta testing began in the fall of 1999,
when over a thousand students at more than a dozen colleges and universities around the world used *Interactive Programming* as their primary text. Additional non-traditional classroom tests are also underway. Ultimately, the textbook is intended for deployment in mainstream undergraduate classrooms as well as certain advanced secondary classes, perhaps AP.

The curriculum itself has attracted widespread attention. It has been presented at a variety of international meetings and its agenda is documented in a variety of publications (see enclosures). The Rethinking CS101 Project at MIT has recently received the donation of a 30-machine teaching laboratory from Microsoft Research/University Curriculum Programs. A strategic relationship with Sun Microsystems is also under negotiation, and the National Science Foundation has selected Rethinking CS101 for an Educational Innovation Award.

**How to Use This Book**

*Interactive Programming* is designed for use by students who have no prior programming experience (typically college freshmen). It ultimately teaches both the fundamentals of computer programming and the details of the Java programming language.

The book is divided into five parts. The first briefly overviews the idea of programs built out of communities of interacting entities. The second part introduces the mechanics of Java programming, from things, types, and names to objects and classes. It is essential to the book and is intended to be read in the order presented. Part three elaborates on these ideas, introducing threads as first-class citizens of the programming world and exploring inheritance, exception-handling, and design. Part four emphasizes a variety of issues in the design of an individual entity. It is not necessary to read this section in any particular order, and certain chapters can be omitted entirely without serious detriment. Part five similarly surveys a variety of interrelated topics, in this case concerning the ways in which communities are coupled together, and its chapters, too, can be taken out of order or omitted.

The five parts, taken together, constitute a single-semester introductory course in computer programming. In such a course, some of the supplementary material (described below) will not be used. For a one-quarter course, part five and selected earlier chapters should probably be omitted. Alternately, the complete book can be spread over two quarters or over a full year, augmented as necessary from the supplementary materials.

**Part By Part**

Part 1 is brief and introductory, providing an overview of the approach to computer programming taken. Part 2 begins with the basic syntax and semantics of programming constructs. At the same time, from the earliest examples, students are introduced to concurrent, interactive, embedded programs. For example, interfaces are introduced early as they specify a contract between two parts of a computer system. By the middle of part 3, students have learned to write what might in other contexts be called "stand-alone" programs -- complete programs including class definitions and a main routine. They have also learned that every program is a part of a system of interacting entities -- including the user, libraries and other software, hardware, etc. -- and that no program truly stands alone.

The remainder of the book addresses issues and alternatives that arise in the design of software communities. Part 4 focuses on ways to extend the basic entities that students build. The notion of a
dispatching control loop provokes an exploration of procedural abstraction, in which separate routines handle each possible case. This in turn leads to a de-emphasis of the central control loop and a shift to event-driven programming, in which individual "handler" procedures take center stage. In a typical event system, dispatch may be provided implicitly, i.e., by underlying hardware or software. A third model -- smart objects that handle their own behavior -- is also explored. Java's AWT is introduced as both a tool and an example of an event-based system.

Part 5 addresses the issue of how entities are tied together. A recurring theme -- throughout the book, but emphasized here -- concerns interface design. This refers both to the Java construct -- a signature specification, introduced in chapter 4 -- and to the more general concept, including human (user) interface design. In addition to learning how to specify an interface, students learn what the interface does not specify. In other chapters, students learn about streams, messages, and shared memory, about connecting to objects in the same name space and to those running under different processes or on different machines, and about how to communicate with them. They also learn the basic ideas of safety and liveness, that shared mutable state can lead to program failures, and some simple mechanisms for coping with them. They do not, of course, learn to build arbitrarily complex programs that avoid deadlock under all circumstances. This topic will be visited later in the computer science curriculum. Instead, they learn to recognize the general preconditions for the possibility of safety failures and the kinds of solutions that might be possible. The goal, throughout this course, is to give students the basic conceptual vocabulary that will allow them to ask the right questions as they meet more complex issues later in their education.

*Interactive Programming* ends with an overview of various patterns of large-scale systems architecture, reviewing tradeoffs among various approaches and providing a common language for software architects. The last chapter examines conventional patterns by which complex concurrent and distributed systems are constructed. The emphasis is on designing and understanding a variety of interactive communities. This chapter also leads naturally into final projects. In courses taught using this curriculum and preliminary drafts of the book, typical final projects have included client/server chat programs and networked video games. Not what you would generally expect from first semester freshmen!

**Pedagogical Elements and Supplementary Materials**

Although this book is primarily intended for an introduction to computer science course, it will include enough reference material to stand alone as a self-study course in Java, without requiring a language supplement. Three kinds of supplementary materials help provide this support: in-chapter sidebars, between-chapter interludes, and auxiliary case studies. Reference charts and a glossary are also included.

To avoid muddying the text with too many language-specific details, sidebars are used throughout to explain details of Java syntax and semantics. The text explicates the conceptual development of the ideas; the sidebars are intended to provide detailed information on technical aspects of the language or the programming process.

Sidebars come in two flavors. Syntax sidebars explain language-specific details and pragmatics in the form of a reference manual. Style sidebars explain good documentation and coding practice. The use of sidebars serves two purposes. First, it frees the main text of some of the details that confuse rather than elucidate the presentation of central concepts. Second, the sidebars, together with the reference charts in Appendix B, form a supplementary desktop reference for students while they are programming.
The narrative of the book is periodically interrupted for an extended example, called an interlude. Interludes are adapted from potential programming assignments. They are presented between chapters, rather than within them, and can be included or omitted at the instructor's preference. Interludes provide detailed illustrations for the student to study. They exemplify the themes of the course in terms of the material studied to that point. They also provide the basis for exercises allowing students to practice and assess their mastery of relevant skill sets. Complete code for each interlude is supplied on the textbook's web site.

Also supplementing the book is a set of case studies. These are not included within the bound text. Instead, they will be made available over the world-wide web. The case studies provide descriptions of current applications exemplifying the principles central to the course. For example, one case study is based on an article in the trade literature on constructing an http server. With only minor modification, this article is an excellent illustration of the relevant themes of the course as well as a concrete example of a real-world application that is accessible to students in the later chapters.

In addition to the materials described above, the supporting materials include a set of exercises, lecture notes, programming assignments, and sample quizzes. Some exercises appear chapter by chapter in the bound book. Other resources are available through the online supplement.

About the Author

Lynn Andrea Stein is Associate Professor of Computer Science and Engineering at the Massachusetts Institute of Technology, where she has been a member of the faculty since 1990. She is also a member of both MIT's Artificial Intelligence Laboratory and its Laboratory for Computer Science. Stein, an internationally recognized researcher and educator, has been teaching computer science since the early 1980s. Stein received her undergraduate degree (AB cum laude in Computer Science) from Harvard and Radcliffe Colleges and both ScM and PhD degrees from the Department of Computer Science at Brown University. While at Brown, she held an IBM Graduate Fellowship and received the Sigma Xi Graduate Student Award.

Stein's research work spans a variety of fields and her publications include seminal work in fields as diverse as the semantics of sharing in object-oriented programming languages, non-monotonic taxonomic reasoning, and cognitive architectures for robotics. Within the past year, Stein has given invited addresses at the National Conference on Artificial Intelligence, the European Meeting on Cybernetics and Systems Research (W. Ross Ashby Plenary Lecture of the International Federation for Systems Research), and the Consortium for Computing in Small Colleges Northeastern Conference (Keynote Address). She will present a keynote address at SIGCSE's European analog, the International Conference on Innovation and Technology in Computer Science Education, in Helsinki in July, 2000.

Stein is a 1993 recipient of the National Science Foundation Young Investigator Award. She recently ended a term of service as an Executive Councilor of the American Association for Artificial Intelligence and is the former Chair of that organization's Symposium Committee. She currently serves on several international advisory and steering committees.

Stein has recently returned from a on sabbatical leave from MIT as an Office of Naval Research Science Scholar at the Mary Ingraham Bunting Institute of Radcliffe College, a multidisciplinary think tank in Cambridge Massachusetts.
Acknowledgements

This document would not have been possible without the generous support of the National Science Foundation under Young Investigator Award No. IRI-9357761 and more recently under Educational Innovation Award EIA-9979859. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. Additional support was provided by the Massachusetts Institute of Technology's Classes of 1951 and 1955 Funds and by the Department of Electrical Engineering and Computer Science, by Microsoft Research, and by Sun Microsystems. The initial revision of these notes was supported by the Office of Naval Research through the Science Scholars Program at the Mary Ingraham Bunting Institute of Radcliffe College, where the author was on sabbatical leave as a 1997-98 and 1998-99 Fellow.

The earliest inklings of these ideas probably took root while I was teaching and learning to teach at Harvard. There is no doubt that they were nourished in the dynamic environment of Brown's CS Department, which supported many parallel growths. Both my students and my teachers over the years have taught me more than I can say, and the debt I owe them cannot begin to be repaid. My more recent colleagues, at MIT and elsewhere, have been a source of inspiration and challenge, both of which have strengthened the work immeasurably. Feedback, especially concerning industrial relevance and especially from the networked community, has been invaluable.

A significant portion of the writing of this book was completed in the wonderfully nurturing environment that is Radcliffe's Mary Ingraham Bunting Institute. I could not possibly do justice to what that space and time meant to me. Suffice it to say that the Bunting Fellows of 1997-99 are a most remarkable cohort, in whose debt I will forever remain. I hope that this project will be one more testament to the power of Polly Bunting's campaign against the "climate of unexpectation."

Many people have contributed to the development of these ideas; here I can only single out a few individuals to whom I owe the greatest debts. Early versions of some of these ideas were developed jointly with Jim Hendler, and his continued support has been of tremendous benefit. Hal Abelson has been both mentor and inspiration throughout this project, and I'm sure I don't begin to appreciate the extent of his contributions. Kim Bruce was among the work's earliest (and greatest) supporters. Robert Duvall has been a fellow traveller along this road.

I have had tremendous assistance from the teaching staff who have helped me over the years with the development of this material: Ben Adida, Alfred C. Ashford, Joshua Reuben Brown, Duncan Bryce, Jennifer Chung, Daniele De Francesco, Matt Deeds, Robert Duvall, Mike Harder, Craig Henderson, Stephanie Hong, Pavel Langer, Emily Marcus, Paul Njoroge, Todd Parnell, Ben Pick, Salil Pitroda, Lydia Sandon, Luis F. G. Sarmenta, Emil Sit, Maciej Stachowiak, Ben Vandiver, Mike Wessler, Nathan Williams, and Henry Wong; also Matt Domsch, Carol Lee, Karsten Ulland, and Anne Wright, who helped me with an earlier but crazier experiment.

The students who have participated in the several versions of this course -- 6.80s, 6.75s, conference tutorials, the various versions of 6.096 and 6.030, and those at other institutions whom I have yet to meet -- have left their mark in untold ways upon this material. I am indebted to them for their insights as well as their patience.

© 2003 Lynn Andrea Stein
This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
List of Sidebars

Java Sidebars by Chapter

- A Community of Interacting Entities
  - Sidebar: Selected String Methods

- Things, Types, and Names
  - Sidebar: Java Naming Syntax and Conventions
  - Sidebar: Java Primitive Types

- Expressions: Doing Things With Things
  - Sidebar: Java Operators
  - Sidebar: Arithmetic Expressions
  - Sidebar: Coercion and Casting
  - Sidebar: Java Operator Precedence
  - Sidebar: Other Assignment Operators

- Building New Things: Classes and Objects
  - Sidebar: Java Types and Default Initialization
  - Table: Comparison of Kinds of Names
  - Sidebar: Method Invocation and Execution

- Designing with Objects
  - Sidebar: Final
  - Sidebar: class Math

- Animate Objects
  - Sidebar: class AnimatorThread
  - Sidebar: Thread Methods
  - Sidebar: class Main

- When Things Go Wrong: Exceptions
  - Sidebar: Exceptions, Errors, and RuntimeExceptions
  - Sidebar: Throw Statements and Throws Clauses
  - Sidebar: Try Statement Syntax

- Reusing Implementation: Inheritance
  - Sidebar: The class Object
  - Sidebar: Abstract Classes

Style Sidebars by Chapter
• **Specifying Behavior: Interfaces**
  • Style Sidebar: Method Documentation
  • Style Sidebar: Interface Documentation

• **Expressions: Doing Things With Things**
  • Style Sidebar: Don't Embed Side-Effecting Expressions

• **Statements and Rules**
  • Style Sidebar: Formatting Declaration Statements
  • Style Sidebar: Formatting Blocks
  • Style Sidebar: Using Booleans
  • Style Sidebar: Documentation

• **Building New Things: Classes and Objects**
  • Style Sidebar: Class Declaration
  • Style Sidebar: Field Documentation
  • Style Sidebar: Method Implementation Documentation
  • Style Sidebar: Constructor Documentation
  • Style Sidebar: Capitalization Conventions

• **Designing with Objects**
  • Style Sidebar: Class and Member Documentation

• **Animate Objects**
  • Style Sidebar: Using main()

• **Reusing Implementation: Inheritance**
  • Style Sidebar: Explicit Use of this. and super()

### All Sidebars by Topic

• **Abstract Classes**
  • Reusing Implementation: Inheritance

• **Arithmetic Expressions**
  • Expressions: Doing Things With Things

• **Capitalization Conventions (Style)**
  • Building New Things: Classes and Objects

• **Class and Member Documentation (Style)**
  • Designing with Objects

• **class AnimatorThread**
  • Animate Objects

• **Class Declaration (Style)**
  • Building New Things: Classes and Objects

• **class Math**
Designing with Objects

class Main
  
  Animate Objects

The class Object
  
  Reusing Implementation: Inheritance

Coercion and Casting
  
  Expressions: Doing Things With Things

Comparison of Kinds of Names (Table)
  
  Building New Things: Classes and Objects

Constructor Documentation (Style)
  
  Building New Things: Classes and Objects

Documentation (Style)
  
  Statements and Rules

Don't Embed Side-Effecting Expressions (Style)
  
  Expressions: Doing Things With Things

Exceptions, Errors, and RuntimeExceptions
  
  When Things Go Wrong: Exceptions

Explicit Use of this. and super() (Style)
  
  Reusing Implementation: Inheritance

Field Documentation (Style)
  
  Building New Things: Classes and Objects

Final
  
  Designing with Objects

Formatting Blocks (Style)
  
  Statements and Rules

Formatting Declaration Statements (Style)
  
  Statements and Rules

Interface Documentation (Style)
  
  Specifying Behavior: Interfaces

Java Naming Syntax and Conventions
  
  Things, Types, and Names

Java Operator Precedence
  
  Expressions: Doing Things With Things

Java Operators
  
  Expressions: Doing Things With Things

Java Primitive Types
  
  Things, Types, and Names
• Java Types and Default Initialization
  • Building New Things: Classes and Objects
• Method Documentation (Style)
  • Specifying Behavior: Interfaces
• Method Implementation Documentation (Style)
  • Building New Things: Classes and Objects
• Method Invocation and Execution
  • Building New Things: Classes and Objects
• Other Assignment Operators
  • Expressions: Doing Things With Things
• Selected String Methods
  • A Community of Interacting Entities
• Throw Statements and Throws Clauses
  • When Things Go Wrong: Exceptions
• Try Statement Syntax
  • When Things Go Wrong: Exceptions
• Using Booleans (Style)
  • Statements and Rules
• Using main() (Style)
  • Animate Objects

See also Java Charts

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Introduction to Program Design

- What is a computer program?
- What are the parts of a program? How are they put together?
- What kinds of questions does a program designer ask?

In this chapter you will learn how a computer can be controlled by a set of instructions called a program. This chapter introduces two different aspects of computation: single-minded instruction following and coordination among instruction followers. The programs in this book involve both aspects of computation.

The first aspect of computation is step-by-step instruction following, like the process of making a sandwich. This kind of computation is a sequence of instructions that produces some desired result. The question that drives this part is "What do I do next?" Pieces are put together using "Next,...", "If ... then ... else ...", and "until...". This kind of computation has an end goal to be accomplished by execution of these instructions. The programs in this book use short sequences of instructions, executed over and over, to create entities that can provide services or respond to requests (e.g., a sandwich-maker).

The second aspect of computation involves coordinating among many of these instruction-following entities. This is like gathering sandwich-makers (and table-awaiters and others) together to run a restaurant. This kind of computation is creating (and managing) a community. The driving questions are "Who are the members of the community?", "How do they interact?", and "What is each one made of?" The members of the community -- the instruction-following entities -- are glued together through their interactions and communications. Executing this kind of computation provides an ongoing program such as your car's cruise control, a web browser, or a library's card catalog.

When you finish this chapter, you will know the basic questions to ask about every computational system. These questions will allow you to begin to design a wide variety of computer programs.
1. To understand what programs are made of.
2. To be able to recognize and articulate use cases.
3. To be able to recognize and articulate requirements, assumptions, and (desired) guarantees.
4. To be able to read and construct sequential, conditional, and looping programs using English steps.
5. To be able to design communities of interacting entities using English specifications.
6. To appreciate the difference between a program and its execution.

Computers and Programs

Computers provide services. A suitably equipped computer can retrieve a web page, locate the book whose author you're thinking of, fly an airplane, cook dinner, or send a message to your friend halfway around the world. In order for a computer to do any one of these things, two things must happen. First, the computer must be told how to provide the required services. Second, the computer must be asked to do so.

The how-to instructions that enable computers to provide services are called programs. A computer program is simply a set of instructions in a language that a computer can (be made to) follow. When the computer actually follows the program instructions, we say that it is executing that program. The program is like the script for a play. It contains instructions for how the play should go. But the script itself is just a piece of paper: no actors, no costumes, no set, no action. Executing a program is like performing the play. Now there is something to watch.

This analogy goes further, too. The same script can be performed multiple times, just as the same program can be executed again and again. If audience reaction (or the director's interpretation, or the theater, or the time of day) influences the performance, two performances of the same script may be quite different. Similarly, user input, hardware, software, or other environmental circumstances may make two different executions of the same program quite different from one another. (Think of running the same word processing program on two different occasions; the experiences are extremely different even though the computer follows the same general-purpose instructions both times.)
When you sit down at a computer, someone else has already told it how to do a lot of things. For example, when you press the power switch, it boots up -- gets started running -- in the way that it has been instructed to.

Personal computers typically come with a fairly sophisticated set of startup instructions already installed. Simply turning on the computer causes the computer to execute this startup program. Starting a computer is called "booting it up", presumably from the phrase "pulling yourself up by your bootstraps". The startup program that a computer executes each time that it is turned on is called the computer's "boot sequence". Each computer has a program that it runs automatically. The program that your desktop or laptop PC runs is called its operating system. A disk drive -- which is really a separate computer plus the electronic equivalent of a huge filing cabinet -- comes equipped with instructions for how to retrieve information from (or store information in) that filing cabinet plus how to transmit that information across the cable that connects the disk drive with your "main" computer. A microwave oven comes with a computer that follows instructions for how to tell time and how to turn on its microwave generator for specified periods. The library's card catalog provides lookup services. Your car's cruise control accelerates and decelerates to keep your car moving at a steady rate. A web browser fetches and displays information it retrieves from the hard drives (file cabinets) of computers scattered around the world at your request (with the assistance of the "web server" programs running on those distant computers as well as the network (transmission) services provided by a set of intervening computers.

When you load a new piece of software onto your computer -- a cool new game, for example -- what you are actually doing is giving your computer a copy of the program -- the set of instructions that tells it how to do display graphics and make appropriate sound effects or whatever it is that the particular piece of software does. Writing down these instructions was the job of the person (or people) who wrote the software, the programmer. Loading the software makes the instructions (the script) available to your computer. Just having these instructions lying around doesn't do you much good, though. To actually play the game (perform the play), you need to do one more thing. You need to run the program. Some computer games can be run off of removable media, like CD ROMs. In this case, you don't need to load the program onto the computer, but you do need to make sure that the disk is in the drive, i.e., that the instructions are available to the computer. Also, some programs run automatically when loaded, e.g., applications for browser plugins. Tomorrow, if you want to play the game again, you only have to run it; you don't have to start by loading it onto your computer.

Thinking like a programmer

A computer program -- "how-to" instructions for your computer -- must be written in a language that the computer can follow. There are many languages designed for instructing computers. These languages are called programming languages, and they are typically quite different from the kinds of languages in which people talk to one another. One of the main differences between talking to
a person and programming a computer is the increased level of precision required to tell a computer how to do things. With people, it is often possible to give very vague instructions and still get the behavior you want. A computer has no common sense. You must be very specific with it. Your instructions must be step by step, in great detail. We've already noted that a program -- these instructions -- are like the script of a play. Now, we'll look at an individual's instructions -- one role -- as the kind of step by step instructions you might find in a recipe. In some ways, programming a computer can be a lot like talking to a very young child or a creature from a different planet.

Imagine teaching a Martian how to make a peanut butter and jelly sandwich. You need to give detailed, step by step instructions:

1. Get a loaf of bread.
2. Remove two slices of bread and put them on the counter.
4. Get a jar of jelly. Put it next to the peanut butter.
5. Get a knife.
6. Open the jar of peanut butter.
7. Pick up a slice of bread.
8. Using the knife, pick up a glop of peanut butter and spread it on the top of the slice of bread.
9. ....

These instructions tell the Martian, in very specific terms, what to do. To follow the instructions, the Martian simply needs to perform each step, one by one, in the order given. As long as each of these instructions is one that the Martian knows how to perform, when the Martian finishes executing this program, the Martian will have a peanut butter and jelly sandwich.

If there is an instruction here that the Martian does not understand, that instruction needs to be rewritten in more detail so that the Martian will be able to execute it. For example, "pick up a glop of peanut butter" might require further explanation:

a. Insert the knife blade half-way into the jar of peanut butter.
2. Remove the knife from the jar of peanut butter at a slight angle so that some peanut butter is carried out of the jar by the knife.
3. ....

An instruction that needs further explanation before the Martian (or computer) can execute it is one that we call high level. Java -- the programming language used in this book -- is a high level language. Detailed instructions that tell your computer how to execute Java statements are supplied as a part of your Java programming environment. You can build your own (even higher level) instructions.
in Java. In this case, you'll need to explain them to the computer (in Java). In general, we can use high level steps in our programs only if we can supply additional instructions to explain how to actually execute these higher level steps.

Although we don't know what instructions Martians are likely to understand, a programmer knows what kinds of instructions are a part of the particular programming language in which s/he is developing a computer program. In this book, we will use a programming language called Java. As you read this book, you will learn how to think like a programmer and how to write instructions that computers can understand. You will also learn specifically about the kinds of instructions that are part of the Java programming language.

As a programmer, you will design sequences of steps much like the peanut butter and jelly sandwich instructions. The goal of such a sequence is to get something done, to find an answer or to create something. In order to design a program like this, you will need to repeatedly answer the question, "What do I do next?" until you have reached your desired result. In many ways, this approach makes computers seem much like sophisticated calculators. In fact, this is where computers got their start: the word "computer" used to refer to people who did (mathematical) computations, and the original mechanical computers were designed to perform these computations automatically.

When you are designing a program, you should ask yourself, "What do I do next?" You don't necessarily have to write out all of the basic steps in one long sequence. You can group them together in bigger, more abstract, higher level chunks:

1. Assemble the ingredients.
2. Spread the peanut butter.
3. Spread the jelly.
4. Put the sandwich together.
5. Clean up.

This is a perfectly good set of instructions. But, as in the case of the Martian who didn't know how to "pick up a glop of peanut butter", these instructions will require further elaboration.

This way of thinking about programs -- outlining the big pieces, then breaking each one down further -- is called top down. A programming language such as Java allows you to make up your own high level steps, like "Assemble the ingredients", and then to explain how to do this: "1. Get a loaf of bread...." Your program is complete only when every line is either understandable by the computer or further explained in terms that are understandable by the computer. When you are done asking yourself "What do I do next?" you must then ask "How do I do each of these things?" until every line of your program is something that the computer knows how to do.

### Programming Primitives, Briefly

What kinds of things do computers know how to do? Most computers don't know how to make peanut butter and jelly sandwiches. Most computers do know how to manipulate numbers and also other kinds of information, like words. In the Java programming language, you will find tools that let
you send messages to other computers on a network or create windows and buttons to communicate with people using your programs. Other computers may have special kinds of instructions. A robot control system has instructions that tell the robot when, where, and how to move. A security system may have an instruction to sound an alarm. These are the basic instructions out of which programs for each of these systems can be constructed.

These basic instructions can be combined by sequencing them, as we've already seen. They can also be grouped into mini-programs and given names, like "Assemble the ingredients". These names can then be used as new instructions. When the computer needs to execute one of these new instructions, it simply looks up the rule for how to do it. (When the Martian needs to assemble the ingredients, it uses the detailed instructions that begin "1. Get a loaf of bread....")

Instructions can also be combined in other ways. Sometimes, there is a choice to be made. For example, after spreading a glop of peanut butter on top of the bread (step 8), the next step in the peanut butter and jelly program might say:

9. If the top of the slice of bread is covered in peanut butter, go on to step 10. Otherwise, go back to step 8.

This step contains a choice; the next step might be 8 or it might be 10, depending on whether the slice of bread is full. The Martian (or computer) executing this program will have to keep track of which step comes next. This kind of choice step is called a conditional, and it is a common construct in programming languages. It is especially useful when the answer to the question "What do I do next?" depends on something you won't be able to figure out until you're executing the program.

We might want to go further, replacing steps 8 and 9 with a new kind of step that says

8. Repeat the following sub-steps until the top of the slice of bread is completely covered in peanut butter
   a. pick up a glop of peanut butter
   2. spread it on the top of the slice of bread.

This step ("repeat until") is called a loop. It, too, is a common construct in programming languages. Some loops tell you to keep going until something is true (like the bread becoming full), while others tell you how many times to do the steps inside the loop. Some loops even go on forever. For example, a clock is basically a loop that moves
its hand(s) (or changes its display) once a minute. Loops are especially useful when part of "What do I do next?" is to repeat (almost) the same thing several times.

Sometimes, a collection of steps is so useful that we want to give it a name and treat it like a basic step. For example, the loop that we just wrote -- repeat \{pick up a glop, spread the glop\} until the bread is covered -- is a nice summary of

II. Spread the peanut butter.

When we originally made that list of higher level instructions above, we had to further specify how to do each instruction. Identifying the high level pieces first is top-down design. Now, we're looking at the same problem from the other side. We have identified a set of useful instructions that accomplish the spreading. We want to give this a name (e.g., "spread") and be able to use it again. For example, we might want to use it again to spread the jelly. Identifying the low level pieces and grouping them together is called \textit{bottom-up design}. Giving a name to the group of steps so that we can use it again is a particular kind of bottom-up programming called \textit{procedural abstraction}.

Each of the techniques described above -- sequencing steps, conditionals, loops, and procedural abstraction -- is an important part of building computer programs. You will learn more about how to do these things in Part 2 of this book. These are the pieces that a programmer uses to answer the questions "What do I do next?" and "How do I do each of these things?" But this is only one part of the programming problem. If a program is the script of a play, we have so far been talking about a single actor in a single role, a one-character monologue. Real programs often involve many separate roles. The complete script involves not only each individual player's lines and actions, but also stage directions specifying how the different roles interact. The second part of programming is coordinating the activities of many interdependent participants in a computational community.

\textbf{Ongoing Computational Activity}

Some computer programs are very much like peanut butter and jelly sandwich making instructions. They start with some ingredients and -- step by step -- calculate whatever it is they're designed to create, producing an answer or result before stopping. The original mechanical computers, which mimicked human computers performing mathematical calculations, were very much like this. Sometimes, you would bring your program to a computer operator and then come back the next day for the result!

Today, most computer programs aren't like this. Instead, computer programs today are constantly \textit{interacting}. They may interact with people, machines, other computers, or other programs on the same computer. For example, a word processing program or spreadsheet waits for you to type at it, then rearranges things on the page or recalculates values as you type. A video game moves things around on your screen, some in response to you and others by itself. A web browser responds to your requests, but also talks to computers all across the network. The cruise control system for your car responds to road conditions, sensor readings, and your input. A robot control system interacts with the robot and, through the robot, with the robot's environment, perhaps with no human input at all.
These computations aren't concerned with solving some pre-specified problem and then stopping. Most computations of interest these days are things called servers or agents or even just applications. Most of them have some basic control loop -- a core sequence of instructions to be executed over and over again -- that responds to requests or other incoming information continually. These computations are embedded in an environment and they interact with that environment: users, networks or other communication devices, physical devices (like the car), and other software that runs at the same time.

Not only do programs interact with things around them, a program may also have interactions inside of it. In fact, each of these programs may itself be composed of many separate pieces that interact with each other (as well as with the world outside the program). Coordinating the activity among the many entities that make up your program -- and their interactions with the world around them -- is the second aspect of computer programming.

This is kind of like taking a group of Martians and organizing them to run a restaurant. Some of the Martians will take orders from and serve food to the customers. Other Martians will need to cook food for the customers. Still others will need to check on supplies, make change, or coordinate other aspects of the restaurant's operation. Each of these Martians will provide services to and make request of other Martians (or to the restaurant's customers or suppliers or other parts of the environment in which the restaurant is embedded). Coordinating the interactions among these Martians (and between the Martian restaurant and its environment) involves different kinds of questions from the instruction-following "What do I do next?"

Before we turn to the coordination of activity, though, let's look closely for a moment at one of the Martians who will staff our restaurant. We will see that, deep down, "peanut butter and jelly" programming still has an important role to play in creating computational activity. Keep in mind that this Martian represents just one of the many things going on in our restaurant.

The instructions that a Martian chef follows might look very much like this:

1. Pick up a new food order.
2. Find the instructions for the dish ordered and follow them.
3. Put the completed dish and the order information on the counter for pickup.
4. Go back to step 1.

Step 2 of this program is the kind of "higher level" step that we described above. It is not itself complete; instead, it refers to other, more detailed instructions to be followed. For example, if an order comes in for a peanut butter and jelly sandwich, the Martian chef will need to use the instructions developed in section 1.2 for how to make a peanut butter and jelly sandwich. Like a computer, the Martian is following simple sequenced steps written in a language that it understands. But while this Martian is making a peanut butter and jelly sandwich, another Martian is asking the customer at table 3...
whether she would like some more water. Later, the Martian waiter will come into the kitchen and pick up the sandwich that the Martian chef just made. And when the Martian chef is done making the peanut butter and jelly sandwich, that Martian will turn to the next food order, continuing its ongoing interaction.

The peanut butter and jelly style of program instructions is an important part of how the Martian chef does its job. But the Martian chef's instructions are not simply the steps of the peanut butter and jelly program. The basic structure of the Martian chef program is an infinite loop -- a loop that goes on forever. This program accepts requests (in the form of new food orders) and provides services (in the form of the completed dishes) over and over again. We sometimes call this kind of loop -- one that provides the main behavior for a participant in the interactive program community -- a control loop. Many program community participants take this form, and we will look more closely at control loops in Part 3 of this book.

Programs with ongoing central control loops like this are the members of our interactive computational community.

Coordinating a Computational Community

At its most basic level, every computer program is made of instructions that are followed, one by one. But a single computer program may have many instruction-followers inside it, just as our restaurant is run by many individual Martians. When you look at the whole program -- like the whole restaurant -- you don't necessarily see the individual instruction steps. Instead, you see coordinated activity among a group of interacting entities. The behavior of this community -- e.g., providing customers with hot meals -- is not the responsibility of any particular member of the community. Instead, it is the result of many community members working together in a coordinated fashion.

Building modern interactive software involves something very much like organizational design. We call this part of programming "constituting a community of interacting entities". The programmer's job to figure out how to tell the computer what to do, and no matter what the specific problem to be solved may be, there are fundamental questions that each programmer must ask. Designing a computation which is a community of interacting entities involves figuring out who the members of this community are, how each one works, and how they interact. This is like setting the cast of a play, or deciding what the sub-units of your business will be, as well as how they should interrelate. In planning the organizational structure of your business (or program), you also have to figure out how each unit works and what -- and how -- they are supposed to communicate. These are the big questions of this second aspect of programming.

When you are designing this kind of activity, you ask yourself several questions:

- What is the desired behavior of the program?
- Who are the entities who interact to produce this behavior?
How does each one work?
How do these entities interact?

In the remainder of this section, we will expand these questions and begin to explore them in somewhat greater detail. Understanding these questions and their ramifications is the theme of this entire book. Coordinating communities is a special focus of Part 4.

What is the desired behavior of the program?

Before you can design a system to solve your problem, you must know what your problem is. This involves knowing not only what you want, but how it should work or fail to work under a variety of different circumstances.

One way to determine what your program needs to do is to envision how it will look to someone who interacts with it. A person who interacts with a computational system is called a user of that computational system. A description of a particular interaction between a user and a computational system is called a use case. One computational system can also be thought of as a user of another, and one computational system's experience interacting with another is a sort of use case as well. A good use case includes

1. the prerequisites: what must be true for the use case to arise
2. the set of possible actions and interactions, perhaps described in terms of conditionals and loops
3. the effects: what changes are made to the computational system and to the user as a result

For example, a typical use case for a restaurant might be: walk in, sit down, get the menu, order food, be served, pay, leave. The (usual) prerequisites are that the customer is hungry and has money. The postconditions are that the user is less hungry and has less money, while the restaurant has more money and less food (not to mention some dirty dishes). Another use case for a restaurant might be: walk in, find out that the wait for a table is too long, leave. In this second use case, no money changes hands.

Use cases are a good way to identify the behavior and interactions that you expect from a system. Use cases can be specified with varying degrees of formality. In this book, we use a relatively casual format for specifying use cases. For a more rigorous approach, see @@.

Some questions that you ought to be able to answer about your desired program include:

- What services should your program provide?
- What guarantees does your program make about these services?
- Under what assumptions (circumstances, conditions) does your program make these guarantees?

What can we say about the behavior of the restaurant? In answering this question, we consider both the experiences of individual customers -- reflected in use cases -- and the ongoing properties that the restaurant must maintain, such as remaining solvent. A basic specification of the service provided by the restaurant might be: Each customer is seated at a clean table, the order is taken, food is served, a bill presented, and payment collected.
There are a number of guarantees we want to make about these services. For example, customers should not have to wait for an unduly long time. Different parts of the restaurant must communicate; customers should not be charged for food that they were not served, etc. Over time, the restaurant should take in at least enough revenue to cover its operating expense. Supplies should not run out, nor should they rot.

We will make certain assumptions in order to be able to provide these guarantees. For example, the "timely service" guarantee will only be possible if the customer load on the restaurant is reasonable. We might decide that we will only be able to uphold this guarantee if the number of people wanting to eat in the restaurant at one time never exceeds its capacity, and if the rate of arrival of these people doesn't exceed the rate at which the restaurant can serve them. How many customers the restaurant can handle at the same time is called its bandwidth. How quickly each one can be served is called its latency. The number of customers per hour that the restaurant can handle is its throughput. These quantities -- bandwidth, latency, throughput -- are common measures of program performance. These assumptions should be made explicit, and we will also need to say what happens when they are violated. (In this case, the timely service guarantee won't be upheld, but how slow the service gets should be related to how overloaded the restaurant is.)

There are other assumptions we do not make about our program, and we can articulate these as well. We do not assume that only one customer will be served at a time. Instead, we expect that multiple tables must be handled (roughly) simultaneously. It certainly won't do to wait until the first customer has eaten, paid, and left before addressing the second. We also permit different interactions with each table to be handled simultaneously or at least overlapped; food may be cooking while checks are being written up.

This description is still fairly general, and we can imagine making it more specific. (For example, are customers constrained to ordering off of a menu?) In general, the more detail you can give of what your program ought to do, the easier your task will be in designing and building it.

Designing the community's members

The previous section was concerned with understanding the desired behavior of our program as a whole. In this section, we look at how the community is built. As with single recipes, sometimes whole programs are designed top down, first figuring out what the program should do and then breaking it into separate interacting roles. At other times, you will begin with pieces -- bottom up -- and combine them to create a community that accomplishes what you need. But whether you start at the top or at the bottom or do a bit of each, you will eventually need to answer the question of who are the members of the community?

This question can't be answered in isolation, because each and every decision you make about who the entities are is also at least a partial commitment to what they are and how they work. So answering this question is in many ways like solving the whole problem. The trick is to answer this question in fairly high-level, general terms, then to sit down and try to figure out the answers to all of the what and how questions. In answering those, you'll almost certainly have to return to this question and rearrange your answer a few times. This is fine; it's even typical enough to have a name: incremental program.
design

How do we know how to divide behavior? Often, we do so by enumerating coherent sets of resources and grouping them. The kitchen -- the stove, the food, the pots and pans and plates -- provides the impetus for one community role. The customer -- getting orders, delivering food, etc. -- motivates another. But there is more than one way to divide responsibility. A good division is one that makes the next question -- *how do the community members interact?* -- easier to answer.

In the restaurant, an appropriate high level division of labor might have a wait staff unit (the people who deal directly with the customers), a kitchen staff unit (the people who cook the food), and a financial unit (who keep track of how much which things cost, collect money, and buy supplies). At this point, we haven't committed to whether these are three roles played by a single Martian, three separate Martians, or even three groups of several Martians each. The final question we'll need to ask is *what is each one made of?:* How is each member of the community built?

**How do the community members interact?**

This question concerns coordination and communication among two or more entities. In many ways, this question is just our original question -- *what is the desired behavior of the program?* -- all over again. Program behavior was determined by looking at interactions between a program and the environment in which it is embedded. Now, we are asking about the behavior of a part of the community. Use cases can be a good tool to help understand within-community interactions as well.

Each entity in your program is a miniature system unto itself. Some of the questions that you should ask about how these entities interact include:

- What are the entities' interfaces?
  - What promises does each one make?
  - What contracts does it fulfill?
  - What services does it provide?
- How do they communicate?
  - What mechanisms do they use?
  - What interaction patterns do they use?
  - How do they preserve liveness, i.e., make sure that things keep moving?
- What interaction patterns are possible?
- What happens when something goes wrong?

Interactions are often described in terms of protocols. A *protocol* is the specification for an interaction between two entities. For example, a common protocol for the interaction between the wait staff and kitchen staff of a restaurant involves a slip of paper with the customer's order written on it. The waiter hangs this piece of paper in the window over the kitchen's food pickup counter, a place where it will be easy to find when someone from the kitchen is ready for a new job. When a member of the kitchen staff is ready to process the order, the piece of paper is removed and used to guide the food preparation. When the order is ready, it is placed on the food pickup counter together with the original
order slip. This identifies the food with the original request when the waiter returns to retrieve it. The slip of paper serves as a crucial reminder of several associated pieces of information: what was ordered, by whom, and where they are seated.

This is a data structure based protocol; it involves the use of a separate piece of information to keep track of what is happening with each order. The state of that piece of information -- where the paper order slip is -- tells the kitchen and wait staff what they need to know about how far along the order has gotten and whether it's ready. Contrast this with a simple request/response protocol in which the waiter stood at the entrance to the kitchen until from when the order was placed to when it was completed. In this protocol, the waiter can't do anything else until the kitchen has responded to the order request (by producing the cooked dish). Obviously, these two protocols will lead to restaurants that run very differently from one another.

Protocols can also address temporal issues. For example, the wait staff/kitchen staff interaction described in the preceding paragraph needs to happen in real time, meaning that the protocol itself can't introduce significant delays. There must also be guarantees made about the frequency with which the wait staff checks for completed dishes (or the kitchen staff for incoming orders). If assumptions such as these are built into protocols, they must be documented so that they are maintained in the behavior of participant entities.

Not all interactions are real time. For example, the wait staff interacts with the financial unit by obtaining prices for food and turning over any moneys collected. These interactions could happen in batch, meaning that it is OK for the wait staff to get the price list at the beginning of the week or for money to be handed over at the end of the day. Batch processing is like the old-fashioned computations in which you handed your program to a computer operator and came back the next day for your results. The difference between real time and batch interactions is only one dimension that must be determined in order to coordinate the activities of the members of your computational community.

A protocol specifies the interface, or meeting, between various entities in the community that constitutes your program. These interfaces are often captured in contracts that spell out the behavior that will be provided. Once the interfaces have been thoroughly fleshed out, each entity can in theory be implemented by a separate programmer (or team of programmers) provided that it is built to spec, i.e., that it meets the specifications of the agreed-upon interface.

What is each member of the community made of?

Finally, we look inside the members of our community. When we do so, we may find that a particular community member is itself a community of interacting entities. Or we may find that it is a relatively simple entity, performing a single task over and over again. Just as in sequential instruction-following, the most basic building blocks are combined to build higher level components, which can then be combined again and again to build increasingly more complex systems.

At the top level, before we could talk about who the community members were, we needed to know what the community would be doing. Similarly, before we can build a single constituent entity, we need to know how it's supposed to fit into the community. So answering the question of what goes inside?
requires knowing something about how they interact. Before you can design the structures that go inside, you need to be able to specify what the community member will do. After all, specifying what interactions each entity needs to support goes a far way towards telling you whether whatever goes inside meets the requirements of the community.

Some subsidiary questions to ask about how each of the entities is constituted include:

- What responsibilities does it have?
- What guarantees (promises, commitments) does it make? Under what assumptions?
- What resources does it control?
- How does it work?
- Is it a community, too?

What responsibilities does it have? The restaurant's wait staff might be responsible for greeting the customers in a timely fashion, supplying each one with a menu (a structure that the program will have to provide and keep updated!), taking the order, delivering it to the kitchen staff, picking up and serving the cooked meal, obtaining a price from the accounting entity, and obtaining payment for that amount from the customer.

What guarantees does it make? The wait staff might guarantee to communicate with (most of) the customers within minutes, provided the total number of customers is limited and the maximum time spent with each is under a certain amount. It might also promise to deliver food within some small amount of time after it's done cooking, provided that the kitchen staff notifies the wait staff in a timely manner.

What resources does it control? The wait staff controls menus, knows which food items were ordered by which customers, and is the only part of the restaurant that deals directly with the customers. And so on.

When it comes to how does it work?, there are two kinds of answers. One answer is that the behavior of the entity is accomplished by a single rule-follower running an interactive control loop. We saw an example of this when we considered the Martian chef earlier. In this case, we ask "What does the Martian do next?" over and over, until we wind up with a well-defined set of instructions for this Martian to follow.

The other possible answer to the question how does this entity work? is that this entity is itself a community. The wait staff might be further divided into the person who takes the order, the person who clears the table, and the person who serves the wine. In this case, we need to figure out how to build each of these entities, asking again what goes inside each one? The problem of figuring out how to coordinate the activity of a community continues until each community member is a single (rule-follower) Martian. Then we ask about the instructions this Martian follows, using the vocabulary of section @@1.3.

In practice, the task of implementing an entity to match a given specification often results in
questions about or revision of that interface. Programming is not so neat a task as students of computer science would often like to believe; there's a cycle of specification and implementation, debugging and testing, usage and revision, that characterizes almost all real-world software. The later stages of this process are sometimes called the software life cycle, but the repeated revision that characterizes those later stages start before a piece of software is even born.

The Interactive Control Loop

This book focuses on the problem of designing, building, and understanding interactive software. The sections above concern basic elements in the design of a computer program. In the next chapter, we will look at the process by which a piece of software actually comes into existence and what happens to it over its life cycle. Although this chapter has focused on design from scratch, most professional software engineers spend most of their time modifying or building on existing code.

Regardless of your particular design problem, you will find it useful to situate your task in the context of these six questions:

- What is the behavior of this program?
  
  If it is a community of entities,
  
  - Who are the entities that combine to produce this behavior?
  - How do they interact?
  - What is each one made of? (A community of entities or a single instruction-following control loop?)

  And, for each instruction-follower,
  
  - What does it do next?
  - How does it do each one of these things?

  At the heart of our approach is the idea of an interactive control loop. This is a simple program that repeatedly receives an input -- a new request, a set of sensor readings, or some other information -- and responds appropriately. An interactive control loop can be just that simple, or it may involve initiating a series of other activities. This kind of program component can be built upon and coupled together to make extremely complex computational systems. Whether simple or complex, the interactive control loop is the crucial bridge that turns an instruction follower into an ongoing participant in an interactive community. In a way, it might be thought of as the "atomic unit" or basic vocabulary element of this kind of computation.

  In concluding this chapter, we will look at one -- extremely simple -- interactive control loop. We will keep this example at hand throughout much
of this book and return to it from time to time to explore new variations and themes as we explore interactive programming.

Perhaps the simplest interactive control loop is an **echo** program. When run, this program waits for the user to type something. When the user finishes typing, the program simply repeats back what it has been given. That is, it is a loop that gets some input, processes that input (in this case trivially), and then spits out its result. It is roughly the computational equivalent of a mirror.

Although the echo program seems too trivial to be of much use, a minor variant of it runs in almost every program you type to: it is what makes the characters appear on the screen. In fact, it is what is "turned off" when you type your password, so that no one can see what you type. The setting for not showing what you type on the screen is sometimes called "password mode" and sometimes called "no echo". Far more importantly, the basic structure of this program underlies essentially every interactive computation. And it demonstrates many of the important properties of an interactive computation:

- **It is embedded in an environment** (in this case involving a user's typing and a display that the user can see).
- **It is interactive** (with that user).
- **It is concurrent**: other things happen at the same time that the program is running. (In this case, the user might be typing the next line even while the echo program is producing its output.)

Let's look at the basic echo program in terms of our six design questions:

- **What is the behavior of this program?**
  It reads in a line from the user and writes that line back to the screen.
- **Who are the entities that combine to produce this behavior?**
  From the outside, it looks like one entity, the echo program. So let's implement it that way. This isn't the only way we might do it, but it's a perfectly reasonable one.
  It is reasonable to implement the simple echoer as one entity because the job of that entity can be succinctly described and easily implemented. If the job of that entity were to be too complex, we would want to break it down into smaller pieces even though it might look to the outside world like a single entity. This is a design decision that is more art than science, though some of the principles behind it are described in various places throughout this book, but especially in the chapters focusing on object-oriented design (8 - Object Design, 13 - Encapsulation, and 14 - Intelligent Objects and Implicit Dispatch).
- **How do they interact?**
  Since there's only one, it doesn't interact with anything except the user.
- **What is each one made of?** (A community of entities or a single instruction-following control loop?)
  The one entity -- the echoer -- is an instruction-following control loop.
- **What does it do next?**
  There are two steps:
1. Read a line from the user.
2. Write that line to the screen.

- **How does it do each one of these things?**

  Each of these instructions is a built-in Java method, i.e., something that Java knows how to do. The specific Java instructions are summarized in the sidebar on the cs101 Console in chapter 3 (Things, Types, and Names).

Now let’s look at a slightly more complex design for the echo program. Instead of having one community member -- a single interactive control loop -- we will separate it into a community of two: one that reads the input from the user, and the other that writes the input to the screen. Now, our six design questions will have slightly different answers:

- **What is the behavior of this program?**

  Still the same as before: it reads in a line from the user and writes that line back to the screen.

- **Who are the entities that combine to produce this behavior?**

  Now, we have two entities: the reader-in and the writer-out.

- **How do they interact?**

  There are several choices we could make here. For now, we’ll let the reader-in tell the writer-out each time the reader-in gets a new line. We will explore other answers to this question in part 5 of this book.

- **What is each one made of? (A community of entities or a single instruction-following control loop?)**

  Now, each of our two entities is instruction-following control loop.

- **What does it do next?**

  The reader-in has two steps:
  1. Read a line from the user.
  2. Tell it to the writer-out.

  And the writer-out has two steps:
  1. Wait for the reader-in to give you a line.
  2. Write that line to the screen.

- **How does it do each one of these things?**

  Again, the instructions are basic things that we can write directly in single lines of Java code.

Of course, this example may seem sort-of forced to illustrate how an echoer can be broken up into pieces and turned into a community. But the division isn’t entirely artificial. You see, now that we have separated the echoer into the reader-in and the writer-out, we can do more.
For example, we can run the reader-in part of the echoer on your computer and the writer-out part on mine. Now, whatever you type on my computer is echoed not on your screen but on mine. Java provides some pretty simple tools to let the reader-in on your computer talk to the writer-out on my computer. See Streams, which are covered in chapter Streams. almost as easily as it can talk to the writer-out on mine. This new, improved echoer is actually doing instant internet messaging!

To really make this idea work, we will of course want to add more features. You should have some control over whose computer you are typing to. I should have some say over whether I want to get instant messages across the internet. And there are plenty of nice graphical user interface features -- windows and buttons and other ways to make the program nicer -- that we probably want to add to our reader-in and to our writer-out. Each of these amounts to changing the way we construct our community, the pieces that interact, and what each one of them is made of, in ways that we will explore through the rest of this book.

The idea of an interactive control loop is the root of this approach to programming. By putting together interactive control loops, you constitute a community of interacting entities. Interactive control loops are what goes inside; communication between them is how they interact. In other words, as they say, all the rest is corollary....

---

*needs revision to bring it up to speed with the current version of the text*

- Computers follow special instructions, called a program, which is written in a special programming language.
- Computation results when a computer has access to these instructions and executes them.
- Each set of instructions must answer:
  - What should the program do next?
  - How should it do it?
- Groups of steps can be combined to make a "higher order" step.
- Steps can involve choices or decisions.
- Steps can be executed over and over again using a loop.
- Most modern programs combine many separate looping instruction-followers into an interacting community.
- Every computation is embedded in an environment and interacts with the other (computational and physical) entities around it.
- The programmer's job is to figure out:
  - What services (behavior) does my program provide?
  - Who are the entities that together provide this behavior?
How does each one work?
How do they interact?
• use cases

1. Give step by step instructions for how to tie shoelaces.

2. Select your favorite recipe and give step by step instructions for how to cook it.

3. Give detailed directions for how to get from your classroom to where you live. Include indications that will tell whether you've gone too far and how to get back on track.

4. Specify the expected behavior for each of the following interrelated services provided by a bank account:
   a. A deposit.
   b. A withdrawal request.
   c. Checking your balance.
   Does your specification permit overdrafts?

5. You are at a fruit market. Describe the protocol by which you purchase a piece of fruit from the fruit seller.

6. Describe the division of responsibility and coordination of activities among the players on a soccer team.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.
Questions or comments:
<webmaster@cs101.org>
Chapter 2.
The Software Development Process

Chapter Overview

- What does a software developer do?
- How does the programming process work?

This chapter builds on the previous one by walking through an example of interactive program design. It reiterates the central questions a programmer asks: What is the overall behavior of the program? Who are the members of the community? What goes inside each one? How do they interact? In addition, it stresses the ideas of incremental construction and testing (start with simple functionality and add on only when the program is working); repeated cycling through designing, building, testing, and back again; and the necessity of modifying and maintaining software on an ongoing basis. The example in this chapter is presented in English rather than in actual Java code; it is intended to introduce students to the idea of programming, to the processes involved, and to the kinds of questions that they will be asking throughout this book.

This chapter differs from the remainder of this book. The rest of the book concerns what you need to know to write programs. This chapter is about the actual experience of doing software development. It provides a context of use for the rest of the book. After all, there is no better way to learn to develop software than to do it, and programming requires a lot of practice. In the remainder of the book, you will learn the things you need to know to get the computer to perform certain tasks. In this chapter, you will learn how to work with the computer to apply that knowledge. Most importantly, this chapter describes the experiences that you will have in working on the programming laboratories that should accompany your use of this book.
Objectives of the Chapter

1. To understand the development cycle, its stages, and their interactions
2. To increase ability to recognize and articulate use cases from a problem description
3. To increase ability to recognize and articulate problem requirements including needs, constraints, and resources of the user, physical environment, and software environment
4. To increase ability to articulate appropriate assumptions and guarantees inherent in a design
5. To be able to use an engineering notebook to track the development process
6. To decompose problem requirements to identify the components (entities, objects, and actions) of a software solution
7. To recursively decompose these components (entities, objects, and actions) until individual recipes are reached
8. To be able to test a design by acting it out
9. To be able to write a staged development plan for constructing a piece of designed software
10. To design tests for various stages of this development plan
11. To understand the process of editing and compiling source code and running the compiled code; to understand when recompilation is necessary
12. From tests, to identify the presence of bugs and to propose strategies to locate and resolve them
13. To use printing statements, debuggers, and interpersonal interaction to locate and resolve bugs

2.1 The Development Cycle

The previous chapter explored what programs are made of. In this chapter, we'll look at the process by which programs are created and what happens to them as they continue to grow and change. Software development -- creating, modifying, and maintaining computer programs -- is often the job of a software engineer.

2.1.1 Software Development

In the previous chapter, we used six questions to think about program design:
- What is the behavior of this program?

  If it is a community of entities, we need to figure out how it is put together; we need to decompose it:

  - Who are the members of the community, the entities that combine to produce this behavior?
  - How do these community members interact?
  - What goes inside each one? What is each one made of? (A community of entities or a single instruction-following control loop?)

  And, for each instruction-follower, we need to write its recipe:

    - What does it do next?
    - How does it do each one of these things?

  As you design your program, answering these six questions, you will likely find that later decisions involve going back and modifying earlier parts of the design, changing them or specifying them in greater detail. You will probably also discuss your design with other programmers -- or, perhaps more importantly, to the users or customers for whom you are creating this service -- and revised your design specification in response to their feedback. As you have answers to these design questions, you can start to build your program (or at least a simplified version of it).

  The implementation phase of the project is similar. In building a program that is supposed to meet your specification, you will often find that you need to go back and change (or at least add details to) that specification. When this happens, you need to be careful to consider all of the interdependencies that led you to your original design. That is, the development of software is cyclic, beginning with design but often returning to it. It will not always be desirable (or even possible) to change your design, but it is quite common to discover additional assumptions or nuances that must be percolated through the design during later phases of development.

  When you begin to build your program, it is advisable to implement only a small piece of your system first. This may mean implementing only some of the entities, or it may mean implementing all of the entities but only simple, basic versions of each. In large scale system development, this initial phase is called prototyping. For example, you may build a restaurant in which there is only one thing to be ordered. [Footnote: No coke, pepsi.] Building a simple version first lets you see that you have gotten the basic structure right. As you get this version working, you can begin to add more complicated features -- such as varying what is ordered, making sure that the waiter can handle a variety of different requests -- one by one.

  Even in most of the smaller scale programs that you will encounter in your early course work, it is a good idea to utilize this approach of incremental program development. Part of developing good programming
skills involves learning to consciously and explicitly design a staged development plan in which smaller simpler programs are constructed and debugged, then gradually expanded until the desired functionality is obtained.

Building a simpler version of your system gives you an opportunity to test your basic approach before you have built up too much complexity. It also means that your bugs, or program errors, will be easier to find. Bugs come in many flavors, ranging from simple syntactic errors such as spelling mistakes, to programming errors such as incorrect variable scoping, to conceptual design problems such as impossible-to-meet but critical guarantees.

Even after you've found the bugs that keep your program from running, you will need to subject your code to rigorous testing. This means trying out not only the "normal" expected behavior, but also checking how your program handles unexpected or anomalous behavior. Think of your program as an opponent you're trying to trick; see if you can get it to misbehave. This testing -- when done right -- will lead you to modify your code or even your design.

2.1.2 Software Lifecycle

How, then, does a programmer provide for this behavior? Software development is an intertwined process of designing, building, and testing. Each of these elements provides feedback to earlier phases of the development process. During the lifetime of a piece of software, the requirements that first shaped it will change and as they do so, the design and implementation of that software will need to change, too.

For example, the SmallTown library may decide to automate its catalog and circulation system. The new system should keep track of what books the library has by author and title as well as which books are checked out to which patrons. How does this software come into being?

Some people think about programming as though the goal were to produce a working piece of software. They will describe what a programmer does as a step by step recipe intended to create this result.

1. Get requirements from user(s).
2. Design solution.
4. Test solution.

...producing a finished program.
This description of the programming process should sound reminiscent of the peanut butter and jelly model of program behavior. Just as making sandwiches -- producing results -- is an important part of program behavior, producing programs is an important part of a software developer's job. But a software developer is more like a restaurateur than a peanut butter and jelly producer, and producing a program is not the whole job. In fact, software development itself is an ongoing, interactive process.

This list of steps in the process of software development is sort-of right. The first part of building software is understanding the requirements that software will need to meet. A software developer producing the SmallTown library system would need to understand the properties of books and library patrons that should be tracked, the kinds of access librarians and the public need to the system, the types of reports on circulation that library administrators want, etc. These are the use cases of a library system, and use cases are always a good place to start.

But the list of steps makes it look like each of these pieces of software development -- understanding requirements, designing, building, and testing -- happens on its own, in sequence. In fact, the different pieces happen in an ongoing, overlapping, interrelated way. For example, after sketching a preliminary design for the library system, the software engineer might bring this proposal back to the SmallTown library administrators to see how well it fits their requirements, even before beginning to build the system. The software construction phase might begin by building a very simple system that allows a library staff member to enter book or patron data. Discussion of this system -- among designers, developers, users, and management -- might lead to a redesign in which different pieces of data could be entered and edited by different staffers at different times.

As the initial prototype is built, each component and phase will need to be tested. Some of these tests will result in additional building; a few of them may even send the software engineers back to SmallTown to discuss requirements and design further. When the design seems settled, the software engineer can build a production version, but even then customer feedback -- and changing library needs -- may modify the software as it is built and tested. As the program is under construction, the library staff may come up with new benefits they'd like to see from the software, or the software engineer may be able to provide the library with additional flexibility, by letting building and testing influence design.

This scenario -- modification of an existing piece of software -- is actually the norm. Development of new software from scratch is the exceptional case. But even in that software developer use case -- the one for which the peanut butter and jelly recipe seems to be an answer -- there is more interaction between design, building, and testing than might initially be apparent. Even in relatively simple software, it's common to build a "quick and dirty" prototype that can be used to influence design decisions. As you see how that software

![Design, Build, Test Diagram]

Figure 2.2. Design, build, test, always with feedback to earlier phases; each is an ongoing process.
works, you can modify it -- add features, change its behavior, make it more complex -- and "grow" the design. Eventually, you may learn enough about the software solution that you start from scratch and rebuild the system, but in building and testing you have learned what you needed about how to design this piece of software. As the software increases in complexity, these steps become increasingly intertwined, so that the actual implementation of the software development process may become more like an interactive community.

If the library system built for SmallTown succeeds, it's likely that the SmallTown library will want to extend its functionality. For example, the library might want to give its patrons access to the catalog over the web. This involves creating a web interface for a previously in-house custom piece of software. It might also be nice to allow a patron to reserve a book from home or even to ask the library to buy a copy of a particularly interesting book. These are wholly new functions not present in the previous system, but that should be fully integrated with that system and take advantage of its existing data. These modifications and adaptations are an important part of the software life cycle.

Shifting requirements are a reality of software development. Sometimes, they are a result of an inadequately developed initial design. Often, they are driven by the changing world within which software is embedded. Consider the web browser. Originally, it was a relatively simple program for retrieving and displaying relatively simple (html) web pages. Over the first few years of web browsing, this requirement shifted slightly to encompass somewhat more sophisticated on-line material. Then, the web took off. Now, a web browser must support text and graphics, "plug-ins" (specialty programs that handle a wide variety of multimedia and other additional functionality), and even an interpreter for one or more programming languages (typically Java and JavaScript at this writing). The simple requirements of a mid-90s web browser have been transformed.

Spurred on by her amazing success with SmallTown's library system, SmallTown's software developer has been named Chief Technology Officer for newly formed Local Area Regional Library Consortium and spends most of her time meeting with government officials. She's recently gotten the Consortium members to create an aggressive five-year plan to create a joint regional library computer system based on the SmallTown software. A new software engineer--hired since the formation of LARLC--needs to understand how each the SmallTown software works; how it keeps track of the books in the collection and records patrons and checkouts; how it could be expanded to handle multiple collections at multiple (member library) sites; and whether any of the necessary adaptations for the Consortium will break the existing library system. Naturally, the CTO has no time to help him figure all of this out.

Although it is important to write code that works correctly, it can be even more important to write code that other people can understand. It turns out that far more time is spent revising existing code than writing new code. By writing your code well in the first place, you can make the job of maintaining and augmenting your code much easier.

Sometimes, you want to include information in your code that is intended only for people (e.g., readers of your code), not for the computer. You can do this by including whatever information you want in
Comments. Comments are parts of your Java code that are not read by the computer, and the details of how to put something into a comment are included in the sidebar on Java Comments.

Documentation, or comments, are an important part of code-writing. Documentation is designed to help people read and understand your code. In spite of the running joke among overworked programmers, code is really not self-documenting. Learning to write good documentation may be even more important than learning to write good code. (This is particularly true since far more time is spent fixing, maintaining, and revising existing code than was spent in writing it in the first place.)

2.1.3 Software as a Process

The big question of this chapter is: How does software come to be? Earlier, we said that some people think of software development as a sequence of steps -- a recipe -- but that it is really much more like running a restaurant. To see this, let's think about the software development process as we just stepped through it and try describe the requirements of the software process. This will help us design a description of the job of a software engineer. We can use the idea of use cases from the previous chapter to help us think about this problem.

By far the most common use case for a software developer is a customer who has an existing piece of software and needs changes made to it, like the new hire who had to grow the SmallTown library system into a system for the library consortium. This often happens because the customer needs additional functionality from the software: to handle new kinds of information, produce new reports, work with another piece of software, run on a different computer platform, incorporate additional sites. Generally, the program was not originally developed by the person expected to modify it. Frequently, many people have worked on the software over time. Even understanding this kind of pre-existing legacy software can be a substantial task. (This task can be made easier if if proper care was taken by the original and subsequent developers to design for modifiability, i.e., construct the program with future software developers and their potential tasks in mind.)

Other software development use cases are variants of these: fix (debug) existing software; build new software starting from two or more separate preexisting software components; understand a piece of software (e.g., to extract its principles for future use); test existing software in a new context; translate (or port) software from one programming language to another or from one platform to another. Most of these use cases can be summed up in the notion of maintaining a piece of software -- keeping it functioning as bugs are discovered, hardware and software needs change, new functionality is needed, etc. Rather than an end result, there is an ongoing property to be maintained -- functioning software -- and there are a set of services available to modify, augment, and improve the software. By now, the job of a software engineer -- this collection of use cases -- should sound a lot like the restaurant model of processes. It is measured in terms of the ongoing adequacy of the software. The software itself must be cared for and developed much as a living, breathing thing. Thus, the name for this process [@@insert correct phases] is software life cycle. [@@stats]
"Correct" software is a moving target. Requirements change. Software is never "done" for all time. Refinement is an ongoing process. Software should be designed, built, documented, and tested for ongoing improvement. Software built using the peanut butter and jelly notion of a correct answer -- code that is complete -- is applicable only when the produced code can safely be discarded after that use. (Even when you're certain this is the case, you'd be surprised how often it turns out that you need the software again.) Software that continues to be used will also continue to need maintenance and growth.

This repeated cycling through and between the various stages of specification (or design) development, implementation, and testing is a crucial skill for any good programmer. Classroom programs are too often written once and tested on obvious cases. Most of the time and money spent on real-world software is spent on revision and maintenance rather than on initial development. Acquainting yourself with this cycle -- and with writing clean, easy-to-read, reusable code -- may be the most important part of becoming a skilled programmer. These issues -- together with a tour through the development cycle -- are the main topic of this chapter.

2.2 Understanding the Problem

In the next few sections, we are going to step through the design of a library system like SmallTown's, including a computerized catalog and checkout system. Why a library system? First, libraries are probably things that you have experience with. The work of the library system we'll build is useful, but not too complicated. And this example illustrates many of the important stages and issues, so hopefully it will give you insights into what you should be doing and how.

We will use the questions of the previous chapter to flesh out the major portions of the design of this system. We will construct this system in English, not in Java, because this chapter does not presume that you know any Java yet. Since we don't have computers that run English, we can't actually execute the program that we build in this chapter. Also, there are aspects of the complete system that we will not get to in this chapter. Still, you should be able to understand how the program works by the time that this chapter is done.

We will also use the problem of designing the library system to explore the process of design and programming itself: understanding the problem, designing a solution, building the system, and testing its behavior. In the remainder of this book, we will explore the conceptual structures of which programs are built and their pragmatic implications. We will not spend much time, in the text outside of this chapter, looking at the larger process of developing software. There is, however, no way to learn to develop software without doing it. In the laboratories that accompany this book, you will have opportunities to build programs of your
own. This chapter is intended to give you the context and background to apply what you learn in the remainder of this book to those laboratories and to software that you develop.

Later in this book, we will return to similar extended examples in segments that sit between chapters, called interludes. Each interlude focuses on a single extended example to illustrate the principles and practices described in previous chapters of the book and to ground them in a concrete example. In those interludes, but not in this chapter, actual working programs will be developed.

As you step through the various stages of program development -- understanding the problem, designing a solution, building the system, and testing its behavior -- you should keep track of your work, what you discover, and what you decide at each stage. You may want to do this in a physical notebook -- ideally a bound notebook, in which you do not remove pages -- or you may prefer to use a computer file. In either case, you should use this notebook to record things but not to erase or delete them. If you make a decision and later decide that it was wrong, it is important to preserve the original decision and the reasons behind it as well as the explanation of why you changed your mind. This is also a good place to record ideas you have about extensions or features you might add to the program or concerns you have about problems that might arise. You should date each entry. Whether it is a physical notebook or a set of computer files, we will refer to this as your engineer's notebook.

2.2.1 What behavior do we expect?

Before we can build a program -- or even begin to design it -- we need to know what that program does. This is the "desired behavior of the program" question. What might we expect from a library's computer system? You should record your answer in your engineer's notebook.

The primary users of the library system will be people like you and me who want to check out books. We will make requests of the library for books by specific authors, with specific titles, or on specific topics. We may know exactly what book we want, or we may need to find further information before we can select a book. Once we have identified the book that we want, we will need to check it out and later to return it. This library system has two main pieces: the catalog, which allows users to identify books, and the check-in/check-out system, which transfers responsibility for books between individual users and the library. The book itself is a physical object -- it will not be in our program, although some information representing it will be.

In more detail: If I want to get a book from this library, the first thing that I will probably do is to go to the catalog station and look up the book I want. This catalog should show me a screen that asks what I'm looking for. When I type in a query -- a request containing information about the book such as the title or author, for example -- the computer should display a list of books that match my request. I can select a particular book, and the catalog will tell me where in the library to find it (or whether it is currently checked out and so unavailable).

For example, I might ask for books about Alan Turing. The computer should produce a list of books that satisfy this property. If I decide that I'm particularly interested in The Enigma I can ask the computer for
more information about it. The computer should indicate whether it is available to be checked out and where in the library I might find it.

Using the specific information provided by the catalog, I next need to retrieve the book I want. This part of the interaction involves physical space and objects, so it can't be handled by a typical computer; we'll see below several ways to solve this part of the problem.

Once I have the book in my possession, I need to give the library system my library card and the book. At this point, the library's computer needs to transfer responsibility for the book to my card. This includes updating its own records so that anyone looking for this book can discover that it is currently unavailable.

Other interactions with the library system involve returning a book -- checking it back in -- as well as variants on the above scenarios, such as identifying a book but then discovering that it's not available -- or extensions -- like adding the ability to reserve a book for future checkout. Another set of interactions -- on which we will not focus in this chapter -- involve library-maintenance functions, like determining which patrons have overdue books or adding new books to the system. In a real software engineering project, it is important to understand the scope of the project at the outset.

### 2.2.2 Use Cases

In this chapter, we will focus on the check-out and check-in interactions of a library patron. In order to understand better what they entail, we will flesh these out further as use cases -- particular patterns of interaction between a user and the desired system -- so that we can make them more precise. Each use case begins with an informal description of the interaction, which is used to clarify which interaction it is to both the user and designer. It also specifies the prerequisites of the use case -- what must be true in order for this use case to arise -- and its effects -- what changes occur as a result of the interaction -- as well as the sequence of actions and interactions that make up the activity of the use case. Of course, your use cases should find their way into your engineer's notebook.

1. **Title: Library-Lookup**

   I come to the library to look for book about jabberwocks. (My father told me to beware....)

   **Prerequisites:**
   
   I have (knowledge of) information/keywords that identify the book I'm looking for.

   **Actions/interactions:**
   
   I tell the computer catalog what I'm looking for.
   
   The catalog tells me which book I need.

   **Effects:**
   
   I know something I didn't know before; otherwise, this use case has no effects.
OK, so maybe that wasn't the most exciting use case. Let's do another one, this one including actually leaving with the book:

1. **Title: Library-Lookup-Checkout**
   I come to library to look for book about jabberwocks and check it out.

   **Prerequisites:**
   - I have (knowledge of) information/keywords that identify the book I'm looking for.
   - That book is available in the library.
   - I have a library card.

   **Actions/interactions:**
   - I tell the computer catalog what I'm looking for.
   - The catalog tells me which book I need.
   - I get the book and check it out.

   **Effects:**
   - The book is transferred from library's possession to mine. (Better record this fact somewhere!)

   Note that this use case is really the same as the previous use case with some extra actions/effects added onto the end. In fact, the second half of the lookup-checkout use case could be its own use case, if I came to the library already knowing precisely what book I was looking for.

   @ see exercise # @

   Of course, the **Library-Lookup-Checkout** use case presumes that the book is available. If it is not, a slightly different use case results:

1. **Title: Library-Lookup-Can't-Checkout**
   I come to library to look for book about jabberwocks and check it out.

   **Prerequisites:**
   - I have (knowledge of) information/keywords that identify the book I'm looking for.
   - That book is not available in the library.
   - I have a library card.

   **Actions/interactions:**
   - I tell the computer catalog what I'm looking for.
   - The catalog tells me which book I need.
   - I try and fail to get the book. *(OR the catalog tells me it is not available.)*

   **Effects:**
   - I know something I didn't know before; otherwise, this use case has no effects.
With all of these books being checked out, we should also include a use case to return a book to the library:

1. **Title: Library-Checkin**
   I return a book to the library.

   **Prerequisites:**
   I have a particular book in my possession. (The library has a record that this book is in my possession.)

   **Actions/interactions:**
   I give the book to the library.
   (The library records this fact.)

   **Effects:**
   The book is transferred from my possession to the library's, presumably updating the library's record system.

   There are, of course, other possible use cases for a library, even one as simple as this. By now, though, the basic ideas should be clear. Each use case lists its prerequisite conditions, its actions or interactions, and the effects it has. For each use case, you can design a test of the system you eventually build; that test will verify that the system supports the desired behavior. For example, Library-Lookup-Can't-Checkout could be tested by ensuring that *Alice in Wonderland* is not available and then asking the system to check out all books written by Lewis Carroll to a particular patron.

   These tests form the beginning of a *test suite*, the set of tests that you develop along with your program and that you will use to ensure that your program behaves as it should. As you develop them, these tests should go in your engineer's notebook along with the use cases. For each test, be sure to record its inputs, the timing of those inputs, and the behavior that you expect to see. You can also record any signs that you ought to see -- part way through -- that things are going right as well as signs that indicate a problem. Developing tests along with -- or even before -- your design is an indication that you understand the integrated nature of the software life cycle.

**2.2.3 Assumptions**

Use cases document the way that a system is intended to be used. Behind these use cases are a series of assumptions. It is always important to make these assumptions explicit and to record them. Many decisions that you as a software developer will make are based on your assumptions. It is essential that you understand the assumptions that you are making and that you check to see that these assumptions are really valid. What assumptions do we make about our library?
Some assumptions reflect operating conditions, i.e., when the program can reasonably be expected to behave properly. For example, we are going to make some strong assumptions about correspondence between what is in the on-line world of our program and what is going on in the real world around it. If (the electronic information corresponding to) a book is checked out to (the electronic information corresponding to the library card of) a patron, we assume that the physical book is in the possession of the appropriate patron. When the book is listed as in the library, we assume that it is in fact there and appropriately shelved. All of these assumptions are likely to be violated by a real library -- in which books are sometimes stolen or mis-shelved -- but we assume that violations are addressed outside of the scope of the program we're designing. (We might want to think about how someone could manually override parts of the program to correct these issues should they arise, though.)

In almost every program, there are things that are outside the scope of that program. Being explicit about the operating assumptions of the program helps its ultimate users understand what the program can and can't be counted upon to do. For example, in this case a librarian might periodically take inventory of what is actually on the bookshelves. This kind of human check on information collected by a program is especially helpful to maintain a correspondence between the online and real world. This assumption, together with, e.g., an assumption that cards won't be forged, are really assumptions about the way that this system is embedded in a larger society and about social practice within that society.

To make our implementations easier, we will assume that each book is equipped with a unique bar code and that the checkout and checkin is performed using a bar code reader. A bar code reader is a piece of computer equipment that can read the bar code on an object and produce a number corresponding to that code inside the computer.

We will also assume that every library patron has a unique library card with its own unique bar code, and that library card is a good stand-in for the patron. We will not, in the system that we are designing here, explore how people are given library cards; this is a separate system that we could build. We will assume that we don't have to worry about forged library cards.

This assumption raises questions about where unique identifiers such as the bar code come from. In fact, there is an additional assumption hidden here, that every library card and every book has a unique bar code. This assumption has to be enforced by a human being (or another computer program) whose responsibility it is to distribute bar codes. If this assumption is violated, the program will not be able to tell which book is checked out to whom. Enforcement of unique identifiers isn't too hard if there is only one centralized place where they're given out, but what if each branch of the library is allowed to assign bar codes to library patrons? How do they make sure not to assign the same number in two places? There are some straightforward ways to deal with this -- assign each branch its own initial sequence that's part of every bar code it issues -- but in general the question of assigning unique identifiers is a complicated one in a distributed system.

Other assumptions may help in simplifying program development, but might eventually be relaxed. Initially, at least, we are going to assume that the library has only a single copy of each book. This is a
potentially dangerous assumption, as it lets us treat the identifying information of a book (author, title, etc.) as interchangeable with the book itself. We will give each individual book its own bar code, but we will also assume that there is only one bar code corresponding to a particular title/author combination. That way, we won't have to worry about two books with different bar codes that are otherwise identical. By making ourselves aware of this assumption explicitly, we can plan for a future version of the system in which there might be additional information determining which of several interchangeable copies of the book we have.

If we were to relax this assumption, we would need to add a new component to the system to manage the multiple interchangeable copies of a particular book. If we design the system carefully, we can later plug such a component in without disrupting the whole system. It's important to know where this assumption matters -- in the catalog -- and where it doesn't -- in the checkout system, where the unique bar code of the book is all that matters.

Another simplifying assumption involves the role of time. We are going to assume that actual time doesn't matter in our initial prototype. We might choose to time-stamp check-out transactions, but we won't worry about how much time has passed, whether books are overdue, or other such issues. In other words, we will not worry about the accuracy of any clock in the system. Even if we eventually decide to add information about due dates, fines, etc., we can do so without having to worry about actual times; we'd only need to keep track of dates.

Some systems, like the library system described here, can have very relaxed notions of time. Other systems, such as a robot controller or an automobile's cruise control system or a microwave oven, need to function in real time: time matters to every aspect of their operation, and time inside the system must be locked to time outside the system. Most systems fall somewhere in between, needing to keep track of time to some extent, but not to be exactly in lock-step with the rest of the world. For example, a hospital's pharmaceutical inventory control system needs to keep track of who got what medicine when, but it is probably not important for inventory control to be accurate to within ten minutes, and it certainly doesn't have to be accurate to within seconds or milliseconds.

We will assume that the programs running this system are robust and that the computers on which they run do not crash in the middle of things. This is definitely a bad assumption -- computers do crash, and a program such as this really does need to be secure even if the computer system crashes in the middle of a check-out. A real-world program of this sort would need to contain extra machinery to deal specifically with this problem. Our version here will not address these issues.

Finally, there are assumptions that we do not make. These non-assumptions should be recorded in your engineer's notebook as well. We will not assume that there is a single check-out point. Instead, we assume that two different people could check two books out simultaneously, and that our system has to be able to handle this. As with unique identifier distribution, this imposes additional complexity on our system. It means that whatever structure keeps track of who has which book will need to be careful not to let the same book go to more than one person, or be simultaneously checked out to a patron and in the library. The issues raised by this assumption are addressed in some depth in chapter [Concurrency].
2.2.4 Promises/Guarantees

In addition to understanding the assumptions our programs make, we need to spell out the promises or guarantees that are a part of the behavior they will provide. These form a basis for the contract our system will make: what you need to know to interact with the system. We will see later that individual component elements of the system may also have their own promises and guarantees -- their own contracts -- that they make to one another. In all cases, promises and guarantees are important to record in your engineer's notebook.

When a user asks the catalog about a title, author, or other characterizing information, the catalog promises to include all matches in the information it supplies the user. Said another way, if a book matches a user's request, that book will be included in the answer provided. We should also include the opposite promise: Only those books that match will be included in the answer. Otherwise, the catalog could simply list all of the books in answer to every question; this would meet the first of these promises (if a book matches, it will be listed), but wouldn't be particularly useful.

The first of these promises -- if a book matches, it will be listed -- is called **completeness**. It means that the system contains (or supplies) all of the (true) information. The easiest way to guarantee completeness is to have the system supply all information, true and untrue, relevant and irrelevant. The second promise -- only matching books will be listed -- is called **soundness**. It means that the information in the system is correct. The easiest way to guarantee soundness is to have no information in your system. An ideal query system -- one that lists exactly those books that match -- is both sound and complete. [[Footnote: Technically, a system is either sound or unsound, either complete or incomplete. It is often useful to talk about how well a system matches these criteria, though, and there are different terms used to describe these properties. **Recall** is the term for how close a system comes to being complete. The hypothetical "return all books" version has perfect recall. The term for how accurately the returned suggestions match -- how closely the system approximates soundness -- is **precision**. The return-everything version has very poor precision. A return-nothing version has perfect precision -- all of its nonexistent suggestions are matches -- but lousy recall. The problem of optimizing precision and recall simultaneously is the subject of the field of **information retrieval**.]]

Why would we want to relax either of these promises? Perhaps the user will issue a query that matches 100 books, or 1000. Do we want to display all of these? Is it OK to display only a set, or to tell the user that the query has too many matches? Perhaps. We won't implement these features in this chapter, but they are extensions you could imagine adding to our system.

Alternately, maybe we want to include some "near misses", books that we think the user might have been asking about even though they don't strictly match. For example, if the user asks for books about "Harry Porter", we might want to suggest J.K.Rowling's Harry Potter series or even the business wisdom of Michael J. Porter.

Again, we will not explore these extensions in this chapter, but they are certainly reasonable add-ons one might pursue. Both involve additional sensitivity to the needs of the human user. We will begin to touch on the issues of user interface design in this book, but the field is one that you will want to learn more about as you develop your software engineering skills.
Another set of guarantees involves the checkout system. We need to ensure that every book is in the possession of exactly one patron, or of the library, at any given time. A book cannot simultaneously be in the possession of more than one of these parties. It wouldn't do to have the system check out a single book to two patrons, or simultaneously list the book as checked out and on the shelf!

This means that when a user checks out a book, that book will be associated with that user's library card and NOT with the library itself. When the book is checked back in, it is cleared from the user's library card and associated with the library again. Further, a book must be in the possession of the library before it can be checked out. So if a book is checked out to a patron, it cannot be checked out to another patron until it has first been returned to the library. Guarantees such as these suggest additional tests that you will want to add to the test suite emerging in your engineer's notebook.

Remember, this is a guarantee about the record-keeping inside the library system, not about the physical book. We assumed above that the physical book would be in the right place, but the computer can't guarantee that itself.

Finally, we want a guarantee that whenever the book is in possession of the library (according to the checkout/checkin system), the catalog lists it as available. Again, this is a two-way condition: the book should be listed as available whenever it is in the library's possession and only when it is in the library's possession. Otherwise, we could always list it as available to satisfy the first promise. We will actually be willing to allow a gap of as much as a few seconds between when the book is checked in and when the catalog lists it as available, but it would not be OK if this gap became hours, or days. This is a constraint on how quickly information has to get from one part of our system to others.

2.2.5 The Community Around You

No implementation happens in a vacuum. The assumptions and guarantees of the previous section are to be met by a system that you will build out of existing parts and to interact with an existing environment. Assumptions document constraints that your implementation will impose on the environment. Guarantees are constraints that you've agreed to let the environment make on you. These two sets of requirements form a part of the specification of the interface between your system and its environment. If written well, they may be all you need to know about the world in which your system will be embedded.

But there is a second way in which your system is affected by things around it. Invariably, you will use tools that you didn't build to construct your system. Some of these -- the Java language, for example -- are very general purpose and you will use these over and over. Others are more specialized and you will only use them on a particular occasion. For example, if your job is to upgrade an existing system to work in a new context, that old system may form a piece of your community. You will also likely find that many specific problems that you encounter have ready-made solutions that someone else has built. Being able to find, understand, and incorporate other people's tools into your systems is an extremely important skill.
In this section, we review some of the major elements of the communities in which your system is embedded.

2.2.5.1 Physical Environment

We have already seen how the assumptions and guarantees of the library system constrain the kinds of environments in which the system would work. Some of these constraints may have come from our desire to simplify the implementation: We are not (yet) building a system that provides internet access to the library, for example. Other constraints may come from the real world requirements of the customer for whom we are building the system: Multiple librarians need to be able to use the system simultaneously.

When you are building a piece of software, you need to understand the requirements of the customer (or the problem definer) before you begin. This can take some back-and-forth as you propose solutions, the customer decides that you've misunderstood or realizes that s/he has an additional need your system won't meet, and you revise your proposals in response. There are also real constraints--the types of computers available, the specifications of components you need to integrate--that are non-negotiable. For example, the library may already have purchased bar code readers; you will have to work with their actual interface.

2.2.5.2 Program Libraries

It is extremely rare to build an entire system from scratch. [[ Footnote: In fact, even "from scratch" usually relies on program-building tools that already exist, but in this case we're talking about using pieces of programs that others (or you, previously) have built. ]] Usually, your problem decomposition will eventually turn up the need for some components that already exist.

For example, our library checkout system will make use of some pre-existing components. Of course, we'll assume that the bar code scanner reads a bar code from a book or from a library card and produces a number. We'll want to be able to use that number to identify the computer's record for a particular card or book. [A record is just the computer's representation of information about that real-world object.]

To accomplish this, we will assume that we have a pre-existing piece of software that can associate a key--like the bar code number--with a value--like the computer's record of a particular book. In fact, this software component should be able to store a large number of keys and their associated values and respond to any key by supplying the associated value. This particular kind of structure is called a lookup table. It is a very common kind of software component, and Java provides several different kinds of lookup tables, as we shall see in later chapters.

Pre-existing software, such as the lookup tables provided by Java, is often collected into groups of inter-related components. These components aren't complete programs by themselves, but they are frequently useful in building other systems. (A lookup
table is a good example; it's not usually of much use until it's incorporated into a larger system.) Such a set of program components that is sitting around, already written and waiting to be used, is called a program library (or, when we're not also talking about the book kind, sometimes just a library). Java has a number of very useful libraries that are part of its standard distribution, but programmers all around the world produce a much wider variety of libraries. Many of these are available on the web. Many are freely available; others require the purchase of a license. In this book, we make use of a set of libraries (the "cs101" libraries) that are freely available and were designed specifically for this curriculum.

In building our checkout system, we are going to assume that we have several already-filled-in lookup tables. For example, we'll assume that we have a lookup table that associates individual book bar codes with generic book descriptions, such as authors or titles. If we were really building this system, we'd need to supply a piece of software that allows a person to enter information about new books. This way, the lookup table can be created or extended as the library grows. A similar lookup table relates library card bar codes with information about library patrons (like the address to send their overdue notices!); a complete system would also allow a way to add a new library card and the patron's information.

We will also need a piece of software that can identify any records that share a particular field. This will be used to identify, for example, all book records that share a particular author. This can be accomplished with a set of key-value lookup tables, but there are also other ways to build such a system. We won't worry about how that component works; we'll just assume that we have one. This lets us get from a query about books written by Lewis Carroll to *Alice in Wonderland* and *Through the Looking Glass*.

Finally, we will assume that we have a number of components that present information to the user nicely and elegantly. These components may use windows, icons, menus, etc., to facilitate the user's interaction with various computer screens. We will begin to explore how such things might be implemented in part 4 of this book, but for now we will simply describe what information needs to be presented to the user or obtained from her, without specifying exactly how that information should appear on the screen.

### 2.2.5.3 Users

In building our system, it is important to remember that not all of the members of the community are program libraries and physical devices (such as bar code scanners) and other pieces of software or hardware. Our systems frequently involve interaction with human beings. People have a set of requirements that differ from hardware and software. People are much more adaptable to your system; their requirements are often more flexible. But good computer systems also make things easier for human users. If a person sits down to use your system, s/he should not have to read a thick manual before getting started. A good system design will incorporate the natural abilities of a human user so that the system is intuitive to use.
The part of your system that interacts with humans is called the **user interface**. Human community members, like hardware or software community members, come with their own sets of assumptions and guarantees and you will need to design an interface that works for human community members just as you would for other members of the community or environment around your system. Human beings typically appreciate visually presented information (though sometimes it's important to use other modalities, such as sound). Humans benefit from clear labeling and instructions that would be superfluous for a machine. Human time scales are typically slower than machines -- responses are measured in **hundreds** of milliseconds -- but people are much less patient than machines when delays become long.

A good understanding of how people work is the goal of the field of **human factors** analysis. Because interaction with human beings is an important part of many computer programs, every computer programmer should learn how to design a good user interface. Many of the properties of good interface design are obvious. A user interface should be simple, clear, intuitive. It should make it easy to do what you want to do and harder to make mistakes. For example, when you insert a new software CD in your computer, you may have to spend five minutes locating the install file. Alternately, inserting the CD might immediately open a window that says, "Do you want to install this software now? If so, click here to begin; if not, click here to close this window." The second is a much more intuitive interface for such a CD. But many aspects of user interface design are more subtle or not given sufficient attention by system designers. Otherwise, how can you explain the difficulties that so many people have in programming a VCR, using a new computer program, setting up a printer. Too many programmers have focused on designing for the physical, hardware, and software components of their environment and paid little attention to the human beings who are their system's most important community members.

Sometimes, a system will have more than one user interface depending on who might be using it. A simple version of this is a web site that offers a frames-and-images version or a very simple text only version of the same information. In a system like our library checkout system, we might provide one clean, simple, intuitive interface for the librarian and/or library patron and another -- more obscure, more complicated, less intuitive -- interface for the system programmer who is responsible for maintaining the checkout system.

**2.2.5.4 Understand their interfaces (and assumptions)**

When you commit to using or interacting with something from the community around your program, you need to understand its behavior, assumptions, and guarantees. These amount to its contract -- the behavior it promises to you and the circumstances under which it makes these promises -- as well as its peculiar properties.

For example, people have very flexible behavior and somewhat negotiable interfaces. You can train a human user to interact with your system in a particular way. (Just think of all of the crazy things people do to get their computers -- or other machines -- to cooperate. [[Footnote: Donald Norman has written an excellent book on this subject, called *The Psychology of Everyday Objects*.]] In fact, it is generally much easier to change the behavior of a human
being than that of a computer program. However, people have some particular expectations that are not really
negotiable. A computer may be willing to wait minutes at a time for an answer; in many cultures, a person is
rarely willing to endure delays that are measured in seconds.

Physical environments tend to be much more rigid than people are. But a program can be artificially
constrained to work only in particular physical environments: most robots only work indoors, not outdoors,
and most wheelchairs cannot go up or down steps. A physical environment can also sometimes be modified,
tailoring it to your program: curbs can be cut to make ramps, or books can be outfitted with bar codes that
uniquely identify each one. Sometimes, you can even find regularities in a physical environment that you can
exploit to make your program work better, like the fact that the title of a book is usually the set of words
printed in the largest type on its title page.

Code libraries, when you use them, should generally have good documentation that explains what that
set of code does and under what circumstances. In this book, we will explore some existing code libraries in
Java and you will learn how to understand what they may be able to do for you. New tools are constantly being
created, though, and you will need to build the skill of understanding a new library. Software engineers rarely
build from scratch. You should always be on the lookout for good tools that can help make your job easier. As
you encounter them, record them in your notebook along with the problems that you think they might
someday help you solve.

2.2.6 Requirements are a moving target

There are many additional features that one can imagine adding to the system as described here. For
example, it would be nice to have another part of the program that could look over all of the books checked
out of the library and determine which of these were overdue; overdue notices to the corresponding patrons
could then be generated. We are not going to design this feature in now, but thinking about it reminds us that
the check-out transaction will need to be date-stamped; that is, we'll need to know when it happened (or at
least when the book is due to be returned). This kind of anticipating possible future augmentation often turns
up modifications to the basic system. Not all of these should necessarily be accommodated -- simplicity is an
important principle -- but thinking about them can often help you create a more robust base system.

In this book we will rarely talk about the solution to a problem, as though there were only one. Instead,
we will explore many ways that particular problems can be solved, and we will compare and contrast these
different approaches. Although there is rarely only one right way to solve a problem, there are invariably
wrong ways, as well as less desirable ways, to do it. In this book, you will not only learn about useful
techniques. You will also develop some of the judgement that a skilled programmer needs about which
approach to use when. Of course, this judgement is something that you will continue to develop through your
experience writing, understanding, testing, and modifying programs.

The moral of this story is this: A good specification makes program development easier. However, it
is a rare specification that is definitive. Instead, most program specifications are representations of the
designer's understanding of the problem to be solved at a particular point in time; every program should be built with the understanding that it is likely to grow and change in often-unanticipated ways. Part of good program design is building something that works and meets the specifications set out for you. Part of good program design is understanding and developing those specifications, including directions in which they might actually change in the future. And part of good program design is developing programs that are easy to understand and modify, documenting not just what your program does but also why and how you made the design decisions that you did, so that it will be easier to modify your program in the future.

Above all, the overriding principle of design is not to unnecessarily complicate systems, especially in the early stages of design. The more streamlined and simple the core of your system, the more likely it is to be able to accommodate unanticipated changes because it will be easier to understand and work with.

2.3 Designing the System

In the previous section, we asked what behavior our program should have. In this section, we will begin to decompose that behavior into the pieces -- the community members -- whose combined efforts will create that behavior. The questions that we will look at in this section are:

- Who are these members of the community? What entities combine to create that desired behavior? This includes an understanding of the desired behavior of each of these entities in turn.
- How do these community members interact? What contracts -- what behavior, assumptions, and guarantees -- do they make with one another?
- What goes inside each one? That is, what is each of these entities made of? Is it, itself, a community? If so, we will need to ask the questions of this section about it, as well. Or is it a simple instruction follower? In this case, we will need to write its recipe. Or is it something outside the system that we are constructing -- a program library, a human being, a physical system like the bar code scanner -- whose properties we need to understand but whose implementation we can simply adopt?

In asking and answering these questions, we will develop the implementation -- the solution -- to the problem requirements we described above. Like the requirements exploration, the solution design is an important piece to record in your engineering notebook. Begin with questions and proceed, step by step, to answer them.

2.3.1 Who are the members of the community?

One way to figure out what things your program needs is to look for nouns and verbs. That is, in your description of the system, you will talk about the things that are a part of your program and the actions that they perform or are performed on them. The things -- the nouns -- are objects or entities that you will likely
need to create. The actions are recipes that these things will follow. In a library, typical nouns include book, library card, and catalog; typical verbs include check in, look up, etc. We'll begin with the nouns.

Consider, for example, book. There is a physical thing -- a book -- in the world, but inside the computer we're going to need some other thing to represent the book. Java, the language we'll ultimately be using to build our programs, is a kind of language called an object oriented language. This means that most of the things in a Java program -- the nouns, the stuff, the bits and pieces that are manipulated -- are objects. (An object is a particular kind of computer structure, about which you will learn more in part 2 of this book. For now, object is how you say thing in Java.) So, in Java, we will create a particular kind of object to contain all of the information about a physical book that we want to represent in the computer. We'll call that object a BookID. (There's nothing magic about this name. You could call it a Fred or a Football, but your program might be harder to understand in that case.) A BookID might contain the title and author of the book; it will certainly need to include its bar code.

Another noun for which we'll need a kind of object is the library card. We'll call the electronic object that contains all of the information about the patron whose card it is a PatronID. When a book is checked out to a patron, we will record this connection between the book's BookID and the patron's PatronID. We will even create one special PatronID for the library itself, because it will be convenient to be able to treat books that are not checked out as being associated with the library's PatronID. We will give the library's PatronID a special name: LIBRARY_ID. [[Footnote: Again, there's nothing particular about this choice of names except that we think it will help us remember. If you'd rather, you can call the library's PatronID Rumpelstiltskin.]]

The card catalog is the old-fashioned place that you would go to look up a book. Now, it's more common to use a computerized version that operates more like a web search engine. For this quaint historical reason, we'll use the name CardCatalog for the kind of object that keeps track of what books which author wrote. Of course, our CardCatalog isn't likely to contain any of those lovely old cards, but it will hopefully be faster to search.

A CardCatalog is the kind of object that you'd like to be able to ask to do something for you. In particular, you'd probably like to be able to make requests like "look up Shel Silverstein" of your CardCatalog. In an ideal world, you'd ask the CardCatalog to look up something, and it would hand you back a piece of information uniquely identifying a particular book -- a BookID. So we'll assume that the CardCatalog has a thing that it knows how to do, called lookup. Note that this is a funny kind of assumption, because we will have to create the recipe for doing lookup later. For now, though, we're saying that CardCatalog will have such a recipe, and not yet worrying about how to implement it.

This is a completely typical way to do design. We are asking, "who are the members of the community?" We are presuming a (partial) answer to the question, "how do they interact?" -- CardCatalog will provide a lookup service -- but not yet worrying about "what goes inside?" We are not done until all questions are answered, but we don't have to answer all questions at the same time.
What other nouns does our system need? Well, somewhere there is a record of who has checked which books out. For similarly quaint historical reasons, let's call the computer representation of this record a CirculationDesk. There are three verbs associated with the CirculationDesk: check out, check in, and verify availability.

In a real library, there is another thing that comes between the card catalog -- the place where you look up the particular book you want -- and the circulation desk -- where you transfer responsibility for the book from the library to yourself. This is the bookshelf, where you go from the description of information that the card catalog gives you to the actual, physical book. Because the actual physical book isn't inside the computer, our system may not need to include a component to deal with it. But if there were something that the computer needed to do with the physical book, Bookshelf might be a kind of object in our system and its associated verb would be fetch the book.

Note that BookID and PatronID don't have associated verbs in the above description. So far, we have no need for either of these kinds of objects to do anything in particular. But if we were going to send overdue notices out, we might, for example, want PatronID to have an associated print mailing address verb. These kinds of things can be added to the system now -- at design time -- or later, as the system continues to grow and improve.

### 2.3.2 How Do They Interact?

Once we know what kinds of things exist in our system, we can write down how those things work together to create the overall behavior of that system. If the "Who are the members?" question is really about nouns, "How do they interact?" involves looking closely at verbs. For each, we specify what inputs it needs, what outputs it provides, and under what assumptions it operates; in short, what contracts it makes. Putting these pieces together should yield your use cases. As always, your notes from this step are fodder for your engineer's notebook.

When I ask the CirculationDesk to check out a book, I need to supply it with two items: the BookID of the book I want checked out and my PatronID. So a CirculationDesk's checkOut recipe needs to be supplied with a BookID and a PatronID. I might, for example, say:

```
circulationDesk, please checkOut this bookID to patronID
```

where I'd need to specify a particular bookID corresponding to the book I wanted to check out and the patronID of the person to whom the book should be checked out. (I could, for example, get those two pieces of information from the bar code reader scanning the book and the library card in question.) It might also be a good idea for the CirculationDesk to let me know whether this checkOut succeeded. So the contract for a CirculationDesk's checkOut action is: needs a BookID and a PatronID, provides a signal of success or failure.

The formal part of a contract says who is offering this behavior (or at least what types of "who"s), what that entity needs to be given, and what it provides in return. The informal part of a contract -- often included
in accompanying documentation -- specifies the relationship between what is given to this entity and what it
returns, what else changes while the contractual behavior is happening, and when the contract can or should
be used. Although a legal contract is generally made between two parties, a software contract is really offered
by one (kind of) entity and can be used by anyone willing to agree to its terms.

The checkIn contract is not quite the same as checkOut. After all, when I'm returning a book, I don't
need to specify a PatronID. So checkIn requires a BookID and provides a signal of success or failure:

circulationDesk, please checkIn this bookID

The CirculationDesk's verifyAvailability action needs a BookID and provides a yes/no answer.

The CardCatalog's lookup action is more complicated. I would like to be able to give it an author or a
title or a keyword or several of these things at once. I might want to distinguish these as CardCatalog
lookupAuthor or lookupTitle actions, or I might want to hide all of that machinery inside the CardCatalog
object and just have one lookup action. There is not a right answer to this question; there are advantages and
disadvantages to designing this object in either way. One important point, though, is that a user who is
expecting the CardCatalog to provide a lookupAuthor action is going to be very surprised if the CardCatalog
only has a lookup action (or vice versa). So even when the decision may seem arbitrary, it is important to make
the decision and to document the decision so that all of the pieces of your program can work together. For the
purposes of this chapter, we will imagine that CardCatalog simply has one action, lookup, and any magic about
titles or authors is handled by the CardCatalog out of our sight. (Of course, this makes the job of the designer
of the CardCatalog harder.)

At this point, we can begin to piece together scripts corresponding to each of our use cases. For
example, a script for the Library-Lookup-Checkout use case might look something like this:

1. Patron enters library, approaches CardCatalog.
2. Patron asks CardCatalog, "please lookup books about jabberwocks"
3. (CardCatalog does its thing. Fleshing out this step requires looking inside the CardCatalog, which we'll
do below.)
4. CardCatalog tells Patron the BookID associated with Alice in Wonderland
5. Patron [[Footnote: or, in another possible implementation, CardCatalog!]] provides this bookID to CirculationDesk
   along with PatronID, asking, "please checkOut this bookID to this patronID".
6. CirculationDesk carries out its action (again, using a script to be written below) and reports success.
7. Patron departs, happy.

In this case, the script is just an embellished list of the actions involved in the use case along with a
detailing of the information provided by each community member to another, i.e., how one entity uses the
contract of another. If you were to act this out with multiple people, this script would be sufficient to describe
all of the actions of the library patron. Eventually, both CardCatalog and CirculationDesk will need to have
scripts -- recipes or playscripts -- of their own so that they can carry out their parts of this drama. We will turn
to these questions next.

2.3.3 What is each one made of?

So far, we have decomposed the library into interacting community members like CirculationDesk and
CardCatalog. Now, we turn to each element and examine what goes inside: what is it made of? When we
described each community member, we asked what its behavior was and what contracts it made with other
community members. Now, we ask how we can accomplish this. Is the community member itself a
community, or is it a simple instruction follower? If it is a community member, we must ask the community
questions -- who are its members, how do they interact, and what is each one made of -- all over again. If the
community member is simply an instruction follower, then we must write its recipe.

Let's now look at some of the individual components of our library. We'll start with the
CirculationDesk.

2.3.3.1 Some Decompositions are Communities

A CirculationDesk has to provide three actions:

- checkOut (a BookID to a PatronID)
- checkIn (a BookID), and
- verifyAvailability (of a BookID).

If these actions are to be requested by other pieces of the system, they will need to be provided as
pieces of program code that can be run, or called, by other pieces of code. (This is like when one recipe refers
to another recipe in the same book.) This collection of callable services is called an interface, and you will
learn much more about interfaces in chapter 4.

The CirculationDesk actions might also (or instead) be requested directly by a human being. In this
case, there ought to be some machinery that makes it easy for the human being to make this request: a user
interface. For example, part of the CirculationDesk might include a bar code scanner with a "check out" button
attached; pressing "check out", then scanning a library card followed by one or more books would be regarded
as a request to check out those books to that library card. The whole CirculationDesk would consist of two
sub-entities: the software that operates the bar code scanner and makes requests, and the piece of software that
provides the checkOut/checkIn/verifyAvailability interface described above.

Or the CirculationDesk might put up its own web page, allowing a person to type in the bar code
number of the book s/he wanted to check out as well as a valid library card bar code number. The web-page-
providing piece of the CirculationDesk would be a different kind of user interface. The web page controller
would know how to make things show up nicely in a variety of different browsers. In addition, like the bar code scanner-and-checkout system, the web page controller would know how to make requests of the same old piece of the CirculationDesk that provides the checkOut/checkIn/verifyAvailability interface.

Once again, there is not a right way to make these design choices. However, since all three designs involve the same checkOut/checkIn/verifyAvailability interface, it makes sense to focus early efforts on that piece of the CirculationDesk and only later to add one or both of the other functions. (See Keep It Simple and Keep It Working, below.) In this case, the promise to the other parts of the system you are building is the same, so the availability of the different user interfaces should not affect the working of the software other than the CirculationDesk.

2.3.3.2 Other Decompositions are Recipes

And what goes inside the checkOut/checkIn/verifyAvailability part of the CirculationDesk? We have said above that we are assuming the presence of a lookup table, a piece of pre-existing software that will associate a key -- like a BookID -- with a value -- like a PatronID. Java provides several possible structures suitable for this purpose, as do most modern programming languages. Let's call the particular lookup table inside our CirculationDesk the masterList. The masterList, like any lookupTable, has two main actions:

- the masterList can put a bookID, patronID pair into its records
- the masterList can get a bookID's record, which provides the associated PatronID

Once we have the masterList to work with, it's not that hard to figure out how to write the recipes for the CirculationDesk functions. For example, CirculationDesk checkOut could be implemented as:

to checkOut a bookID to a patronID:
1. masterList, please put the pair bookID, patronID into the record
2. report success

The first step of this recipe is simply an invocation of masterList's own put recipe; the second step completes the checkOut contract.

Similarly, we can build recipes for CirculationDesk's other actions in terms of masterList's actions:

to checkIn a bookID:
1. masterList, please put the pair bookID, LIBRARY_ID into the record
2. report success

recalling that LIBRARY_ID is the special PatronID assigned to the library.

to verifyAvailability of a bookID:

1. masterList, please get bookID's record; call the associated PatronID whoHasIt
2. report availability exactly if whoHasIt is LIBRARY_ID

In other words, look to see who is recorded as having the book, and say it's available if this answer is "the library."

The decomposition of the CardCatalog is very similar in principle to CirculationDesk, relying on a structure that keeps track of keyword associations to BookIDs. However, since the lookup needs to be able to happen in several different ways -- by author, by title, by keyword, etc., and by various combinations of these things -- the underlying record-keeping is likely to use a more complicated structure than a simple lookup table like masterList and the associated recipes are likely to be somewhat more involved. Additionally, the CardCatalog may have a very sophisticated user interface, allowing easy presentation of large amounts of information, or provide support for incremental refinement of search criteria as the user tries to narrow down what s/he's looking for.

2.3.3.3 Community Members Come in All Shapes and Sizes

You may recall that there was a third member of the library community, invisible in our implementation but potentially important in other versions. This is the Bookshelf, which would fetch the book.

In our implementation, we presume that the human being in the library is responsible for bringing the BookID supplied by the CardCatalog's lookup to the CirculationDesk for checkOut, presumably stopping by a physical shelf to pick up a physical book along the way. So Bookshelf's fetchTheBook action is actually performed by a human being -- the library patron -- in this case.

The informal system just described actually corresponds to a formal system in place in certain closed stack libraries, such as the United States Library of Congress or the New York Public Library's main branch. In a closed stack library, individual patrons are not allowed to wander into the stacks of bookshelves and select books for themselves. Instead, a request for a precise book -- essentially, a BookID -- is given to a staff member of the library, whose job it is to go and fetchTheBook from the closed shelves and provide it to the
patron. Our library system would work just as well in this kind of a setup, provided that the BookID produced by the CardCatalog's lookup action was given to the library staff. Again, though, the work of this recipe is being done by a human being; in this case, a library staff member.

A mail order library or circulating collection, such as Books On Tape (TM), works similarly. You request a particular item from the library, and fetchTheBook is implemented by the employees of the mail order company, often in conjunction with the post office or parcel service. In this case, the behavior of this system component may be provided by a literal community -- the various order-takers and shelf-pullers of the Books On Tape corporation plus a fleet of trucks or planes and associated personnel at the delivery service -- in order to fulfill the same behavioral interface.

A fourth alternative implementation involves an eBook, in which there may be no physical object to be transmitted. In this case, the Bookshelf's fetchTheBook action might involve a lookup table like the one used by the CirculationDesk. This lookup table would map the BookID key to an electronic version of the text of the associated book. The recipe for the Bookshelf's fetchTheBook action might involve looking up the BookID, getting the associated e-text, and then providing it by email or some other transfer protocol to the patron. All of this could happen inside the computer, without involving any human beings at all.

[Footnote? Or teacher's guide? Or exercise: It is worth noting that the contract of the mail order/Books on Tape and eBook versions of the fetchTheBook recipe are intertwined with the CirculationDesk's checkOut action in ways that the library patron's or library staff's implementations are not. This in fact represents a change of contract between the two implementations. As exercise: Have students act out the two versions, allowing the human user to be nasty and try to steal the book. Presume that the library has a gate that beeps if an un-checked-out book leaves the building; note that the book is already in the patron's hands and outside any beeping machine once fetchTheBook completes in both the mail order and eBook scenarios.]

### 2.3.4 Testing your Design

At this point, we have a rough design for the library checkout system. There are three major components: the CardCatalog, the CirculationDesk, and the Bookshelf. Each of these components has some behaviors (also called actions or services) as summarized in the table below. In addition, we have many BookIDs and PatronIDs, though -- for the present purposes -- these don't have any active behaviors.

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Behaviors/Actions/Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>CardCatalog</td>
<td>• lookup <em>(a keyword)</em></td>
</tr>
<tr>
<td></td>
<td>--&gt; provides a bookID</td>
</tr>
<tr>
<td></td>
<td>• checkOut <em>(a bookID to a patronID)</em></td>
</tr>
<tr>
<td></td>
<td>--&gt; reports success or failure</td>
</tr>
<tr>
<td></td>
<td>• checkIn <em>(a bookID)</em></td>
</tr>
<tr>
<td></td>
<td>--&gt; reports success or failure</td>
</tr>
</tbody>
</table>
Using these components, we can build a system that handles the use cases of section @?@, above. For example, **Library-Lookup-Checkout** is a request to CardCatalog.lookup (yielding a `bookID`) followed by a CirculationDesk.checkout (of that `bookID` to the user's `patronID`). At this point, you should be able to flesh out a full playscript -- like the one at the end of "How do they interact?", but with all of the roles filled in -- for each use case. (If not, you have some more designing to do.)

These scripts -- together with the formal descriptions of each component and the recipes for each component's behaviors -- are the computer program that you are creating. Of course, there is the small matter of writing them in a language that the computer understands. But before you get to that step, it is a good idea to build your system out of human beings first. This gives you an opportunity to see how your system might work. It gives your customer an opportunity to decide that the specification needs to be changed. And it gives you an opportunity to design some tests that you'll want to use when you have actually built the computer program. Each of these is a valuable and important piece of software engineering.

This activity -- acting out the behavior of your system, testing your design -- is something that you can do by yourself, with a pencil and paper. But just as it is difficult to edit your own text, it is very hard to find the bugs in your own program when you are acting it out. It is much easier to see where things go wrong if you have a group of people each following your recipes very literally and precisely. So grab some friends and assign each one to a component. Give each component-actor a set of instructions -- a recipe -- for exactly how to perform each of his or her actions. Another actor plays the user and gets the use case scripts. No one is allowed to do anything unless the script specifies it; each actor has to follow the steps of the script literally. When information is transmitted from one component to another, write it down on an index card and let one actor hand it to the other. Before you begin, provide pieces of paper and pencils for any component actors who will need to remember things. Some of these may start with information already on their pages. For example, the card catalog actor will need to start with a lookup table listing the keywords and books in the library. The circulation desk actor will need another lookup table describing which patron currently has each book. (What should that list say when you begin?)

As you walk through this process, you will likely run into unexpected situations. For example, what happens if two check out requests come, one after the other, for the same book? Try this using the recipe for checkOut above. Do you see a problem? Can you fix the recipe? There is a problem; it as well as some possible solutions are described in the section on debugging, below. See if you can figure it out before you get there.

Your dramatization should make use of the test suite that you have been developing (and that you have continued to record in your engineer's notebook). You should make a point of running through each of the tests that you describe. It may be that several of your tests can be collapsed to make a simpler, more
streamlined test suite. Or you may realize that you have neglected to test crucial features. By running through tests with human beings before you build your computer program, you can simultaneously identify problems in your design and build a more robust set of test cases for the computer program you will eventually write.

A dramatization also allows you an opportunity to show your design to prospective users. Ideally, your use cases capture everything that the user might want to do. Often, though, an opportunity to actually use the system -- or a mock-up of it -- helps the user to realize that additional features are necessary. Even if the specification doesn't change, it is still useful to hear your users' concerns at this early point in the software engineering process.

2.4 Building The Program

You have designed your program. You have identified use cases, decomposed the problem into its nouns and verbs, created descriptions of community members and recipes for their interactions, acted the whole thing out, and received approbation from your intended audience. You have, along the way, identified issues and revised your design to accommodate some of them; in other cases, you have listed areas for future revision and expansion of the system.

You are ready to begin developing your code.

2.4.1 Developing Code

It would be easy to imagine that designing a good program and trouble-shooting your design are sufficient preparation for constructing flawless software. However, just as a design needs trouble-shooting, an implementation will need debugging. You can make your debugging easier by implementing your system in pieces and stages, and by testing each one thoroughly as you go along. Designing your development process -- figuring out how to incrementally build your system so as to minimize the complexity of what's being tested at each stage -- is an important part of the development cycle. But even with the best design and development process, remember: debugging is normal.

Before you write any code, you should come up with a development plan. This plan is a sequence of steps that you will take to develop your program. At each step, you will write some code, test it, debug and revise it as necessary. Once the code for one step is working to your satisfaction, you should checkpoint that version of the code before moving on to the next step. Each step should include only a small amount of additional functionality. Your development plan should describe each of these steps, including what code you will write in each step as well as what new behavior you expect to see and what tests you will run to ensure that the new behavior -- as well as the old behavior -- works as expected. Notes that you have made up to this point about a test suite will prove particularly useful at this stage. Your development plan should be written in your engineer's notebook. As you proceed through it, you should verify your expectations, add comments whenever your code surprises you, and modify the plan to ensure that each step is small enough to be readily testable.
There are several things that you can do as you develop your code that will make your job easier. First, start simple. What is the most basic version of your program that you can imagine testing on its own? Start with the simplest, most stripped down functionality that you can think of. Or pick a component and simulate its interactions with the rest of the system rather than building the whole system at once. Add function incrementally. Always keep a working version (checkpoint!). Make minimal modifications, then test again.

In the library system, you might build a very simple masterList with just one bookID. Then, write enough of the CirculationDesk to be able to verifyAvailability of that bookID. (If the book is available, this action should always say so; if you manually set it to be unavailable, checkAvailability should reliably report that.) Or start with no books at all. (Now if the system tells you a book is available, you'll know you have a problem!) Once that is working, add a few more books and test their availability as before. Next, add one patronID -- make sure that that hasn't changed anything -- and then implement checkOut. Now you can check out a book, then verify its availability. What happens if you try to check it out twice in a row?

@@ see exercise # @@

Along with your development stages, think about how you will test each one. Set targets for what your code should be able to do at each point. What functionality can you demonstrate after you've built the most stripped-down version? How would it respond to inappropriate input? Can you break it? When you add the next feature, how will you test it? Don't forget to test basic functionality after you add features; sometimes seemingly unrelated changes cause previously working aspects of your program to fail, especially if there are interactions in your design that you didn't yet discover.

Finally, don't forget to document your code as you write and test it. Explain what it does, how it works, and why you made these choices. Remember, the next person who reads your code may not know how it's supposed to work. Often, after a break, even you will have trouble remembering why you did what you did.

You should always implement your code in simple, testable stages, building on each stage only after it works robustly. Don't be afraid to move slowly and carefully through the software development process. Basic but elegant, well-tested, well-documented, and well-understood code will serve you better in the long run than featureful but poorly written/documented/tested code.

2.4.2 Compiling Code

In the previous chapter, we described the process by which a computation actually takes place. First, the instructions for that computation must be available. This is like having a script for the play. Second, the computer must execute those instructions. This is like having actors actually perform the play.

When you write a program, you go through a similar process. First you create the program, writing the script(s) that the computer will follow. Typically, you do this using a program called an editor. Later, you ask the computer to perform using those scripts. This is called running your program.

When you build programs using the programming language called Java (as we do in this book), there is an intermediate step that you must take. This is because Java -- as you write it -- is not directly executable by
the computer. It is a bit like having written a play in English and then asking that it be performed in French. (Java is like English in this analogy; what your computer executes would be French.)

After you have written your program (and saved the file), but before you can run it, you must compile it. **Compiling** the program translates it from the version of Java that you write (and that is made to be read by people) into a different notation (called Java byte codes) that is directly executable by an appropriately equipped computer.

This point is important, if subtle. The Java program that you write is not directly executable by your computer. Instead, you must compile it, creating an executable set of instructions. Once it has been compiled, you can run this program as many times as you like. Compilation is a translation step that turns the Java you write into directly executable Java byte code. (Because the compiler starts with your Java program, that program is sometimes referred to as **source code**: source for the compiler.) Compiling the program is not writing it -- you must write the program before it can be compiled -- and it is not running it -- the result of compiling the program is a computer-readable version of the script that can be run. You must write (edit, save) your program, compile it, and run it in order to see what happens.

Depending on the actual system that you are using to write your program, you may be more or less aware of when you switch from writing to compiling to running. Many programmers today use a special piece of software called an **integrated development environment**, or **IDE**. Typically, an IDE includes an editor, a compiler, a run-time environment (i.e., the ability to run your programs), and a debugger (on which more below). In a good IDE, you can move back and forth -- from one of these pieces to another -- easily. While this makes program development easier for skilled programmers, it can confuse beginners unless they keep in mind the differences between writing source code, compiling that source code, and running the resulting compiled program.

One of the nice (or not-so-nice) things about compiling code is that it gives you an opportunity to discover certain kinds of errors, or bugs, in your programs. For example, compilers can usually tell you when the code that you've written is not legal (Java). For example:

- \(6 + *4\)
- `else`
- `{()}

These are called **syntax errors**. Syntax errors can by typographic, like the transposition of the `s` and the `l` in `else`, or they can result from mis-remembering a name (e.g., calling something `getSize` when it's really `getDimension`). A syntax error can also be the result of bad punctuation or of accidentally commenting out more (or less) than you intended.
Because syntax errors make your code illegal, the compiler will not be able to figure out what you mean and it will complain. Unfortunately, the compiler may not trip over the bug at the point where the syntax error actually arises. Often, the compiler will do its best to figure out what you mean and only discover that it is mistaken after it's done processing the line, the block of text, or even the whole file. So when a syntax error occurs, the compiler will tell you, but it may be difficult to figure out exactly what (or where) the syntax error occurs from the compiler's error message.

As you encounter compilation errors, keep track of what the error message is, where the compiler said the error occurred and, when you find it, what and where the actual error was. After a while, you will start to see patterns in how certain mistakes in the program cause the compiler to object. Usually, it's a good idea to start at the place where the compiler reports the error and work backwards, but this can vary tremendously from one compiler to the next. Learning to understand compilation errors is a good time to have someone around to ask questions of.

When you find a compilation error, you will need to go back and re-edit the source (Java) code file. Once you've eliminated the errors reported by the compiler, you will need to compile your file again. Sometimes, eliminating one compilation error will cause others to show up. Often, a compiler will only report the first few errors it finds. Eventually, though, you will eliminate all compilation errors, compile your code, and find that you can run it. Now, you can begin to test it.

[WARNING: Remember that you need to compile successfully each time that you modify the source code; otherwise, you could be running an old version of your program!]

### 2.4.3 Scaling Up Slowly

At each step in your development plan, you will need to carefully test both new and old behavior of your code. In order to test your code thoroughly, you will want to draw on the test suite that you developed in specifying the problem -- including its use cases and guarantees -- and in designing your implementation and development plans. As you scale up the actual running software that you have built, you should continue to test basic functionality from earlier stages -- to ensure that it is still working -- along with the new functionality that you have added. It is OK to combine tests as you go along, but you should not generally drop a test entirely unless the same behavior is exercised by another test. It is important not to go on to the next step of your development plan until you understand what your code is doing at this step.

Be sure that you understand how your code actually behaves, rather than simply how you think it should behave. It is all too easy to kid yourself into believing that your code is correct. There is no better way to demonstrate your code correct than to thoroughly test it. Better, have someone else test your code for you.
Make certain that you know how your code will behave under inappropriate as well as appropriate circumstances. At all times, you should be able to describe what your code does as well as how it does it. If your code is surprising you, stop to figure out what it is doing and why. Surprises often come back to haunt you later if you don't take the time to figure them out when you first encounter them.

Write documentation of your code as you go. Documentation is information that you leave for yourself or for other software engineers who may be unfamiliar with the code that you wrote. It should describe what your code does. It should not be an English version of the code; instead, it should summarize the functional behavior of the code: what job does it do? The documentation should articulate the formal and informal contracts that the code makes, including its assumptions and guarantees. Good documentation is so important that, throughout this book, we will include style sidebars that explicitly describe what kind of documentation is essential for each of a variety Java elements.

When you get a version of your software to work -- including testing it in every way that you can think of -- you will want to keep it around even as you go on to improve it. That way, you'll still have the working version when your next modification breaks it. Keeping a version is called **checkpointing**. You should checkpoint your program whenever you have a working version and before every major revision, just in case it turns out to be a mistake. When you checkpoint, checkpoint the whole system, even if it is in multiple files. It can be hard to revert just one component of an interacting system.

Professional software developers and even advanced students often make use of versioning software to do their checkpointing. This software keeps track of different versions of your work and can help you compare these versions or even go back to an old one or merge two different sets of changes. Versioning software is a useful tool for serious software development (or for group projects, where different people may be working on different parts of the system at the same time).

You can do a simple sort of versioning yourself by periodically saving your project in a time-stamped backup. For example, this morning before you start working on your program, you can save a

```
/*
 * The following code reports
 * (to the requester) whether
 * the book is currently available
 * in the library.
 */
```

This documentation doesn't repeat what the code says, but does summarize how it behaves. In a longer, more complicated piece of code, the differences between the code and its accompanying documentation would be more apparent. A longer piece of code might also have additional information about its assumptions and guarantees. Examples illustrating these aspects appear in sections of this book where more complicated code is presented. Remember:

http://www.cs101.org/ipij/design

09/18/2003 11:40:36 AM
The most important function of documentation is to make it easier to understand and modify otherwise unknown code.

After you fix the bug that's been bothering you, you can save another copy in a folder called 09-23-1245, and when you're done programming for the day you can save that version in a folder called 09-23-2120 (9:20pm; programmers are often "night people"). The details of exactly how you do this will vary depending on the Java environment you're using, but the idea remains the same. If you think the version of your code you're currently working with is an improvement over a previous version, save a copy somewhere so that you can go back to it if your next change makes things worse.

2.5 Debugging is Normal

Debugging is normal. Everyone debugs. In fact, trial and error -- test and debug -- is a perfectly legitimate technique in building a piece of software. The important principle is that it should be informed trial and error; you should have a plan and a reason for trying the things that you are trying. (It's also a good idea to couple trial and error with a good checkpointing strategy; see above.) A good software engineer is sometimes a good experimentalist -- trying things out to learn from how they work and why they don't -- as well as a good experimental designer.

Leave room for debugging in your development process. Developing in pieces and stages so as to simplify your debugging process -- and designing in tests to verify behavior at each stage -- are important aspects of being a good software engineer. As you go, record your bugs -- the circumstances that reveal unexpected behavior, the sources of that behavior, and the solution to it -- in your engineer's notebook. Also record additional tests you might want to run later or concerns about other bugs that might arise down the line. These strategies will help prevent you from running into the same problems over again.

Debuggers can be your friends. Each particular development environment has a different debugger and it is important to understand how to work with yours. Consult your instructor or your documentation for specific help. Most debuggers let you stop your program at various points, look at objects or state, or walk through the program one step at a time. However, debuggers are not necessarily very good at working with programs where more than one thing can be going on at a time. You will need to have other tools as well.

Debugging, then, is like solving a mystery. Something is going on, and you need to find out what and why. You can approach debugging in most of the same ways that Sherlock Holmes would approach a mystery. You can sit and think. You can discuss the scenario with Dr. Watson (or anyone else who happens to be handy). You can play things out in your mind (or on paper, or with your friends). And, perhaps most importantly, you can set traps for the culprit: deliberately design experiments that will give you more information about what's going on, and where, and why, and under what circumstances this mystery arises. Your advantage (over Sherlock Holmes) is that your experiments don't run the risk of scaring off the culprit, so you can conduct as many of them as you'd like to try to solve the mystery. Just remember to save a copy of

http://www.cs101.org/ipij/design
your code at the point that the bug arose, so that you can always go back to that version, rather than the one containing experiments and traps, to fix the bug.

### 2.5.1 Entomology Field Guide: A Taxonomy of Bugs

There are many different kinds of bugs that arise in programs and, as a result, many different ways to try to catch them.

First, there are syntax errors. Syntax errors are things that you write that are not legal Java. We have actually discussed these above, in the section on compilation, because a good compiler will catch your syntax errors. Although it can be tricky to learn to understand the errors your compiler reports, you will soon learn how to read them and find the underlying problem. Eventually, you'll probably even be grateful to your compiler for finding all of these bugs for you.

Not all mistyping leads to a syntax error. For example, if you replace a + with a - in your code, you will probably still have legal Java, but it is unlikely to do what you want. This is not, strictly speaking, a syntax error (and the compiler is extremely unlikely to think there's anything wrong it at all). It is a simple kind of **logic error**. Logic errors occur when you write legal code to do the wrong thing. Generally, a logic error doesn't prevent the program from running; it just causes the program to behave strangely. Other simple logic errors include using a legitimate but incorrect name (e.g., calling verifyAvailability when you meant checkIn), or starting a counter off with the wrong value.

Simple logic errors are generally easiest to catch by adding steps to your program that print things out. For example, imagine that the library system keeps a count of how many books are currently in the library. Each time a patron returns a book, the program should add one to its counter, booksInHouse:

```
to checkIn a bookID:
  1. masterList, please put the pair bookID, LIBRARY_ID into the record
  2. add one to booksInHouse
  3. report success
```

But perhaps the program author used - instead of +, causing the number of books in the library to fall each time that a book is returned. Printing booksOut each time a checkout happens makes this error easier to find. This would mean adding an instruction (between #2 and #3) that says:

```
  2.5 print the current value of booksInHouse
```

Actually, it would be a good idea to include where this line was printed and other things about its context. It is worth investing a little bit of time to print nice debugging messages, because you will often want to reuse them again later. A better version of this message might say:
2.5 print the following things:

- the phrase "In CirculationDesk checkIn, after returning book "
- the value of bookID
- the phrase "booksInHouse is "
- the current value of booksInHouse

A debugger may also let you watch the value of a quantity like booksInHouse directly.

Logic errors can also be a bit more complicated. For example, in the design section above, we asked what would happen to the code as designed for checkOut if two patrons tried to checkOut the same book, one after another. Imagine, for example, that someone named Abbott successfully checks out (the bookID corresponding to) a book called *Who's On First?* This means that the library's master list records the bookID for *Who's On First* as being associated with the patronID for Abbott. Five minutes later, Costello tries to check out the same bookID. Using the recipe for checkOut above, masterList is asked to put the information that *Who's On First*’s bookID is with Costello's patronID. So now Costello is recorded as having the book!

The problem that this reveals is a logic error. Even if each line of code that you wrote were, in itself, correct, your program would do the wrong thing. Acting it out is, in general, an excellent way to catch logic errors, but be careful that you (or your actors) do what the program *actually says*, not what you think it should say (or wish it would do). Sometimes, the logic error is too subtle for human actors to be able to recreate it. In that case, running the program using a debugger and inspector (or inserting a lot of printing statements that tell you what each part of the program thinks and knows as it happens) might be necessary.

In this case, the fix for the logic error is pretty straightforward. Costello should not have been allowed to check out the book, because the library didn't have it. [[Footnote: If the library did have the physical book, it would be because Abbott forgot to take it home, and it would be important to clear the book from Abbott's record before checking it out to Costello!]] So we need to add another piece to the checkOut recipe: First, the recipe should verify that the library is formally in possession of the book. A revised recipe might say

to **checkOut** a **bookID** to a **patronID**:

1. masterList, please **get** bookID's record; call the associated PatronID whoHasIt
2. report failure unless whoHasIt is LIBRARY_ID
3. masterList, please **put** the pair bookID, patronID into the record
4. report success

[[Footnote:

If you look carefully, you'll see that the first half of this new recipe is actually just a CirculationDesk's verifyAvailability recipe by another name. So, instead of writing it out explicitly here, we can use this CirculationDesk's verifyAvailability action directly:
to checkOut a bookID to a patronID:

1. (ask this CirculationDesk): please verifyAvailability for bookID; this responds with bookID's availability
2. report failure unless it is available
3. masterList, please put the pair bookID, patronID into the record
4. report success

This technique -- reuse of your own recipes -- is excellent programming style whenever the recipe you're reusing has the same intent as the steps you're substituting it for. There are several reasons for this:

- Usually, the substituted steps are longer than the call to the other recipe, so this also helps keep your recipes shorter. (That's not true in this particular case, but "spread the jelly on the bread" is shorter than "repeat until the bread is full: pick up some jelly with your knife, put it on the slice of bread.")
- Even if it isn't textually shorter, recipe reuse can make your code easier to read. For example, saying "verify availability" tells you why you're bothering to get the patronID currently associated with the bookID.
- When you need to modify the code, there's only one place to do it. If the system were changed so that the library had two patronIDs -- ADULT_COLLECTION_ID and JUVENILE_COLLECTION_ID -- verifyAvailability would be modified correspondingly. In this case, the version of checkOut that calls verifyAvailability would automatically work while the version that checks the masterList directly would break.

When, why, and how to reuse code in this was is covered in the chapter on procedural abstraction.

This modification solves the sequential checkout problem. Recall, however, that we said that our library might have multiple checkout stations. What if Abbott and Costello each tried to check out the bookID for Who's On First? at the same time? This could lead to a problem even with the revised code: Each station might checkValidity, find the bookID in possession of LIBRARY_ID, and OK the transaction. Then both stations would go on to step 3, marking the book as checked out, each to a different PatronID. This kind of problem, which happens because multiple things are going on in the system at once, is called a concurrency error. Concurrency errors and their solutions are the major subject of the chapter on synchronization. They can sometimes be identified by acting things out; at other times, it is more useful to have different pieces of your program report on which steps they are executing or about to execute and what they think is happening. It is always important to keep in mind what else might be going on in your program and whether one part of it can interfere with another.

Other errors arise, for example, when you misuse a piece of code. The CirculationDesk's verifyAvailability service is meant to check whether a particular book is available, not whether the CirculationDesk itself is available. Although that's not an error you're likely to make, similar misunderstandings of other code -- especially unfamiliar code libraries -- often lead to program bugs.
2.5.2 Good Tests Catch Bugs

Designing for debugging is a fine art. So is knowing which tests to run. The logic issue above -- Abbott and Costello each checking out the same book -- is a classic test; it is just the kind of misuse your system might not be designed to prevent. Two books with the same name (but different bookIDs) might be a good test for your system. A book that is lost -- never checked back in -- could present a problem. What would happen if the CardCatalog's lookup recipe returned a BookID that the CirculationDesk didn't know about? Could this ever happen in your system? What else could go wrong? Also think about the "normal" cases: each use case should have at least one corresponding test. As you become a more experienced programmer, you will build a mental catalog of various kinds of errors and the tests that catch them. Eventually, you will recognize and anticipate these kinds of issues. For now, your engineer's notebook is a good place to start this catalog.

Timing collisions -- two things happening at just the wrong times -- are another standard kind of problem for a system. Unfortunately, timing problems can be intermittent and so they are harder to identify or reliably replicate. Again, experience will give you better intuitions as to when a timing problem might be arising and how to prompt it to reveal itself. In the interim, debugging with a skilled assistant is useful, but in all cases make sure that you understand what was wrong with your program and that you fix it yourself.

Along with the catalog of tests and bugs that you will accumulate -- in your notebook and through your experience -- there are some techniques that can help you figure out what your code is doing -- right or wrong -- and help you to trap bugs.

2.5.2.1 Make state manifest

It is much easier to tell what your code is doing if you can see it work. A debugger may let you step through your program, one line at a time. A visualizer may even let you watch how things change. However, many debuggers and visualizers don't work very well with the kinds of programs that we will be discussing in this book, programs in which more than one thing can be going on at a time. In this case, you may have to get your code to tell you what it is doing without the help of the debugger or visualizer.

You can get your code to tell you what it is doing by adding steps to each recipe that print out (on the computer screen) what is happening. We did this with line 2.5 of the CirculationDesk's checkOut recipe. It can be a good idea to include the following steps in a recipe:

- When the recipe starts, a step that prints the name of the recipe and the message "starting".
- When any major change is taking place, a step that prints the name of the recipe and information about the change.
- When the recipe finishes, a step that prints the name of the recipe and the message "completing".

You can also include steps that print out where within a long recipe the instruction-follower is. Note that each printing step begins with the name of the recipe. If there are multiple objects that run the same
recipes -- multiple CirculationDesks, for example -- it is important to include information as to which one is using that recipe step, too.

In addition to code that tells you which step in a recipe is happening, you may want to add code that can tell you about a particular object. For example, you might give each PatronID might have a recipe that prints any information associated with that patronID, such as the patronID number, the patron's name and address. [[Footnote: Java actually provides a particular recipe to do this -- it's called toString() -- but you need to supply the steps to make the toString() recipe useful. More on this in subsequent chapters.... ]] Or the CardCatalog might have a recipe to print the masterList, with all of its book circulation information. These kinds of recipes that let you see the state of things can be particularly helpful for debugging.

Even when you have fixed your bugs, don't be tempted to get rid of these recipes. You can leave them in place, but eliminate (or "comment out") the steps that invoke these recipes. After all, you never know when a new bug will pop up and you'll want to use these recipes again!

2.5.2.2 Explain it to someone.

One of the hardest things for a programmer to do is to recognize that the program is not doing what s/he thinks it should. Don't make the mistake of assuming that your logic is correct. Believe what the code is telling you. If you don't want to believe it, add more steps to get the program to tell you more about what is actually happening. Eventually, you will believe that it is doing what it says it is and, sooner or later, you'll understand why.

Sometimes, you can find a bug just by explaining how your code works. In fact, many bugs are caught by people spelling out why the behavior of the program is simply impossible....oops! The reason is actually quite simple: You know, better than anyone, how you wrote your program. You also know why you thought that the code that you wrote would do the right thing. But you probably didn't go over each detail all that carefully before you wrote the code and compiled it. When you actually take the time to explain, in detail, why you did what you did, you are likely to realize where your logic wasn't completely correct. Rumor has it that one college programming class has a requirement that a student seeking help with a program first explain the program's behavior to a teddy bear that they keep on hand for just such purposes. I'm sure that the teddy bear has found as many bugs as the lab assistants!

Imagine, for example, that your library program also keeps track of how many books are actually present in the library. But, somehow, the number of books in the library keeps getting smaller. In fact, after a while, it becomes negative! How could this be? Well, each time that the CirculationDesk runs its checkOut recipe, it reduces the number of books present in the library by one. And each time that it runs its checkIn recipe....There it is! Maybe you forgot to add one to the number of books in the library in the CirculationDesk checkIn recipe.

You'd be surprised at the number of bugs that are caught mid-way through an explanation. And if you can't explain how the program was supposed to work (in a clear, coherent, organized way) that's a telling sign,
too. In fact, explaining BEFORE coding is a good idea to get your bugs out early. Remember: Developing software is an incremental process. It cycles back and forth, extending the behavior of your program and then verifying that it does what it should. Debugging is normal, and learning to debug your programs is an essential part of becoming a software developer.

**Chapter Summary**

- Clean coding and good documentation enhance the lifetime of your software.
- Program construction is a cycle of designing, building, testing, and then designing again.

**Exercises**

1. Write the **Library-Checkout** use case.

2. Add a **Library-Reserve-Book** use case, in which I ask the library to hold a specific book for me, and a **Library-Lookup-Reserve** use case, which combines **Library-Lookup-Can't-Checkout** and **Library-Reserve-Book**. How about **Library-Checkout-Reserve** (check out a previously reserved book), which ought to look a lot like **Library-Checkout** but include a step to verify that the checkout patron is the person who held the reservation? Also include tests for these use cases.

3. Write down the scripts (as described at the end of "How do they interact?") for each of the use cases listed in this chapter:
   a. **Library-Lookup**
   b. **Library-Checkout**
   c. **Library-Lookup-Checkout**
   d. **Library-Lookup-Can't-Checkout**
   e. **Library-Checkin**
   f. * **Library-Reserve-Book**
4. Design a book reservation system that allows a patron to request that a particular book be held for his/her future checkout.

a. Which component of the library should be responsible for recording that a book is on reserve?

b. Write an action contract and the recipe for placing a book on reserve.

c. Incorporate that recipe into a script for Library-Reserve-Book

d. Describe at least one test that you could use to verify the behavior of this script

e. Write an action contract and the recipe for checking out a previously reserved book.

f. Incorporate that recipe into a script for Library-Checkout-Reserve

g. * How should the reserve system affect the existing checkout code? What could go wrong happen if you did not modify the existing checkout code? How could you fix it?

5. Add three more steps to the library development plan described in the "Developing Code" section of this chapter. Be sure to include a test sequence for each.

6. Use the original checkOut recipe. What happens if two check out requests come, one after the other, for the same book?

7. If you've edited (saved), compiled, and run your program and want to run it again, do you need to compile it first?

8. If you've edited (saved), compiled, and run your program and you modify (edit, save) your program, what do you need to do to see what the program does now?

9. * The contracts of the mail order/Books on Tape and eBook versions of the fetchTheBook recipe are intertwined with the CirculationDesk's checkOut action in ways that the library patron's or library staff's implementations are not. This in fact represents a change of contract between the two implementations. Have students act out the two versions, allowing the human user to be nasty and try to steal the book. Presume that the library has a gate that beeps if an un-checked-out book leaves the building; note that the book is already
in the patron's hands and outside any beeping machine once fetchTheBook completes in both the mail order and eBook scenarios.

10. Extend the library system described in this chapter to provide the following additional features of a library reserve system:

   a. Notification when a reserved book becomes available.

   b. Allowing a second person to reserve the book even before the first reservation is filled.

   c. Limiting the number of reservations that a single patron can make.

11. Extend the library system described in this chapter to provide the following additional features:

   a. Datestamp checkOuts

   b. Track when books become overdue

   c. Send overdue notices to patrons whose books are outstanding

   d. Charge a fine when an overdue book is checked in

   e. Refuse checkouts to a patron who has accumulated more than a specified amount in fines

12. Design an electronic calendar system with an alarm notification for each event. You may assume that you have access to a clock program library. (You may make further assumptions about the behavior of the clock library, but you must spell these out explicitly. Alternately, you may use the clock found in Chapter [@@animacies].

13. Design a restaurant system including at least 3 constituent community members. Write scripts for each of their behaviors.
Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Interlude: A Community of Interacting Entities

Overview

This interlude provides a whirlwind introduction to most of the basic concepts of Java programming. It uses a simple community of word games and other String transformers to illustrate this exploration.

This interlude is not intended to be read as standalone coverage of these ideas. Instead, it introduces many concepts only briefly, but in context. Each of the programming concepts presented here is reintroduced in much greater detail in the chapters of section 2 of this book.

Objectives of this Interlude

1. To increase familiarity with the design process.
2. To understand how to describe a system design in terms of types, components, and interactions.
3. To discover how design translates into executable code.
4. To be able to read and begin to understand fragments of Java programs.

Introduction: Word Games

When I was a child, we used to amuse ourselves by speaking to one another in a special language called Pig Latin. The simplest version of Pig Latin has just one rule: To turn an English word into a Pig Latin one, you take the first letter off the word, then add the first letter plus "ay" to the end of the word. So, for example, "Hello" in Pig Latin is "ello-Hay", and "How have you been?" is "ow-Hay ave-hay ou-yay een-bay?" There are more sophisticated rules for Pig Latin that deal with consonant blends and words that begin with vowels, but the basic idea remains the same. It turns out that there are children's games like Pig Latin in many, many languages, though each has a slightly different set of rules. Another such game, popularized by the children's Public Television show Zoom, is Ubby Dubby, in which you add "ubb" before every vowel (cluster): "Hubbellubbo", "Hubbow hubbave yubbou bubbeen?"

This interlude explores such word- and phrase- transformations. In fact, we're going to build a system in which you can have many of these different transformers, and you can glue them together in almost any order. In this sense, the transformers will be interconnectable modules like Lego(tm) or Capsela(tm).

[Insert pictures of various objects here.]

In addition to transformers such as Pig Latin and Ubby Dubby, we'll want capitalizers ("HELLO"), name droppers ("Lynn says Hello", or "Chris says Hello", or "Pat says How are you doing?"), even delayers (e.g., that don't produce "Hello" until after they've already received "How are you doing?") or network-
senders (that can move one of these strings-of-words from one computer to another). We'll also have some community members that can read information that a user types to them or display information on a computer screen. And we'll have transformers that can take listen to two different inputs, producing only one output, as well as transformers that can produce two outputs from only one input. (The first of these is a collector; the second is a repeater. The first is good when you have lots of people trying to talk all at once; the second is a nice way to circulate (or broadcast) information that needs to get to a lot of people.)

In the system that we're going to explore, we will need a way to create individual transformer-boxes like the ones described above. We'll also need a way to connect them together. Finally, the transformer-boxes will need to act by themselves, to read inputs, do transformations, and produce outputs. The complete system will be a community of interacting entities, many of which will themselves be communities. At the most basic level, each of these entities will need to follow specific instructions. In this interlude, we will explore both the design of the community and the specific instructions that some of these entities will follow.

Designing a Community

We need to design

- What behavior does the system provide?
- Who are the members of the community?
- How do they interact?
- What goes inside each one?

We can start at the bottom (bottom-up design) or at the top (top-down design). Both are legitimate and useful design techniques. However, design in practice often mixes these techniques. In this case, we're actually going to start in the middle; in this particular system, that is one of the easiest places to begin thinking about what we want to produce.

At the end of the design process, we should be able to sketch out a scenario for each of the major interactions with our system, including what roles need to be filled (i.e., the types of things in our system), who fills these roles (i.e., the individual objects that make up the system), and how they communicate among themselves (i.e., the flow of control among these objects).

A Uniform Community of Transformers

There are several communities implicit in the system that we're building. Let's start in the middle, where the system can be understood as a community of interacting transformers. In this picture, each transformer is an entity. The interactions in this community are quite simple: Each string transformer reads in a phrase and writes out a transformed version of it. In this system, we want to be able to interconnect these transformers in arbitrary ways. This means that the services each transformer provides will need to be compatible, so that one transformer can interact with any other transformer using the same connection mechanism.

Transformer Entity interactions, version 1

- Read a word/phrase (from a connection)
Write a word/phrase (to a connection)

We will accomplish this generic connection between transformer entities using a computer analog of the tin can telephones that we built as children. [Footnote: Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.] This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end. The computer analog will be Connection objects that allow one transformer to write a word or phrase and another transformer to read it from the connection. The transformers on either end don't have to know anything about one another; they can simply assume that the transformers will interact appropriately with the Connection. And the connections don't have to know much of anything about the transformers, either

**Connection Entity interactions**

- Accept a word/phrase written to you
- Supply a word/phrase when requested (read)

Connections provide one particular way of providing interconnections among objects. In this system, the components are designed so that any outputter can be connected to any inputter. In other parts of this book, we will see examples of other kinds of interaction mechanisms. For example, in some systems, the pieces to be interconnected are not uniform. In others, the particular choices of interconnections must be made at the time that the system is designed rather than while the system is running. In part 4 of this book, we will pay particular attention to the tradeoffs implicit in different interconnection mechanisms.

**The User and the System**

Before we look at how each transformer (and connector) is built, let's step back from this community of interacting transformers to ask how it came into existence. At this level, the members of our community are the user who constructs the community and the system to be constructed. The user expects the system to provide a way to create transformer entities and a way to connect them.

**System/User interactions**

- Create a Transformer (of a specified type)
- Connect two Transformers (in a particular order)

![Picture of Control Panel & transformers.]

We'll accomplish the first of these by adding another entity to the community: a user interface containing a control panel that allows the user to specify that a transformer should be created as well as what type of transformer it should be. The second interaction, connecting transformers, we will handle by letting the user specify two transformers (through the user interface) and then asking the specified transformers to accept a new connection. So allowing the system to interact with the user creates one additional entity (the user interface) and adds an interaction to the transformer:

**User Interface interactions**
Create a Transformer (of a specified type)
Create a Connection between two Transformers

Transformer Entity interactions, version 2

- Accept an input Connection [Footnote: Maybe more than one.]
- Accept an output Connection
- Read a word/phrase (from a connection)
- Write a word/phrase (to a connection)

Specifically, the Control Panel will have buttons representing each kind of transformer available. Clicking on a button will create a new transformer of the appropriate type. Clicking on first one transformer, then another, will create a connection between them. This task is actually cooperative: the user interface will create the connection and it will ask the Transformers to accept it.

**What Goes Inside**

In the two subsections immediately above, we've designed transformer-transformer interactions (via connections) and user-system interactions (via the user interface). We've addressed the question of who our community members are (UI, transformers, connections, and -- stepping back -- the user) and, to a first approximation, how they interact. In terms of system design, transformers and connectors represent kinds of things of which there may be many separate instances. For example, a particular community of transformers may contain five transformers and four connectors, or eight transformers and three connectors, or twelve. Each community will contain only a single control panel, though.

The next step in a full design process would be to look inside each of these entities to discover whether they are, themselves, monolithic or further decomposable into smaller communities. We will not decompose the user interface further in this chapter; much of the necessary background for this task will not be introduced until part 3 of this book. Instead, the remainder of this interlude will look inside the transformer type to see how these objects are built.

**Building a Transformer**

We have seen above the specification of the interactions that a Transformer Entity will be expected to fulfill. We can turn this interaction specification around to provide a specification of the behavior that an implementation will need to satisfy: A Transformer must be able to:

- Accept an input Connection
- Accept an output Connection
- Have its own instruction-follower that acts independently to read its input, transform that input as appropriate, and write its output.
In fact, this Transformer is itself a community. The connection acceptors are each entities that are activated only on a connection accept request; their jobs are to remember the connections that they have been handed. For example, the acceptInputConnection instructions basically say, “To accept an input connection (let's call it \texttt{in}), simply store \texttt{in} away somewhere so that you can use it later.” There's also a little bit of additional code to say what to do if you've already got an input connection stored away. Output connections -- another part of the community inside an individual Transformer -- are handled in the same way as input connections. Also, some kinds of Transformers will have code that needs to be run when an individual Transformer is created. Finally, the independent instruction-follower is an additional ongoing interacting entity. It makes use of the connections (such as \texttt{in}) that the connection-acceptors have stored. Each transformer will have its own instruction follower, allowing the transformer to do its work without any other entity's needing to tell it what to do.

For the moment, we will focus on the heart of the Transformer, the work done by this independent instruction-follower, especially the transformation it actually performs. We begin by looking at some specific Transformers and describing the behavior we expect.

**Transformer Examples**

The instructions for the behavior of a Capitalizer will say

1. Read the input.
2. Produce a capitalized version of it.
3. Write this as output.

Every individual Capitalizer is the same, and each one does the same thing. You can tell them apart because they're connected to different parts of the community and are capitalizing different words, though.

NameDropper is a different kind of Transformer. Each individual NameDropper has its own name that it likes to drop. So the instructions for a NameDropper will say

1. Read the input.
2. Produce a new phrase containing your name, the word "says", and the input.
3. Write this as output.

Variations in Transformer behavior aren't restricted to the transformation itself. Yet another kind of Transformer is a Repeater. The repeater is different because it can accept more than one OutputConnection: two, in fact. The instructions for a Repeater say:

1. Read the input.
2. Write this to one OutputConnection.
3. Write this to the other OutputConnection.

And, of course, the instructions for a (simple) PigLatin should say

1. Read the input.
2. Produce a new phrase containing all but the first letter, then the first letter, then the letters "ay".
3. Write this as output.

As you can see, the basic instructions for a Transformer are of the form

1. Read the input.
2. Produce a transformed version of it.
3. Write this as output.

We will begin by looking at the second of these instructions.

**Strings**

In Java, there is a special kind of object, called a String, that is designed to represent these words or phrases. In fact, in Java a String can be almost any sequence of characters typed between two double-quote marks, including spaces and most of the funny characters on your keyboard. (The double quotes aren't actually a part of the String itself; they simply indicate where it begins and ends.) For example, legitimate Java Strings include "Hello" and "this is a String" and even "&((())&%'^". (Strings don't have to make sense.) The Transformers that we will build are really StringTransformers, since each one takes in a String at a time and produces a corresponding, potentially new or transformed String as output.

**String Concatenation**

Once you have a String, there are several things that you can do with it. For example, you can use two Strings to produce a third (new) String using the String concatenation operator, +. In Java,

```
"this is a String" + "%%^$^&&)) mumble blatz"
```

is for all intents and purposes the same as just typing the single String

```
"this is a String%%^$^&&)) mumble blatz"
```

[Footnote: Note that there is no space between the g at the end of String and the % at the beginning of %%%^$^&&)])] So, for example, a NameDropper transformer might use + to create a new String using the input it reads, the name of the particular dropper, and the word "says". Pig Latin and Ubby Dubby might use +, too, but they'll have to pull apart the String they read in first.

**String Methods**

Java Strings are actually rather sophisticated objects. Not only can you do things with them, they can do things with themselves. For example, you can ask the String "Hello" to give you a new String that has all of the same letters in the same order, but uses only upper case letters. (This would produce "HELLO".) The way that the String does this is called a method, and you ask the String to do this by invoking its method. In this case, the name of the method that each String has is `toUpperCase()`. You ask the String to give you its upper-case-equivalent by putting a . after the String, then its method name:

```
"Hello".toUpperCase()
```
yields the same thing as "HELLO".

You can also ask a String for a substring of itself. In a String, each character is numbered, starting with 0. (That is, the 0th character in "Hello" is the H; the o is the 4th character.)[Footnote: Computer scientists almost always number things from 0. This is apparently an occupational hazard.] So you can specify the substring that you want You can do this by supplying the index of the first character of the substring, or by supplying the indices of the first and last characters. "Hello".substring(3) is "lo"; "Hello".substring(1,3) is "ell"; and "Hello".substring(0) is still "Hello".

These and other useful functions are summarized in the sidebar on String Methods.

Selected String Methods

Below are some selected methods that can be invoked on individual Strings, along with brief explanations and examples of their usage.

- `toUpperCase()` produces a String just like the String you start with, but in which all letters are capitalized. For example,
  
  "MixedCaseString".toUpperCase()

  produces
  
  "MIXEDCASESTRING"

- `toLowerCase()` produces a similar String in which all letters are in lower case. So
  
  "MixedCaseString".toLowerCase()

  produces
  
  "mixedcasestring"

- `trim()` produces a similar String in which all leading and trailing white space (spaces, tabs, etc.) has been removed. So
  
  "      a very spacey String      
  "  .trim()

  is just
  
  "a very spacey String"

- `substring( fromIndex )` produces a shorter String containing the same characters that you started with, but beginning at index `fromIndex`. Bear in mind that the index of the first character of a String is 0.

  `substring( fromIndex, toIndex )` produces the substring that begins at index `fromIndex` and ends at `toIndex - 1`.

  "Hello".substring(3) is "lo"
  "Hello".substring(1,4) is "ell", and
  "Hello".substring(0) is "Hello" again.

- `length()` returns the number of characters in the String. For example,
  
  "Tee hee!".length()

  is 8. Since the String is indexed starting at 0, the index of the final character in the String is the
  String's length() - 1.

- `replace( old, new )` requires two characters, `old` and `new`, and produces a new String in which each occurrence of `old` is replaced by `new`: [Footnote: A character is, roughly, a single
alphanumeric or symbolic character (one keystroke) inside single quotation marks. For more detail on what exactly constitutes a character, see the chapter on Java types.] For example,

    "Tee hee!".replace('e', '*')

produces

    "T** h**!"

• charAt ( pos ) requires an index into the String and returns the character at that index. Recall that Strings are indexed starting at 0.

    "Hello".charAt( 2 ) is the same as "Hello".charAt( 3 )

• indexOf ( character ) returns the lowest number that is an index of character in the String.

    "Hello".indexOf( 'H' ) is 0 and
    "Hello".indexOf( 'l' ) is 2. Also,
    "Hello".indexOf( 'x' ) is -1, indicating that 'x' does not appear in "Hello".

• lastIndexOf ( character ) returns the highest number that is an index of character in the String.

    "Hello".lastIndexOf( 'H' ) is 0 and
    "Hello".lastIndexOf( 'x' ) is -1, but
    "Hello".lastIndexOf( 'l' ) is 3.

Rules and Methods

Using the String manipulations described in the previous section and sidebar, we can construct the instructions that a variety of Transformers would use to transform a String. For example, we might write:

    to transform a String ( say, thePhrase ),
        return thePhrase.toUpperCase();

This rule describes the transformation rule for an UpperCaser. Note that the String is intended to stand in for whatever String needs to be transformed. The transformation rule can't operate unless you give it a String. Within the body of the transformation rule, a temporary name (in this case, thePhrase) is used to refer to this supplied String. The formal term for such a piece of supplied information is an argument, and the formal term for the temporary name that is used to refer to it is a parameter.

A different transformation rule -- this one for a pedantic Transformer that seems to think it knows everything -- might say

    to transform a String ( say, whatToSay ),
        return "Obviously " + whatToSay;

Note that we have chosen a different temporary name to represent the String argument. The parameter name doesn't matter; we can choose whatever (legal Java) name we wish.[Footnote: Legal Java names are covered in the sidebar on Java names in the chapter on Types.] It can be the same name in every transformer rule, or different in each one. It is only important that we use the same name in a particular rule both when we're specifying the parameter (in the first line of the rule) and in the body of the rule.
Q. Can you think of another kind of Transformer and write its rule? Remember, it should take a String and produce a String.

The rules as we've presented them aren't really Java code, but they are pretty close. To make them legal Java, we need to add a bit more formality and syntax. The formal name for a rule in Java is a method, just like the String methods -- toUpperCase(), substring(index), etc. -- above. Somewhere, someone has provided instructions for how to toUpperCase() so that you can use that method without worrying how it is done. Here, we are providing the instructions for transform, so that someone else can use it.

A definition of UpperCaser's transform method might say:

```java
String transform ( String thePhrase )
{
    return thePhrase.toUpperCase();
}
```

Aside from the syntax (the details of which are covered in chapters 6 and 7), the one big difference from the rule specification above is that the method definition begins with the word String to indicate that the method will produce a String when it is invoked.

Q. Quick quiz: How would you write the pedantic Transformer's transform method?

Classes and Instances

What we just described was how to specify a rule. This rule is the rule used by all Transformers of that particular type. In fact, the rule is really the only thing that distinguishes Transformers of that type from other Transformers. We can describe a type of Transformer by wrapping the method definition in a bit of code that says it's a type. In Java, a type that provides instructions implementing behavior is called a class.

```java
class UpperCaser extends StringTransformer 
{
    String transform ( String thePhrase )
    {
        return thePhrase.toUpperCase();
    }
}
```

This says that UpperCaser is a type (or class) that is very much like the more general class StringTransformer. Its behavior differs from generic StringTransformers by using the particular transform rule contained inside the braces {} that delineate UpperCaser's body.

Pedant is similar:

```java
class Pedant extends StringTransformer 
{
```
String transform ( String whatToSay )
{
    return "Obviously " + whatToSay;
}

Q. A class that uses your transformer rule should be very much like these. Can you write it?

These classes are descriptions of what an UpperCaser or a Pedant should do. They are not UpperCasers or Pedants themselves, though. They're really more like recipes from which a particular UpperCaser or a particular Pedant can be made. To make an UpperCaser, you use the special Java construction expression new UpperCaser(). Think "cooks up" a particular UpperCaser using the recipe we just wrote. A Pedant is created similarly, but using a different recipe: new Pedant(). If we say it again, we can "cook up" another Pedant: new Pedant().

Stepping back, this is exactly what we want the buttons on our control panel to do. Pressing the button marked Pedantic Transformer should invoke the expression new Pedant(), causing an Pedant to appear on our screen. Pressing it again should invoke it again, making a second Pedant appear. We can connect these two together using other user interface functions. Now, if we send the String "I'm here!" through a Connector to the first Pedant, it should send the String "Obviously I'm here" to the second Pedant, and the second Pedant should produce "Obviously Obviously I'm here".

Q. Connecting a Pedant's output to an UpperCaser's input and supplying the Pedant with "not much" will produce "OBVIOUSLY NOT MUCH". What happens if you connect an UpperCaser's output to a Pedant's input?

Q. How about Pedant, then Pedant, then UpperCaser, then Pedant? Then UpperCaser?

Fields and Customized Parts

You can already see from the examples in the previous subsection how one class, or type, can describe many different instances. For example, phrases passed through the first Pedant contain at least one "Obviously" at the beginning; phrases passed through the second Pedant will begin with at least two "Obviously"s. But to really appreciate the power of multiple distinct instances of a type, we need to look at a type that has local state associated with each instance. The NameDropper Transformer type is a good example of this.

The transformation rule for NameDropper is

to transform a String ( say, thePhrase ),
    return my name + " says " + thePhrase;

But my name here isn't a parameter. It isn't a piece of information that is supplied to the NameDropper each time the NameDropper performs a transformation, the way that thePhrase is. Instead, my name is a persistent part of the NameDropper. And it is a part of the particular NameDropper instance, not a part of the NameDropper type. After all, each NameDropper drops its own name.
So where does this name come from? As each individual NameDropper is created, it must be supplied with a name. Then, the particular NameDropper remembers its own name, and when it comes time to transform a String, the NameDropper uses its own name.

To do this, we need to create a local storage spot that sticks around between transformations. This is done using a special kind of name that is associated with the NameDropper instance. Such a name is called a field. In this case, we'll use a field called name, because that's what it will hold. To make it clear in our code that we're referring to a field, we use a syntax sort-of like saying my name; we refer to the field using this.name. In Java, this is a way of letting an individual instance say "my own".

So the actual transform method for NameDropper should read:

```java
String transform ( String thePhrase )
{
    return this.name + " says " + thePhrase;
}
```

This way, if one NameDropper has the name Pat and another has the name Chris, Pat would transform the String "Hello" into "Pat says Hello" while Chris would make it "Chris says Hello".

[Picture of Pat and Chris transforming the same String, with field visible.]

This method definition needs to be embedded in a class, of course. We also need to add a bit more machinery to the class to make sure that the name is available when transform needs it. The first change is to actually create a place to put the name; the second is to write explicit instructions as to how to create a NameDropper so that it has a name from the very beginning. This second -- constructor -- rule will need to say:

```java
to construct a NameDropper with a String ( say, whatTheNameShouldBe ),
    assign my name the value of whatTheNameShouldBe;
```

When we translate this into Java using the special syntax for a constructor rule, it looks like this:

```java
NameDropper( whatMyNameShouldBe )
{
    this.name = whatMyNameShouldBe;
}
```

So the whole NameDropper class reads:

```java
class NameDropper extends StringTransformer
{
    String name;      // the persistent storage,
    // a permanent part of each NameDropper

    NameDropper( whatMyNameShouldBe )
    {
        // the creation rule
        this.name = whatMyNameShouldBe;
    }
}
```
```java
String transform ( String whatToSay )
{
    // the transform rule
    return this.name + " says " + whatToSay;
}
}
```

Now, when we invoke NameDropper's construction method, we give it a parameter:
new NameDropper( "Pat" ), for example.

We have actually seen -- or at least alluded to -- a similar situation earlier. When discussing the other entities that together constitute a Transformer, we said that the input-connection-acceptor's job was to stick the input connection it receives somewhere where the rest of the Transformer community can use it. Like NameDropper, the generic StringTransformer accomplishes this using a field.

Fields, methods, and constructors are the building blocks of Java objects. We will see each of these things in action in the next several chapters. In chapter 7, on Classes and Instances, we will go through each of these items in greater detail. For now, it is enough to have a general sense of how things fit together.

**Generality of the approach**

In writing this code, we have relied on the existence of a generic StringTransformer class. In that class, we include rules for how to accept an input connection (using a field to store it away), how to accept an output connection, and how to create an individual StringTransformer, including creating its own instruction follower to explicitly invoke the transform method over and over again on each String read from the stored input connection. The ways in which this StringTransformer class is put together are much like the ways in which the examples here are constructed, but the StringTransformer class is about four times the size of the classes described above. The complete code for StringTransformer is included in the on-line supplement to this book.

The transformers that we have written here each obey the same general rules and interfaces. Each defines a transform method that takes a String and returns a String. The apparent uniformity among StringTransformers makes it possible for the connection mechanism that we outlined in the previous section to work with each of them. The differences among StringTransformer behaviors are hidden inside the transform method that each of them implements. In the course of this book, we will see many different cases in which hiding behavior behind a common interface makes a system more general and more powerful. Good design specifications are crucial; they amount to deciding in advance how entities will interact.

**Summary**

In this chapter, you have been exposed to many of the most basic pieces of Java programming. None of these has been presented in sufficient detail to achieve mastery of it. Each of these topics will be revisited, most in the next part of the book. But the example described above gives a context within which to place the detail that occupies the next several chapters.
In the next chapter, we will explore the role of types in Java systems and the relationship between types and names. The final chapter of this section looks at interfaces, the contracts that one type of object makes with another. In the next section, we turn to expressions -- such as method invocation, field access, instance construction, and even String concatenation -- and learn how evaluating an expression produces a value of a specified type. Expressions are combined to make statements, the step-by-step instructions of Java code that produce behavior and flow of control. Classes allow us to implement behavior and to encapsulate both instructions and local state -- such as the NameDropper's name -- into individual objects. And self-animating objects contain their own instruction followers that execute sequences of instructions over and over, communicating with other objects and interacting to provide desired behavior on an ongoing basis.

**Suggested Problems**

See the text for things marked with a Q. Also:

1. Implement LowerCaser.

2. Implement SentenceCaser (1st letter capitalized, rest not).

3. Implement Pig Latin.

4. An improved Pig Latin would leave the first letter in place if it were a vowel, and add -way instead. This requires understanding basic conditionals and flow of control. (See Statements.)

5. Ubby Dubby is pretty hard. You may want to look carefully at the chapter on Dispatch.

6. Combiners and Repeaters involve extending StringTransformer in other ways, overriding acceptInputConnection or acceptOutputConnection. (See the online code supplement for StringTransformer source code.)

7. Really challenging problem: extract words, one word at a time, only reading an input when all words have been used up.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Chapter 3.

Things, Types, and Names

Chapter Overview

- What kinds of Things can computers talk about?
- How do I figure out what they can do (or how they interact)?
- How can I keep track of Things I know about?

This chapter introduces some of the conceptual structure necessary to understand Java programs. It begins by considering what kinds of things a program can manipulate. Some things are very simple--like numbers--and others are much more complex--like radio buttons. Many complex things can actually act, either by themselves (e.g. a clock that ticks off each second) or when you ask them to (e.g. a radio that can play a song on request). These complex things are called objects.

To manipulate objects, you need to understand two important concepts: types and names. A type tells you what kind of thing an object is. It tells you what kind of behavior you can expect from that object. A type is a way of setting expectations -- for the computer program and for the software developer -- about the object. In Java, the type of a thing lets the computer keep track of what that thing can do.

A name is a way of referring to a thing that already exists. A name doesn't bring a thing into existence, but it is a useful way to get hold of a thing you've seen before. Every name has an associated type, which tells you what sorts of things the name can refer to. It also tells you what you can expect of the thing that that name refers to.

In other words, a type describes how you can interact with the thing that a name names. There are actually two different kinds of names in Java: primitive (dial) names and reference (label) names.
Sidebars in this chapter cover the details of legal Java names, Java primitive types, and other syntactic and language-reference details.

**Objectives of the Chapter**

1. To recognize Java types.
2. To understand that documentation for a type tells you what objects of that type can do.
3. To be able to declare and define names.
4. To understand that a declaration permanently associates a type with a name.
5. To understand how a label name can have a referent or have no referent (i.e., be null).
6. To be able to tell when the values associated with two names are equal.
7. To distinguish Java object from primitive types.
8. To recognize that each dial name contains exactly one value at any time.

### 3.1 Things in Programs

In the first section of this book, we have explored what computational systems are and how they are created. In this section, we shift our focus to the pieces out of which such systems are created. The previous section took a top-down approach to system design. In this section, we learn the basic building blocks that enable us to also approach the problem from the bottom up.

In building a computer program, we will make use of many Things. Some of these Things are relatively simple and, when we need them, we can just write them down. For example, 4 and -6.3 are Things we might want in our program, and we can put them there just by writing them down in the appropriate places. But other Things -- like the library's record for *Moby Dick* or the current location of the cursor on the computer screen -- are harder to write down explicitly in the way that we are able to write 4 or -6.3. For these kinds of Things, we need a way to refer to them without explicitly writing them down each time. We do this using a name.

A **name** is a way to refer to something in your program. At any given time, a name refers to at most one Thing. What the name refers to may change over time, though, so the current location of the cursor may
be at coordinates (26, 155) now and at (101, 32) later. This is true in real life, too: the student whose birthday comes next changes every time someone celebrates becoming a year older.

In this chapter, we explore how names refer to Things in Java. We also explore some specific kinds of Things that can be included in a Java program. Although there are details in this chapter that are particular to Java, many of the principles we identify work in almost any computer programming language. These include:

- Names can be used to refer to Things, allowing you to hold on to a Thing even if you can't spell it out entirely.
- Some Things can be written directly into your programs. These are called literals.

In addition, in Java and in many other languages, each name can be used only to refer to Things of a particular type. A language in which a name is restricted to refer to Things of a particular type is called a strongly typed language. In a strongly typed language, like Java,

- A name must be declared, meaning that you must say explicitly in the program what type of Thing this name can refer to.

We will learn more about types and names below. To begin, let's look at Things themselves. What kinds of Things exist in computer programs?

[[ Footnote: You may have noticed that the word Thing is capitalized almost every time that it appears in this chapter. This is to indicate that, in this chapter, we are using it in a very particular way. In Java, no one word refers to all of the objects -- like KlingonStarships and BookIDs -- and also all of the simpler things in programs -- primitive data like 6 and -43.5 -- at the same time. To avoid using another word incorrectly or imprecisely, we'll use Thing with a capital 'T' to refer to all of the entities, objects, and primitive data that appear in programs. ]]

3.1.1 Most Java Things are Objects

We have already seen a number of different kinds of program Things. Some -- like 4 and -6.3 -- probably look quite familiar. Others, like Moby Dick or the current location of the cursor on the computer screen, may look different once we represent them inside the computer, but they, too, will be Things we can talk about in our Java programs.

Different programming languages allow you to talk more or less easily about different kinds of Things. Some languages -- like Basic -- are very restricted in the kinds of Things that a program can talk about explicitly. Other languages -- like Java -- allow you to create and talk about almost any kind of Thing that you can imagine. Java is one of a family of languages called object oriented programming languages. In Java, as in most object oriented programming languages, most Things -- and all of the Things that you will create -- are called objects. Almost any Thing that you can describe in a programming language can be an object, so, in Java, object is a pretty general word for "programming language Thing". Not all programming languages allow you
to talk about objects, though, and -- towards the end of this chapter -- we will learn about a few Java Things that are not objects.

An object can be just about anything that you might want to represent in a computer program. Some example objects include the radio button that the user just clicked, the window in which your program is displaying its output, or the url of your home page. The Klingon starship racing across your computer screen -- in a Star Trek game -- is a Thing that the computer has to keep track of. If your Star Trek game is written in Java, that Klingon starship is almost certainly an object.

Objects are frequently complex Things with internal state. For example, the window may have a "close me" control or a background color; the url has a host computer name. Many objects also know how to act and can perform tasks. The radio button may be able to tell you whether it has been selected, or the window may know how to close. The library checkout desk of the previous chapter keeps track of book circulation while the card catalog can tell you who wrote what books. And, of course, the Klingon starship has a position and a velocity and a certain amount of ammunition left to shoot at you with....

Some objects can even act on their own without being asked to do anything; they are "born" or created with the ability to act autonomously. For example, an Animator may paint a series of pictures rapidly on a screen, so that it looks to a human observer like the picture is actually moving. The Animator may do this independently, without being asked to change the picture every 1/30th of a second. Similarly, an alarm clock may keep track of the time and start ringing when a preset time arises. A starship may move -- changing its position on the screen -- or shoot at you or even explode.

Objects are useful because we can ask them to act -- or have them take actions on their own. But how do we know what we can do with a particular Java object? In order to use an object, we need to know its type.

### 3.1.2 A Type Tells You What You Can Do With A Thing

A type tells you what kind of behavior you can expect from an object. We have been using the idea of types all along, but we have not yet made it concrete and specific. For example, when we talk informally about a library book's representation in the computer or even something simple, like a number, we mean to be referring to a specific kind of Thing. In a programming language, we also talk about categories of Things this way. In English, we can refer informally to the kind of Thing, like "a number". In a programming language, we need to say what we mean by a number. In this chapter, we'll be using types that someone else has defined. In the next several chapters, we'll be learning about how to figure out what a certain type of Thing is like. In chapter [Classes and Objects], we will at last learn how to build our own types of Things.

#### 3.1.2.1 What a Type Is

A type specifies what a Thing can do (or what you can do with a Thing). Types are like contracts that tell you what kinds of interactions you can have with Things. What an object's structure is -- and what that object can do, or what you can do with it -- is fully determined by its type, i.e., what kind of object it is.
In the previous chapter, we saw a number of Things that make up a library system: the circulation desk and the card catalog, book IDs and patron IDs. If we were to write Java code corresponding to the system that we designed there, we would make each of these kinds of Thing a type of object. The description for the type CirculationDesk would tell us that a CirculationDesk can checkIn a bookID or checkOut a bookID to a patronID, for example. In other words, the type of the Thing -- CirculationDesk -- tells us what a particular CirculationDesk -- like the Smalltown Library's -- can do.

Java itself has certain predefined object types, such as DialogBox -- the little window that pops up on your computer screen to give you an urgent system message -- and URL -- the internet address of a web page. If you are using the cs101 course libraries, you'll also have access to object types such as AnimateObject -- something that can move by itself -- and DefaultFrame -- an easy kind of user interface window to use.

Other object types may include KlingonStarship (if you're building a space battle adventure game), IllustratedBook (if you're building an electronic library system), or PigLatinTranslator (if you're building a networked chat program). In the rest of this book, you will be learning to define object types -- and create instances of those types -- to do what you want. These types -- whether a part of the Java language, included in a library you choose to use, or of your own defining -- are all kinds of objects. Note that, by convention, the name of each object type -- each class -- starts with a capital letter.

### 3.1.2.2 Types and Instances

A type itself doesn't do anything; it is objects of that type that are generally useful in one's programs. These individual objects are sometimes called instances (of the type). So KlingonStarship is a type, but the particular Thing streaking across your computer screen, phasers blasting, is an individual KlingonStarship instance.

A particular object types may describe many different individual objects -- the three specific KlingonStarships visible on your screen, the five hundred and seven IllustratedBooks in the children's library, or the particular PigLatinTranslator that your particular chat program is using. These instances share a type -- they are the same kind of Thing -- but are independent objects. For example, the KlingonStarship that you just destroyed is a different KlingonStarship instance from the one that is getting ready to fire its phasers at you. We will explore this idea in greater detail in chapter 7. By convention, an individual object -- as opposed to its type -- is generally given a name that starts with a lower case letter.

Each kind of object -- each object type -- determines what individual objects of that type can do. For example, windows can close; dictionaries can do lookups; KlingonStarships can fly around the screen. Each particular kind of object provides a particular set of services or actions that objects of that kind can do. Further, each individual object of that type can perform these actions. For example, if myWindow and yourWindow are two different window-type objects, myWindow can close, and so can yourWindow. But if myWindow closes, that doesn't in general affect yourWindow.
3.1.2.3 Asking for Action

Each individual object comes ready-made with certain type-specific properties and behavior.\[ Footnote: It is tempting to say that this is typical behavior, and indeed this is just what the word typical means: pertaining to the type. \] An IllustratedBook has an author and an illustrator, for example. A PigLatinTranslator may be able to translate a word that we supply it into Pig Latin. We ask objects to act (including telling us about themselves) using specific services -- behaviors -- that these objects provide.

A common thing to do in a computer program is to ask a particular object to perform a particular service.\[ Footnote: Also to engage in a specific behavior, to take a particular action. As we shall see, the formal name for this is "method invocation". \] An object's services are requested by giving the name of the object we're asking followed by a dot (or period), followed by the request we're making of the object. So if theLittlePrince is the name of an IllustratedBook,

```java
theLittlePrince.getAuthor()
```

would be a request for the name of the author of the book: "Maurice de Saint Exupery". Similarly, if myTranslator is a PigLatinTranslator, myTranslator.processString("Hello") might be a request to myTranslator to produce the Pig-Latin-ified version of "Hello", which is "ello-Hay". We will explore this service-requesting further in the next few chapters, but for now you can regard it as a special incantation to use when you want to ask a Thing for something that it knows how to do for you. These requests are the most basic form of interaction among the entities in our community.

Sometimes, the same Thing can be viewed in different ways, i.e., as having multiple types. For example, a person can be viewed as a police officer or as a mother, depending on the context. (When making an arrest, she is acting as a police officer; when you ask her for a second helping of dessert, you are treating her as a mother.) A Thing's type describes the way in which you are regarding that Thing. It does not necessarily give the complete picture of the Thing.

3.1.3 Some Useful Types of Things...

Now that we have seen how objects and types work, let's look at some useful objects and object types. We have begun -- this chapter and your programming experience -- by using objects and types that are provided to us, because it is important to learn how to interact with an object. Many of the objects that you will use throughout your programming career will be designed by other people for you to use. Later -- in chapter [Classes and

Java Notes

Selected String Behavior

Below are some selected services provided by individual Strings, along with brief explanations and examples of their usage.

- `toUpperCase()` produces a String just like the String you start with, but in which all letters are capitalized. For example, "MixedCaseString".toUpperCase() produces `MIXEDCASESTRING`
- `toLowerCase()` produces a similar String in which all letters are lowercase. So "MixedCaseString".toLowerCase() produces `mixedcasestring`
- `trim()` produces a similar String in which all leading and trailing white space (spaces, tabs, etc.) has been removed. So
Objects] -- we will learn how to build objects of our own.

One particularly useful kind of object -- built in to Java and to most programming languages -- is called a string. A string is a piece of text, a sequence of characters. To describe a specific string -- for example, the message that your computer prints to its screen when you first boot it up -- you can write it out surrounded by double quotation marks: "Hi, how are you?" or "#^$$&&**$" or even "2 + 2" are strings. Note that the quotation marks are not actually part of the string; they're just there to make it clear where the string begins and ends. Your computer doesn't understand the string, it just remembers it. (For example, the computer doesn't know of any particular relationship between the last example and the number 4 -- or the string "4".)

Strings are useful, for example, to communicate with a program's user. Error messages, user input (i.e., what you type to a running Java program), titles and captions are all examples of strings. Strings can perform some (moderately) interesting tasks. For example, if myName is "Rigoberto Manchu", then myName.toLowerCase() is "rigoberto manchu" and myName.length() is 16.

Strings are part of the Java language, so they are available to every Java program. For more information on the String type and what it can do, see the sidebar on Selected String Behavior.

Another useful object is Console, described in the next sidebar. Console is an object that can print a String to the Java console: a standard place on the computer screen where someone running a Java program can look for information. Console can also readln a String that the user types to the Java console.

"a very spacey string".trim() is just "a very spacey string".

substring(fromIndex) produces a shorter String containing the same characters that you started with, but beginning at index fromIndex. Bear in mind that the index of the first character of a String is 0. substring(fromIndex, toIndex) produces the substring that begins at index fromIndex and ends at toIndex - 1.

"Hello".substring(3) is "lo",
"Hello".substring(1,4) is "ell", and
"Hello".substring(0) is "Hello" again.

length() returns the number of characters in the String. For example, "Tee hee!".length() is 8. Since the String is indexed starting at 0, the index of the final character in the String is the String's length() - 1.

replace(old, new) requires two characters, old and new, and produces a new String in which each occurrence of old is replaced by new. [[Footnote: A character is, roughly, a single alphanumeric or symbolic character (one keystroke) inside single quotation marks. For more detail on what exactly constitutes a character, see the chapter on Java types.]] For example, "Tee hee!".replace('e', '*') produces "T** h**!"

charAt(pos) requires an index into the String and returns the character at that index. Recall that Strings are indexed starting at 0.

"Hello".charAt(2) is the same as "Hello".charAt(3)

indexOf(character) returns the lowest number that is an index of character in the String.

"Hello".indexOf('H') is 0 and
"Hello".indexOf('l') is 2. Also,
"Hello".indexOf('x') is -1, indicating that 'x' does not appear in "Hello".

lastIndexOf(character) returns the highest number that is an index of character in the String.

"Hello".lastIndexOf('H') is 0 and
"Hello".lastIndexOf('k') is -1, but
"Hello".lastIndexOf('l') is 3.

Strings are unusual among object in being able to be written out this way. Most objects can't be spelled out like this; they need to be explicitly created in a somewhat different fashion. [[Footnote: We will see how to do this in the chapter on Classes and Objects]] Strings are also unusual in a second way: once created, a String cannot change. If you use one of the String services described here that provides a String as a result, that result is a new String differing from the original. (The original remains, unchanged.)
Unlike String, Console is not built in to Java. It is a part of the cs101 libraries. This means that Console is only available to Java programs that use the cs101 libraries, a group of free code libraries developed for use with this textbook. Later in this book, we'll learn how to replicate the function of Console using only Things built in to Java, in case you don't always want to use the cs101 libraries.

There is another difference between Console and String: String is a kind of object -- a type -- while Console is actually a particular object. Console has a type, of course, but you will not be interacting with other similar objects. That is why the Console sidebar gives you information about the particular object, Console, but the String sidebar talks generically about all Strings.

Once you know the name of a particular object -- like Console -- and you know about the type of Thing that object is, you can ask it to perform tasks for you. For example, you can ask Console to print out "Hi there" on the screen. In Java, this is spelled `Console.println("Hi there");`. (Recall that the . and () are special syntax -- notation -- that indicate that we're asking the Thing called Console to do its println behavior (on the String "Hi there").) Specific information about the Things that you can do with Console is contained in the sidebar. In the next chapter, we'll explore in greater depth how you can find out what kinds of actions any particular kind of Thing can perform.

### 3.2 Doing Things With Things

A few objects, like certain Strings, can be typed directly on your keyboard. If you want to use a particular String in your program, one option is to type it in right there, as we have done in the sidebar. Other objects, like Console, are always available for use through their names. When you want to use a particular object, you will need to figure out how to identify it so that you can use it in your program. A name is one way to do this, but it is not the only way.

A name -- like Console -- gives you direct access to the object it names. For example, I might have a robot that I call Robbie; then I can ask Robbie to move using its name. I can also tell you about Robbie using this name. Below, you will see how to give names to Things, and -- using that technique -- I could even give Robbie the nickname Fred by giving the robot a second name. A name like console is a special object that knows how to communicate with the user in some very basic ways. If your program says

```java
Console.println("Hello there!");
```

Then the String "Hello there!" will appear on the Java console. (Remember, the double quotes aren't part of the string; they're just used to indicate where it starts and ends.) The statement

```java
Console.print("Hi");
```

is similar, except that `Console.print` doesn't end the line of output while `Console.println` does. This means that

```java
Console.print("A ");
Console.print("is for apple.");
```

would produce the output

```
A is for apple.
```

while

```java
Console.println("A ");
Console.println("is for apple.");
```

would produce

```
A is for apple.
```
Robbie or Console names the object, and is like a label that allows you to access the object directly.

Imagine that, in your program, mobyDick names a book. You might be able to refer to the author of Moby Dick by using a name designated for this purpose, like hermanMelville. You can also refer to an object without using a name for it. You can do this, for example, by using a related object to help you. So, for example, mobyDick.author() might refer to the author of that book. (As with Console.println("Hi there"), above, the . and () are special syntax. In this case, we are asking the Thing called mobyDick to tell us its author.)

In this way, we can access an object -- like hermanMelville -- indirectly -- through mobyDick. This is useful if we don't have otherwise have a name for it. (The same trick works in English, too: Omoo was written by Moby Dick's author.)

There are differences between direct and indirect reference. One is the length and complexity of the way we say it: HermanMelville vs. mobyDick.author(). Another has to do with whether the referent -- the object referred to by the name or the service request -- is expected to change. Names like hermanMelville (or Console) might always refer to the same Thing, as would mobyDick.author(). But some indirect references are expected to produce different Things each time the service request is made. Imagine that fortuneTeller is the name of an object that knows how to produce a random pithy saying (a fortune). The name fortuneTeller might even always refer to the same oracle. But you'd hope that a request like fortuneTeller.getFortune() would produce different pity sayings from one request to another, or the Thing named fortuneTeller wouldn't be very interesting!

3.3 Use Names to Keep Track of Things

With all of these Things floating around in our program, it is pretty easy to see that we'll need some ways to keep track of them. The simplest way to keep track of Things is to give them names. This is called assigning a value to a name. Giving something a name is sort-of like sticking a label on the Thing. We sometimes say that the name is bound to that value.

Unfortunately, not every Thing in our program comes with a name. Consider the fortune teller of the preceding section. We can ask the fortuneTeller to produce a fortune, but if we don't do anything with that fortune, we will have no way to refer back to it. It is as if, as soon as we get the fortune from fortuneTeller,
we drop it on the floor with all of the other Things that our program has created or used. Unless we somehow keep hold of what fortuneTeller gives us, we cannot get it back later.

We can solve this problem by putting a label on the first fortune when we get it. Then, if we want it back later, we can ask for it by name: the name on the label. In fact, that's just what names are for Java objects: labels that let us keep track of Things.

To actually assign a value to a name -- to create a binding between that name and that value, to stick the label on the Thing -- Java uses the syntax -- that is, the notation --

\[
\text{name} = \text{value}
\]

So we can remember the fortune our fortuneTeller provided by giving it a name:

\[
\text{myPersonalFortune} = \text{fortuneTeller.getFortune()}
\]

This associates the fortune produced with the name myPersonalFortune. Once a particular name refers to a particular Thing, we can use the name wherever we would use its value, with the same effect:

\[
\text{Console.print( "And your fortune is...." );}
\]

\[
\text{Console.println( myPersonalFortune );}
\]

The name becomes a stand-in for the Thing it refers to: the fortune told by the fortuneTeller. In the next chapter, we will see that a name is a simple kind of expression.

A name, like a label, can only be affixed to one Thing at a time. In other words, only one value may be associated with a name at any given time. One Thing can be referred to by any number of names at once (including, potentially, no names at all). The same person can be "the person holding my right hand", "my very best friend", and "Chris Smith". But only one person is "the person holding my right hand". [[Footnote: Barring weird interpersonal pileups, of course.]]

3.3.1 Declarations: Creating a Typed Name

But where do these names come from? Except for the rare name, like Console, that is available when your program is started -- and that is
probably already being used to keep track of something important -- you will need to come up with your own names and to tell Java about them, i.e., to generate the labels that you'll want to stick on Things. The way that you do this is with a declaration.

We already know that each Java Thing comes into the world with a type, i.e., an indication of what kind of Thing it is. In Java and other strongly typed languages, every name -- and every label -- also has a type. A Java name can only be used to label objects of the appropriate type. This type is associated with the name when the name is created. The type associated with a particular name never changes.

Since every Java name must have an associated type, the way that you create a name in Java is to tell Java what that type is. The term for telling a programming language what type a name has is declaration. Declaring a name means stating that that particular name is to be used for labeling values (Things, objects) of some particular type. A declaration creates a name -- a label -- suitable for sticking on objects of that particular type. The label can only be stuck on things of that type.

Names are declared using the type-of-thing name-of-thing rule:

```java
String bookTitle;
KlingonStarship enemyFighter;
```

The second word on each line is a name that is being declared. The first word on each line is the type that the name is being declared to have. In the first line of the example above, `bookTitle` is being declared to have type `String`. This is the Java type for a piece of text, like "The Forgotten Beasts of Eld". Finally, each declaration ends with a semicolon (;). So the first declaration here creates a name, `bookTitle`, suitable for naming pieces of text (or, in Java, `Strings`). The second line creates the name `enemyFighter` and says that it can refer to any Thing of type `KlingonStarship`.

In both of these cases, a new name -- a new label -- is created. That is, each of `bookTitle` and `enemyFighter` is now a label of the corresponding type. But neither of these statements says what the label is stuck on. In fact, it's quite possible that the label is stuck on nothing at all. Later in this chapter, we will see a special kind of name that doesn't create a label. But in Java, objects are named with label names.

A name has a certain lifetime, sometimes called its scope. Within that scope -- over its lifetime -- the name may be bound to many different values, though it can only be bound to one value at a time.

---

**Java Notes**

**Java Naming Syntax and Conventions**

Java identifiers can contain any alphanumeric characters as well as the symbols $ and _. The first character in a Java identifier
For example, enemyFighter may initially be the big ship right in front of you, but later change to be the high-speed ship just entering your field of vision. The association between a name and a type persists for the lifetime of the name, however. enemyFighter can only name a KlingonStarship, not a String or a BookID.

3.3.2 Definition = Declaration + Assignment

Declaring a name begins its useful lifetime, or scope. At that time, nothing else necessarily needs to happen -- and frequently, it doesn't. But sometimes it is useful to associate the name with a value -- to stick the label onto something -- at the same time that it is declared. This combination of a declaration and an assignment is called a definition.

- A declaration creates a name, telling you what type is associated with a name.
- An assignment sets the value of a name, telling you what value that name is bound to.
- A definition combines the "what kind of Thing it can name" and "what value it has" statement types.

For example:

```java
String bookTitle = "Weaving the Web";
String answer = Console.readLine();
Person melville = mobyDick.author();
Cat myPet = marigold;
```

cannot be a number. So _luckyDuck_ is a legitimate Java identifier, as _Alice_In_Wonderland_, but _24T_ is not.

Certain names in Java are reserved words. This means that they have special meanings and cannot be used as names -- i.e., to refer to Things, other than any built-in meaning they may have -- in Java. Reserved words are sometimes also called keywords. These are:

```java
abstract boolean break byte case catch char class const continue default do double else extends final finally float for goto if implements import instanceof int interface long native new package private protected public return short static super switch synchronized this throw throws transient try void volatile while
```

Java is case-sensitive. This means that `double` and `Double` are two different words in Java. However, you can insert an amount of white space -- spaces, tabs, line breaks, etc. -- between two separate pieces of Java -- or leave no space at all, provided that you don't run words together. You can't stick white space into the middle of a piece of Java -- a name or number, for example -- though.

Punctuation matters in Java. Pay careful attention to its use. Note, however, that white space -- spaces, tabs, line breaks, etc. -- is not matter in Java. Use white space to make your code more legible and easier to understand. You will discover that there are certain conventions to the use of white space -- such as lining up the name in a column, as we did above -- although these tend to vary from one programmer to the next.
myPet is bound to the actual Cat, not to the name marigold. But to understand this fully, we need to really think about what it means for a name to be a label.

### 3.3.3 Names are Labels

A Java object name really is just a label that can be stuck onto an (appropriately typed) object. When a label-type name is declared, a new label suitable for affixing on Things with that type is created. For example, a building name might be a cornerstone label, a person's name might go on a badge, and a dog's name might belong on a collar. You can't label a person with a cornerstone or pin a badge on a dog, at least not without raising an error. Unlike cornerstones or dog tags, though, labeling a Java object doesn't actually change that object. It just gives you a convenient way to identify (or grab hold of) the object.

In Java terms, if we declare

```java
RadioButton myButton;
```

this creates a label, myButton, that can be stuck onto Things of type RadioButton. Note that is not currently so stuck, though. At the moment, myButton is a label that isn't stuck to anything. Cornerstones and badges and dog tags don't come with buildings and people and dogs attached, either. Having a label is different from having something to label with it. **Labels don't (necessarily) come into the world attached to anything.** The value of a label not currently stuck onto anything is the special non-value null. That is, null doesn't point, or refer, to anything. So the declaration above is (in most cases) the same as defining

```java
RadioButton myButton = null;
```

Of course, we can attach a label to something, though we need to have that something first. We'll return to the question of where Things come from in a few chapters. For the moment, let's suppose that we have a particular object with type RadioButton, and we stick the myButton label onto it. (Now myButton's value is no longer null.)

After we give myButton a value -- stick it onto a particular RadioButton -- we can ask it whether it is currently pressed:
myButton.isSelected()

This request will return either true -- yes, I'm pressed -- or false -- no, I'm not. Let's imagine myButton currently is pressed, so it returns true. [[Footnote: This is an expression that returns a boolean value; see the discussion of expressions in chapter [Expressions].]]

Now, imagine that we declare

RadioButton yourButton = myButton;

The result of this declaration is that a new label is created. This new label is attached to the same object currently labeled by myButton. Assignments of label-type names do not create new (copies of) objects. In this case, we have two labels stuck onto exactly the same object, and we say that the names myButton and yourButton share a reference. This just like saying that "the morning star" and "the evening star" both refer to the same heavenly body.

Because myButton and yourButton are two names of the same object, we know that

myButton.isSelected()

and

yourButton isSelected()

will be the same: either the button that both names label is pressed, or it isn't. But we can separate the two labels -- say

myButton = someOtherButton

-- and now the values of

myButton.isSelected()

and

yourButton.isSelected()

might differ (unless, of course, someOtherButton referred to the same Thing as yourButton). Note that moving the myButton label to a new object doesn't have any effect on the yourButton label.

Note also that the labeled object is not in any way aware of the label. The actual radioButton doesn't know whether it has one label attached to it, or many, or none. A label provides access to the object it is labeling, but not the other way around.
3.4 A Tale of Things and Names

Let's walk through an example of how Things are named and how names refer to Things. This example involves a story that took place some years ago at a fancy party, the kind of party where one leaves one's hat at the door and whose attendees might include a few famous names...This story concerns one such party and three such personages: Charlie Chaplin, King George VI of England, and Eleanor Roosevelt.

First among our characters to arrive was Charlie Chaplin. He was wearing his usual bowler hat.

Hat charlieChaplinHat;

The code above just tells us that charlieChaplinHat is a label suitable for naming a Hat, but we will also assume that the the charlieChaplinHat label is, as usual, stuck on the bowler:

When Chaplin arrived in the lobby, he saw the hat check. He took off his hat and handed it to the hat check. (Fortunately, the hat check had a checkHat service, requiring a Hat:

hatCheck.checkHat( charlieChaplinHat );

Of course, now Charlie Chaplin wasn't wearing a hat:

charlieChaplinHat = null;

So unburdened, he walked in to the party.

Next to arrive was the King of England. He was wearing his crown.

Hat kingGeorgeHat;

Again, we assume that -- prior to the execution of the rest of this code -- kingGeorgeHat is labelling the crown.

When King George arrived at the hat check, he, too, removed his hat and gave it to the hat check.

hatCheck.checkHat( kingGeorgeHat );

kingGeorgeHat = null;

[[ Footnote:

Q. What would have happened if the King had executed these lines in the opposite order:

kingGeorgeHat = null;
hatCheck.checkHat( kingGeorgeHat );

Then he, too, went in to the party. Both men had a lovely evening at the party. Mr. Chaplin left first. Reentering the lobby, he approached the hat check and observed that it also provided a returnHat() service. So he executed

charlieChaplinHat = hatCheck.returnHat();

Much to his surprise, the hat handed to him by the hat check was not his simple black bowler. Instead, it was the magnificent crown of the King of England. Chaplin was still staggering around the lobby under the weight of this crown when King George emerged from the party and approached the hat check.

kingGeorgeHat = hatCheck.returnHat();

I'm sure it won't surprise you to hear that the King now found himself in possession of a simple black bowler. Being a man of simple good taste and few pretensions, George was half tempted to leave with that hat, but it occurred to him that there might be some other gentleman who would then regret the loss of such a sturdy topper. Further, he knew well from experience that the Crown of England can be a heavy burden and he hated the thought that someone else might have to suffer under it. Turning, he saw Charlie Chaplin staggering under just that burden.

"Sir," King George observed, "It seems our headgear has been exchanged by the hat check. Perhaps we should remedy this situation."

And so the two men proceeded to try just that. But there was a problem. Between them, the men had only two Hat labels, and each was occupied. For example, they considered executing

* kingGeorgeHat = charlieChaplinHat;

but this would have resulted in both men holding the crown, and Mr. Chaplin's trusty bowler lost in limbo with no way to retrieve it (since it would have been labelless).

Q. Explain why this would have been a bad idea.

The two men were puzzling over this dilemma when who should emerge from the party but Eleanor Roosevelt. (She'd snuck in while the reader wasn't looking.) This savvy diplomat immediately took measure of the situation

"I see that you two gentlemen have run into a bit of difficulty. Perhaps I can be of some assistance. You see, I am not wearing a hat, and so my head can be a temporary resting place while the two of you work your exchange.
"First, we will need a label for my hat:

   Hat eleanorRooseveltHat;

"You see, of course, that there's no Hat there: eleanorRooseveltHat is null. It's simply a label that could be stuck on a Hat; it's the potential Hat-holder representing my head."

[[Footnote: Actually, what Mrs. Roosevelt really said was, "As you can see, eleanorRooseveltHat == null is now true, but of course you won't be introduced to ==, the identity operator, until the next chapter." ]]

Next, the erstwhile Mrs. Roosevelt offered to take the Crown of England from Mr. Chaplin, to which he readily agreed:

   eleanorRooseveltHat = charlieChaplinHat;

Now, the tall thin diplomat and the short comic actor found themselves jointly holding England's crown. At this, Mrs. Roosevelt suggested that Mr. Chaplin release the crown by seizing instead the simple black bowler to which he was accustomed (and of which the King of England was growing overly fond).

   charlieChaplinHat = kingGeorgeHat;

Mrs. Roosevelt now had sole possession of the crown; Mr. Chaplin and King George both held the bowler. At this, King George released the bowler to take possession of his crown:

   kingGeorgeHat = eleanorRooseveltHat;

Finally, Mrs. Roosevelt released her hold on the crown, freeing her to return to important business:

   eleanorRooseveltHat = null;

(Of course, this step was not strictly necessary as each gentleman now held the proper hat. Mrs. Roosevelt simply wanted to leave open her options for wearing other hats later.)

And the three figures left the party satisfied that all was well.

3.5  Primitive Types, Literals, and Dial Names

Objects are extremely useful and every Java program that you use will make use of objects. But there are a few other kinds of programming language Things that do not have the same complex internal structure or behavior. Java, like many programming languages, has some built-in facilities for handling and manipulating simple kinds of information, like the number 42 or the character '§'. These Things are not objects and they cannot act on their own or

**Java Notes**

**Java Primitive Types**

**Numeric literals are...**

Each Java primitive type has its own built-in name. For example, `int` is a name for a type-of-thing corresponding to an integer value. There are actually four Java names for integers, depending on how much space the computer uses to store them. An `int` uses 32 bits or binary digits. It can represent a number between -2^31 to 2^31-1 which is big enough for most purposes. An integral number (i.e., a number without a decimal point) appearing literally in a Java program will be interpreted as an `int`.

If you need a larger range of numbers, you can use the Java type `long`, which can hold values between -2^63 and 2^63-1. You can just type in a value like 80951151051778, though. Literals intended to be interpreted as `long` end with the character `L` (or `l`):
be asked to provide services. They do have types and can be named, but their names do not behave exactly like object names. Because they are simple, these types are called **primitives**.

Java object types may have complex internal state or the ability to perform interesting behaviors. Java primitive objects do not have any internal state, nor can they do anything by themselves. They cannot ring like an alarm clock, close like a window, or be selected like a radio button. They cannot even add themselves or display themselves on a screen. Only objects can be asked to perform actions. In chapter 5, we will learn how we can use primitive Things to accomplish useful tasks. But, unlike object Things, primitive Things cannot accomplish anything by themselves.

### 3.5.1 Primitive Types

A type is Java’s way of indicating what kind of Thing something is and what it can do. Like objects, Java primitive Things have types. But unlike objects, you cannot create any new Java primitive types. Java has exactly eight primitive types. These types are built into the Java language, so they are always available to you. Four of these types correspond to integers. Two of the types correspond to decimal numbers. One of the types is for single characters. The eighth type is for true-or-false values, or booleans. These are all the types permitted in Java. The name of each of Java’s primitive types begins with a capital letter.

#### Character Literals Written Inside Single Quotes

Character literals written inside single quotes: `'x'`.

The Java character type is called **char**. Java characters are represented using an encoding called **unicode**, which is an extension of the **asci** encoding. Ascii encodes English alphanumeric characters as well as other characters used by American computers using 8 binary digits. Unicode is a 16-bit representation that allows encoding of most of the world’s alphabet. Character literals are enclosed in single quotation marks: `‘x’`.

Characters that cannot easily be typed can be specified using a **character escape**: a backslash followed by a special character or number indicating the desired character. For example, the horizontal tab character can be specified `\t`; newline is `\n`; the single quote character is `\'`, double quote is `""`, and backslash is `\` `. Characters can also be specified by using their unicode numeric equivalent prefixed with the `\u` escape.

The true-or-false type is called **boolean**. There are exactly two boolean literals, true and false.

### Floating Point Numbers

Real valued numbers are represented using **floating point** notation. There are two versions of real numbers, again corresponding to the amount of space that the computer uses to store them. One is **float**, short for floating point; the other is **double**, for double precision floating point. Both are only approximations to real numbers, and double is a better approximation than float. Neither is precise enough for certain scientific calculations. A float is 32 bits. The biggest float is about $3.4 \times 10^{38}$; the smallest is about $-3.4 \times 10^{38}$. The double type can represent numbers to an accuracy of about 8 significant decimal digits.

A double is 64 bits. The biggest double is about $1.8 \times 10^{308}$; the smallest is about $-1.8 \times 10^{308}$. The double type can represent numbers to an accuracy of about 16 significant decimal digits.

The double type gives more precise representation of numbers (as well as a larger range), and so is more appropriate for scientific calculations. However, since error are magnified when calculations are performed, computations with large numbers of calculations mean that unless you are careful, the imprecision inherent in these approximations will lead to large accumulated errors. [[Footnote: These issues are studied by the field of mathematics known as numerical analysis.]]

The default floating point literal is interpreted as a **double**: literal to be treated as a `double` must end with `d` or `D`. (A double literal optionally ends with `l` or `L`.)

Floating point numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., `9.87E-65` or `3.14`).

#### Character Escape: A Backslash Followed by a Special Character or Number

Characters that cannot easily be typed can be specified using a **character escape**: a backslash followed by a special character or number indicating the desired character. For example, the horizontal tab character can be specified `\t`; newline is `\n`; the single quote character is `\'`, double quote is `""`, and backslash is `\` `. Characters can also be specified by using their unicode numeric equivalent prefixed with the `\u` escape.

The true-or-false type is called **boolean**. There are exactly two boolean literals, true and false.

### Numbers

There are also two smaller integer types: the 16-bit **short** and the 8-bit **byte**. There are no **short** or **byte** literals. For most purposes, the int is probably the Java integral type of choice.

**WARNING**: The limited range of all the integral types means that calculations using such numbers can cause errors or give incorrect results if they would require going beyond the ranges of their types involved. The programmer must be aware of such limits.

Numbers to an accuracy of about 16 significant decimal digits.

A short is 16 bits. The biggest short is about $3.2 \times 10^{38}$; the smallest is about $-3.2 \times 10^{38}$. The short type can represent numbers to an accuracy of about 8 significant decimal digits.

A byte is 8 bits. The biggest byte is about $1.28 \times 10^{38}$; the smallest is about $-1.28 \times 10^{38}$. The byte type can represent numbers to an accuracy of about 7 significant decimal digits.

#### Numbers to an Accuracy of About 16 Significant Decimal Digits

Floating point numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., `9.87E-65` or `3.14`).

### Numbers to an Accuracy of About 8 Significant Decimal Digits

Floating point numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., `9.87E-65` or `3.14`).

### Numbers to an Accuracy of About 7 Significant Decimal Digits

Floating point numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., `9.87E-65` or `3.14`).
whose value is directly "understood" by the computer. In addition to integers, Java recognizes literals that approximate real numbers expressed in decimal notation as well as single textual characters.

All of the following are legitimate Things to say in Java:

- 6
- 42
- 3.5
- -3598.43101

Details of Java numeric literals -- and of all of the other literals discussed here -- are covered in the sidebar on Java Primitive Types. As we will see in chapter 5, you can perform all of the usual arithmetic operations with Java's numbers.[[ Footnote: Be warned, though, that non-integral values -- like 1/3 and 1.234567890123456789 -- are, in general, represented only approximately. ]]

Java can also manipulate letters and other characters. When you type them into Java, you have to surround each character with a pair of single quotation marks: 'a', 'x', or '%', for example. Note that this enables Java to tell the difference between 6 (the integer between 5 and 7) and '6' (the character 6, which on my keyboard is a lower case '^^'). The first is something that you can add or subtract. The second is not.

One character by itself is not often very useful, so Java can also manipulate sequences of characters called strings. Strings are used, for example, to communicate with the user. Error message, user input (i.e., what you type to a running Java program), titles and captions are all examples of Java strings. To describe a specific string in Java -- for example, the message that your computer prints to the screen when you boot it up -- you can write it out surrounded by double quotes: "Hi, how are you?" or "#^$%&*%^$" or even "2 + 2". Your computer doesn't understand the string, it just remembers it. (For example, the computer doesn't know of any particular relationship between the last example and the number 4 -- or the string "4".)

It turns out that it's also useful for many programs to be able to manipulate conditions, too, so Java has one last kind of primitive value. For example, if we are making sandwiches, it might be important to represent whether we've run out of bread. We can talk about what to do when the bread basket is empty:

if the bread basket is empty, buy some more bread....

Conditions like this -- bread-basket emptiness -- are either true or false. We call this kind of Thing a boolean value. Booleans are almost always used in conditional -- or test -- statements to determine flow of control, i.e., what should this piece of the program do next? Java recognizes true and false as boolean literals:
if you type one of them in an appropriate place in your program, Java will treat it as the corresponding truth value.

As we have just seen, primitive-type Things can be referenced by literals. In addition, Strings can be referenced by literals, by using the double-quotiation mark syntax: "What, me worry?" So all three of the following are literals -- "5" '5' and 5 -- but the first is an object-type Thing (a String), while the latter two are primitive-type Things (a char and an int, respectively).

There are a lot of rules about how these different Things work and how they are used. For details on Java's primitive types -- including their names and their properties, see the sidebar on Java Primitive Types.

### 3.5.2 Primitive Names are Different

If you want to refer to a Java object, you can do so using an appropriately typed name. You can also refer to a Java primitive Thing using a name. Declaration, assignment, and definition are used with primitive names in much the same way that they work with object names. For example:

```java
int myLuckyNumber = 6;
int yourGuess;
yourGuess = myLuckyNumber;
```

The first line creates a new name, `myLuckyNumber`, and binds it to 6. The second line creates a new name, `yourGuess`, but does not give it any initial value. [[Footnote: Whether yourGuess is given an initial value by this line depends at least on where the line appears within code and in some cases on the particular implementation of Java in which you are executing. It is always inadvisable to assume that yourGuess has been given a value, though -- being a dial -- it must, of course, be set to some value.]]

The final line assigns the value of `myLuckyNumber` -- 6 -- to the name `yourGuess`.

So far, names with primitive type look a lot like object names. But it turns out that there are some important, if subtle, differences. Java names with primitive types aren't exactly labels, as object names are. You see, there may be an unpredictably large number of Buttons or KlingonStarships, and so a label is the best way to keep track of any particular KlingonStarship that comes along. But Java primitives are different.

For example, there are only two boolean values possible in Java: true and false. This means that we can use a very different mechanism for a name that has type boolean. Java does just that. A name that can have only one or the other of two values can be represented using a switch: It can be on, or it can be off. This is just what Java does: **Boolean names are just switches.** Let's say we have a boolean name, `isSunny`. This name is just a switch. It is always set in one of the two positions: on/true, or off/false. So if the switch corresponding to `isSunny` is on, we'll know that `isSunny` is true. If the `isSunny` switch is off, we'll know that `isSunny` is false. We can tell the value corresponding to the boolean name just by inspecting the switch.
But if we use a switch to indicate true-or-false, how do we do assignment? When we use label names -- for objects -- we just stick a second label on the same object. When we use switches as names, creating a new name means creating a new switch. That is, **dial name assignment copies one dial's setting to another**. So what do we do when, for example, we have

```java
boolean amHappy = isSunny
```

(which means that I am happy if it's sunny, and not happy if it's not)? Simple enough: we set the new switch -- amHappy -- to the same position that isSunny is in. (Note: We do this once, at the time of the assignment. After that, the two switches are completely separate. More on this later....)

It turns out that this analogy works for all of the Java primitive types. For example, there are only a fixed, finite number of ints possible in Java. (See sidebar.) Although it might be confusing to imagine a switch with 4,294,967,296 positions, you can imagine that an int name is just a very large dial with those same 4 billion settings. By reading the setting of the dial, you can tell what value an int name corresponds to.

As remarkable as it may seem, each of the Java primitive types is represented in this way. The dial here is metaphorical, but the actual representation is very much as described. A Java primitive is stored in such a way that every name with a primitive type indicates its value just as the metaphoric dial does. A name corresponding to a byte is simply a dial with 256 positions. A long name has $2^{64}$ positions. And even float and double names have only finite numbers of positions; this is why floating point numbers don't *really* represent real numbers and in fact aren't all that good for extreme precision calculations. [[Footnote: Even a double precision floating point number can only represent $2^{64}$ values between $-1.8e^{308}$ and $1.8e^{308}$, so many values just can't be captured accurately. Note also that the actual precision of a double varies over this range.]]

![Figure 3.8. A boolean name is just a switch.](image)

![Figure 3.9. Copying values from one switch to another.](image)
over many more values than just a-Z, but there are still only $2^{16}$ possible characters in Java, so a char name is like a dial with that many positions.

For example,

```java
text 1;
```

associates i with a dial that's just the right size for a 32-bit integer.

### 3.5.3 Dials and Labels are Different

How is this different from a label? There are at least three big differences.

1. **Labels can be null; dials cannot.** When a label is created, it may not be attached to anything. A dial can't be created without having a value; by its very nature, *the dial's hand is on some value.* [[Footnote: In fact, the declaration of a dial-type name not only sets up an appropriately sized dial, it also sets that dial. This means that you must give a name a value before you can use it. Some special kinds of names get values by default. We will mention these values as the names are introduced. In fact, this isn't just true when it's created. A label can be unstuck -- null -- but a dial always has exactly one value.]]

2. **When a type X label is declared, no object of type X is created; when a type Y dial is declared, the dial is created and has a value.** When a label of a certain object-type is declared, a label name appropriate for that object-type is created (and storage allocated for it), but no object of that object-type is created. When a dial of a certain primitive-type is declared, a dial name appropriate for that primitive-type is created and also, by the very nature of a dial, its hand is set to *some* value of that primitive-type. [[Footnote: Note, however, that it may be illegal to access this "default" value; you should always be sure that a name has been given a value before you access it. You will learn more about the values initially given names as you learn about different kinds of names in Java.]]

3. **Labels can share a value; dials cannot.** Two labels can be stuck on to the same object; then, changing the object by using the first label to reference it changes the (same) object referenced by the second label. While two dials can have the same value (both are set to the same position on the dial), the dials are independent -- changing the setting on one dial never changes the setting on another dial.

In particular, assigning one label to another means that both are stuck on the same object (or both are null), so that they "share" a value. When one dial is assigned to another, the setting on the latter dial is copied onto the former. That's it; after that, the assignment is complete and the two dials go their separate ways. There is no further relationship between the values on the dials.

The last point is worth a closer look. Consider the following two very similar-looking sets of statements (one on the left, the other on the right):

```java
int i;
Cat marigold;
```
\begin{verbatim}
i = 3;           marigold = new Cat();
int j = i;       Cat phoebe = marigold;
i = 4             marigold.haveKittens();
\end{verbatim}

In both cases, the first statement declares a name and the second statement assigns the name a value. The
name \texttt{i} on the left is a dial-name and is set to 3; the name \texttt{marigold} on the right is a label-name and is set to a
new \texttt{Cat}.

In both cases, the third statement declares another name and assigns the first name to the second.
However, assignment has a different meaning in these two cases. On the left, the dial-name \texttt{j} is set to the same
setting (3) as the dial-name \texttt{i}. On the right, the label-name \texttt{phoebe} is stuck onto the same object that the label-
name \texttt{marigold} is stuck on.

In both cases, the fourth statement changes something by using the first name (\texttt{i} on the left and
\texttt{marigold}; on the right). However, the effects are quite different. On the left, the value of \texttt{i} changes to 4, of
course, but the value of \texttt{j} remains 3. If we asked whether \texttt{j} is 4, the answer would be "No." On the right, the
object referenced by \texttt{marigold} has kittens, so that (same) object referenced by \texttt{phoebe} has kittens. If we asked
whether \texttt{phoebe} has had kittens, the answer would be "Yes!"

The left side of the figure shows the \textit{dial}
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{dial_names_label_names_effect_assignment.png}
\caption{Dial names versus label names: the effect of assignment.}
\end{figure}

names \texttt{i} and \texttt{j}. The right side of the figure shows the
\textit{label} names \texttt{Marigold} and \texttt{Phoebe}. The middle row
shows the different effects of assignment in the two
cases.

To summarize:

- When you are using names that are declared to
  be of object-types, think of those names as
  labels that can be stuck on objects of the
  declared type.
- When you are using names that are declared to
  be of primitive-types, think of those names as
  dials that always are set to some physical point.
- This implies three key distinctions:
  - \textit{Labels can be null}; dials cannot.
  - \textit{Declaring a label does NOT create an object of
    the declared type}.
  - \textit{Labels can "share" a value}; dials cannot.
It is easy to recognize primitive-types: they are all lower-case letters, while (by convention) we always use an upper-case letter as the first letter of any object-type.

A more formal term for dial types is **value type**.

**Chapter Summary**

- In Java, most Things are objects. A object's type tells you what that object can do.
- Generally, it is an instance of a type, rather than the type itself, that provides behavior. You can make a request of an object using the . () syntax:

  
  ```
  object.service()
  ```

  asks *object* to perform *service*.

- *String* is the type for arbitrary text. *String* is the only Java object type with literals.
- *Console* is a cs101 library object that allows you to communicate with a user using Strings.
- A Thing can be accessed using a name or an indirect reference. If you do not hold onto a Thing -- using a name -- it may be dropped on the floor and you will only be able to access it through an indirect reference, if at all.
- Names can be used as placeholders for values. Every name is born (declared) with a particular type, and can only label Things having that type. A name can be associated with a value through assignment. Definition combines declaration with assignment.
- Object names are labels.
  
  - Declaring an object name does not create an object of the declared type. An object name can be null (not stuck on anything).
  - An object name can be null (not stuck on anything).
  - Assigning an object name simply sticks that label on the object. Object name assignments do not create new copies of objects.
  - Two object names can label (be stuck on) the same object, sharing it as a value. If you change the object, the change will be visible through either name as long as they share that value.
- Java has eight primitive types:
  
  - *char* is the type for single keystrokes (letters, numbers, etc.)
  - *int* is the standard type for integers. Other integer types include *byte*, *short*, and *long*.  
• double is the standard type for floating point numbers, which are approximations to real numbers. float is another floating-point type.
• boolean is a type with only two values, true and false.
• The limited range of all the numeric types means that calculations using such numbers can errors or incorrect results when they are used outside of their ranges. In addition, the limited precision of the floating-point types means that repeated calculations involved them can lead to large accumulated errors. The programmer must be aware of such differences between computer arithmetic and real arithmetic.
• The Java keyword for each of the eight of primitive types begins with a lower-case letter.

All other Java types are object types. By convention, we begin each object type with an upper-case letter.

• Primitive types have dial names. A dial name always has an associated value. Two dials cannot share a single value; each has its own copy.
• Literals are Things you can type directly to Java. Only the primitive types and the single object type string have literals.

Exercises

1. Assume that the following declarations apply:

```java
int i; char c; boolean b;
```

For each item below, give the type of the item.

a. 42

b. -7.343

c. i

d. 'c'

c. "An expression in double-quotes"
2. For each of the following definitions, fill in a type that would make the assignment legal.

\[
\begin{align*}
\text{f. } & b \\
\text{g. } & \text{false} \\
\text{h. } & "\text{false}" \\
\text{i. } & c \\
\text{j. } & 'b' \\
\text{k. } & "b"
\end{align*}
\]

3. This problem checks your understanding of assignment.

a. Assume that the following statements are executed, in order.

\[
\begin{align*}
\text{int } & a = 5; \\
\text{int } & b = 7; \\
\text{int } & c = 3; \\
\text{int } & d = 0; \\
a = & b; \\
c = & d;
\end{align*}
\]
a = d;

What is the value of a? of b? of c? of d?

b. Assume that the following statements are executed, in order.

```java
int a = 5;
int b = 7;
int c = 3;
int d = 0;
a = b;
b = c;
```

What is the value of a? Of b? Of c?

c. Assume that the following statements are executed, in order.

```java
char a = 'a';
char b = 'b';
char c = 'c';
char d = 'd';
a = b;
c = a;
a = d;
```

What is the value of a? Of b? Of c? Of d?

d. Assume that `myObject` is a name bound to an object (i.e., `myObject` is not null).

After the following statements are executed in order,

```java
Object a = myObject;
Object b = null;
Object c = a;
a = b;
```

Is the value of a null or non-null? What about b? What about c? What about myObject?
c. Assume again that myObject is a name bound to an object (i.e., myObject is not null). After the following statements are executed in order,

```java
Object d = myObject;
d = null;
```

is the value of d null or non-null? What about myObject?

d. Assume one more time that myObject is a name bound to an object (i.e., myObject is not null). After the following statements are executed in order,

```java
Object e = myObject;
myObject = null;
```

Now is the value of e null or non-null? What about myObject?

4. Which of the following could legitimately be used as a name in Java? (Note that none of them would be wise choices for names, except possibly in a Star Wars game, as none of them is likely to convey meaningful information to readers of a program.)

3PO
R2D2
c3po
luke
jabba_the_hut
PrincessLeia
Han Solo
obi-wan
foo
int
Double
character
string
goto
elseif
fi
5. Assume that the following declarations have been made:

```java
int i = 3;
int j;
char c = '?';
char d = '\n';
boolean b;
String s = "A literal";
String s2;
Object o;
```

Complete the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>dial or label?</th>
<th>Value (or null?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. In the section on names as labels, we put a label on the fortune teller's fortune:

```java
myPersonalFortune = fortuneTeller.getFortune()
```

Assume that this fortune is "You will live a long and happy life." Now assume that the
fortune teller is asked for another fortune:

```java
fortuneTeller.getFortune()
```

Note that this fortune -- assume that it is "Things change." -- is not assigned explicitly to
any label. At this point, we execute the fortune-printing statements from the same section:

```java
Console.print("And your fortune is....");
Console.println(myPersonalFortune);
```

What is printed?
7. Assume that there is an already-defined object type called `Date` and that `today` is an already-defined Date name with a value representing today's date. Suppose that you wanted to declare a new name, `yesterday`, and give it the value currently referred to by `today`. This would be useful, for example, if it were nearly midnight and we might soon want to update the value referred to by `today`.

Explain why the following attempt will not successfully solve this problem.

```java
Date yesterday;
yesterday = today;
```

[[Footnote: At this point of this book, you should understand why the above will not work, but we have not yet discussed what would work. We'll see the basic ideas for a successful solution in Chapter 7, Building New Things: Classes and Objects. Also relevant is the discussion about `clone` and Cloneable in Chapter 10, Inheritance.]]

8. Continuing the previous problem, now assume that `today` is an already-defined `int` (not Date) name with a value representing today's date, where 1 represents January 1, and 32 represents February 1, and 33 represents February 2, and so on, with 365 representing December 31. (Assume that this is not a leap year.) Again suppose that you wanted to declare a new name, `yesterday`, and give it the value currently referred to by `today`. Now, the problem is solvable with the tools from this chapter.

Give the solution (that is, declare `yesterday` appropriately and give it the value referred to by `today`). Then explain how this problem is different from the previous problem.

9. (From the footnote): In the tale of two hats, what would have happened if the King had executed his hat-checking lines in the opposite order?

```java
kingGeorgeHat = null;
hatCheck.checkHat( kingGeorgeHat );
```

10. Recall again the tale of two hats. Later, after the events of that story were long past, someone suggested that the problem here was that the hat check hadn't issued claim checks, which might have changed the circumstances. These claim checks would have been `ints`, not `Hats`.

"If the names had been dials rather than labels, would Mrs. Roosevelt's assistance still have been needed?"

As it happens, there were claim checks involved, but Charlie Chaplin and King George had clumsily dropped them, and they were unable to determine which claim check was whose. This is why they'd each used the `hatCheck.returnHat()` service, rather than a service that required a claim check as input.
Suppose that the situation were as follows:

After the mixup, Mr. Chaplin found himself holding claim check 2:

```java
int charlieChaplinCheck = 2;
```

while King George was in possession of claim check 1:

```java
int kingGeorgeCheck = 1;
```

However, the crown can only be retrieved with claim check 2, the bowler with check 1. So the two gentlemen are now faced with a swap of integers rather than hats. As the story originally unfolded, they gave up and later were forced to proceed with the hat swap. Imagine, instead, that Mrs. Roosevelt had walked up at this moment -- when the claim checks needed rearranging -- and write code to resolve the situation WITHOUT using the literals 1 and 2 further. (You may, of course, use charlieChaplinCheck and kingGeorgeCheck in your code.)
Chapter 4.
Specifying Behavior: Interfaces

Chapter Overview

- How do programs (and people) know what to expect?
- How do I describe a part or property of an entity to other community members?

This chapter introduces the idea of interfaces as partial program specifications. An interface lets community members know what they can expect of one another and what they can call on each other to do; in other words, interfaces specify "how they interact". In this way, an interface describes a contract between the provider of some behavior and its user. For example, the post office promises to deliver your letter to its intended recipient if you give it to them in the appropriate form. This promise (together with its requirements for a properly addressed and stamped envelope, etc.) constitutes a part of the post office's interface.

In this chapter, you will learn how to read and write Java interfaces. These allow you to use code designed by others -- in the same way that you can drop off an appropriately addressed letter at the post office -- and to tell others how to use the services that you provide. You will also learn about things that an interface doesn't tell you. For example, when you drop a letter off at the post office, you don't necessarily know whether it's going by truck or by train to its destination. You may not know when it is going to arrive. This chapter concludes with a discussion of what isn't specified by an interface and how good documentation can make some of these other assumptions explicit.

This chapter is supplemented by a reference chart on the syntax and semantics of Java interfaces. (See Java Charts/Interfaces)
Objectives of the Chapter

1. To learn how to recognize and read Java method signatures.

2. To understand how an interface specifies a contract between two entities while separating the user from the implementation.

3. To be able to read an interface and know what behavior can be expected of an object that implements it.

4.1 Interfaces are Contracts

Programs are communities of interacting entities. How does one entity know what kinds of services another entity provides? How do programmers know what kinds of behavior they can expect from objects and entities that they haven't built? The answer is in the interface that one entity provides to another. In other words, interface is a piece of the answer to the question of how things interact.

In the previous chapter, we saw objects in use. We said that a the object's type specifies what that object can do. In Java, an interface has a very special meaning: some types are interfaces. In this chapter, we will learn how a Java interface specifies the behavior of objects and see how to read an interface definition to understand what the objects of that interface type can do. We will begin by learning about what an interface is in a more general sense.

4.1.1 Generalized Interfaces and Java Interfaces

The dictionary defines interface as "the common region of contact between two independent systems." In Computer Science, we use interface to mean the boundary between two (or more) things. In general, when you are constructing a community of interacting entities, interface refers to the "face" that one of these entities shows another: what services it provides, what information it expects. One entity may, of course, have many interfaces, showing different "faces" to different community members.

For example, in software engineering, the term user interface refers to the part of a computer program that a person using that program actually interacts with. For example, a graphical user interface (GUI) is one that uses a certain interaction style, e.g., typically contains buttons and menus and windows and icons. [[Footnote: In fact, graphical user interfaces are sometimes called WIMP interfaces, for Window, Icon, Menu, Pointer.]]

Before GUIs, computer interfaces typically used text, one line at a time, the way that some chat programs work now.
A good interface meets the needs of its customers. For a user interface, this means taking into account the properties of the program and the rather different properties of human users. Some not-so-good user interfaces are not-so-good precisely because they overlook the fact that humans and computers have different skill sets. Like user interfaces, every interface should be designed bearing in mind the needs of the entities on both sides. We will learn more about graphical user interfaces in particular in Parts 3 and 4 of this book.

This more general Computer Science use of the word interface is one sense in which we will use the term in this book. In Java, there is a second, related but much more limited use of the word interface. A Java interface refers to a particular formal specification of objects' behavior: a Java type. The keyword interface is used to specify the formal declaration of a particular kind of contract guaranteeing the behavior of this type. (For example, there might be an interface defining clock-like behavior.) The Java language defines the rules for setting out that contract, including what can and can't be specified by it. A particular Java interface is a particular promise.

In this book, when we use the term "Java interface" or the code keyword interface, we are referring to this formal declaration. When we use the terms "generalized interface" or "user interface", we are referring to more general computer science notions of interfaces. A Java interface is one way to (partially) specify a generalized interface. There may also be things that are part of the general promise -- such as how long a particular request might take to answer or the actual appearance of a component on the computer screen -- that can't be specified in a Java interface.

This chapter deals specifically with Java interfaces. The ideas of generalized interfaces permeate all parts of this book; the generalized notion of an interface is central to interactive program design. We will explicitly revisit this issue -- generalized interface design -- in the chapters on Protocols and Communication in Part 4 of the book.

4.1.2 Interfaces Encode Agreements

Why do we want interfaces? What work do they do for us? Interfaces -- in both their general and theirs Java-specific senses -- are very useful to us. They allow us to use things without knowing precisely how they work. And they allow us to build things without knowing precisely how they will be used. Let us now look at a particular every-day kind of interface to see how this works.

An excellent example of a standardized interface is an electrical outlet. In the United States, there is a particular standard for the shape, size, and electrical properties of wall outlets. This means that you can take almost any US appliance and plug it in to almost any US wall outlet and rest assured that your appliance will run. The power company doesn't need to know what you're plugging in -- there are no special toaster outlets, distinct from food processor outlets, for example -- and you don't need to know whether the power company produced this electricity through a hydroelectric plant or a wind farm. The outlet provides a standard interface, with a particular contract, and as long as you live within the parameters of that contract, the two sides of the interface can remain relatively independent.
This is the power of an interface: An interface is a contract that one object or entity makes with another. Interfaces represent agreements between the implementor (or builder) of an object and its users. In many ways, these are like legal contracts: they specify some required behavior, but not necessarily how that behavior will be carried out. They also leave open what other things the parties to the contract may be doing. As a result, an interface separates what the user needs to know from what the implementor needs to know.

In some cases, there may be multiple different interfaces that provide similar services. For example, US appliances don't generally work in European outlets. There are several standard electrical outlet interfaces throughout the world. It isn't clear that one of them is particularly better than another, but it is unquestionably true that you can't use one side of the US outlet interface (e.g., a US appliance) with the other side of the European interface (a 220V outlet). In the same way, software will only work if the user and the implementor are relying on the same interface. If you want to mix and match disparate electrical interfaces, you will need a special adapter component. The same is true for software.

There are also, even in the US, certain appliances that can't use standard wall outlets because they don't meet the conditions of the interface contract. For example, an electric oven draws too much current, and so needs a special kind of wall outlet. The physical connector -- the plug -- is different on this appliance, to indicate that it fits a different interface. You can't plug an electric oven in to a standard US wall outlet. This is because its needs don't meet the (sometimes implicit) constraints of standard (15 or 25 amp) US circuits. Sometimes this happens in software, too -- you need a different interface because the standard one doesn't provide precisely the functionality that you need.

4.1.3 A Java Interface Example

Consider, for example, a counter such as appears on the bottom of many web pages, recording the number of visitors. Most such counting objects have a very simple interface. If you have a counting object, you expect to be able to increment it -- add one to the number that the counting object keeps track of -- and to be able to read -- or get -- its current value. This is true pretty much no matter how the counting object actually works or what other behavior it might provide. In fact, by this description, a stopwatch might be a special kind of counting object that automatically increments itself. So we might say that increment and getValue form a useful interface contract specifying what a (minimal sort of a) counting object might be. In Java, we write this as:

```java
interface Counting {
    void increment(); // describes the increment contract
    int getValue();  // describes the get value contract
}
```

Figure 4.1. A counter like this one appears on the bottom of many web pages. It could also be used as an automobile odometer.
By the end of this chapter, you will know how to read this interface declaration.

Once you and I agree on an interface for a counting object, I can build such an object -- and you can use it -- without your needing to know all of the details of how I built it. You can rely on the fact that you will be able to ask my counting object for its current value using \texttt{getValue()}. Your code, which uses my counting object, doesn't need to know whether increment adds one point (for a soccer goal) or six (for a touchdown in American football). It doesn't need to know whether I represent the current value internally in decimal or binary or number of touchdowns, field goals, etc.

If I one day exchange my original counting object for a more sophisticated one that can be reset before each game (or each time I rewrite my web page), your code should continue to work. This is because your code depends only on being able to increment and read the value of my counting object -- the properties specified by the \texttt{Counting} interface -- and my new object still satisfies that contract. Similarly, I can go off and build a counting object using whichever internal representations I wish to provide, so long as I meet the contract's commitments (\texttt{increment()} and \texttt{getValue()}).

Of course, you may want to know more about my counting object than what the \texttt{increment/ getValue} interface tells you. Some of this information may be contained in the documentation for \texttt{Counting}. (This counting object's value will always be non-negative.) Other information may be contained in the documentation for my particular implementation. (My \texttt{BasicCounter} counting object implementation is guaranteed to increase; its value cannot decrease.) If you want to know whether my object provides additional services, though, you may need to use an interface that specifies this additional behavior (e.g., a \texttt{Resetable} interface). We will explore the kinds of information conveyed by an interface, and that which should be included in interface documentation, towards the end of this chapter.

### 4.2 Method Signatures

In the previous chapter, we saw how to ask an object to perform a service using the .() notation. For example, to find the author of the book \textit{Moby Dick}, we could ask it:

\begin{verbatim}
mobyDick.getAuthor()
\end{verbatim}

This asks the object named \texttt{mobyDick} to perform its \texttt{getAuthor} behavior. In an object-oriented programming language such as Java, the formal name for a behavior (or service) provided by an object is a \textit{method}: a method is a thing that an object knows how to do. In an interface, we focus on the specifications for these methods (or services) and not on the instructions for how to achieve them. That is, an interface is a collection of service specifications. Any object that implements that interface must satisfy those specifications, though there are virtually no limits on how it might do that.

The formal name for this kind of service specification is a \textit{method signature}. For example, the \texttt{Counting} interface specifies two services -- \texttt{increment} and \texttt{getValue} -- that every counting object must provide. The body of the interface declaration is these two method signatures, or service specifications. A
method signature describes what things that method expects (or needs to know about) and what the method will return. It also needs a name, so that you can refer to and invoke the method (of course). In the chapter on Exceptions, we will see that there is one other kind of thing that can be a part of a method specification.

A method signature specifies a particular behavior that an object provides. It does not say anything about how the object will perform that behavior. A method signature does not, for example, contain a recipe that an object could follow to produce that behavior. As we saw in the first chapter of this book, a program needs to have a recipe for every piece of behavior that it produces, and in the next few chapters we will see how such recipes can be constructed. But producing behavior is not the job of an interface; an interface only specifies what behavior needs to be produced.

The goal of an interface is to provide an agreement between the provider of some behavior and its users. If it is done right, it allows both the provider and the user to work independently. In Java, an interface is a collection of method signatures, each of which describes a particular behavior that is part of an object's contract. Each method signature consists of three parts:

1. the name, or what the method is called;
2. the parameter specifications, or what things the method needs to do its job; and
3. the return type, or what the method will provide when it is done. [[Footnote: There is actually one other part of some method signatures, the throws clause. Every method signature must have a name, parameter list, and return type, but some methods do not have a throws clause. The throws clause will be introduced in the chapter on Exceptions. In addition, certain modifiers -- such as abstract, explained below -- may be included in a method signature.]]

The next three subsections describe each of these three pieces of a method signature. For each part, we will look especially at what that part tells the user of the interface and at what it requires of someone providing the behavior behind an interface. Once we have understood how a method signature works and is used, we will return to see how method signatures can be put together to form a complete interface.

4.2.1 Name

The purpose of a method name is to make it possible to talk about the method. The name is how you ask an object to perform that behavior using the . () syntax. The method name should, of course, make it easy for both behavior providers and behavior users to figure out what the method is supposed to do.

When you are using an interface, the name of the method is whatever name the interface says it is. You can determine this by reading the interface specification. Hopefully, the name was chosen well so that it is easy to remember and to figure out what that method does. The interface should, of course, also have documentation to help you understand how it works.

If you are building an interface, you can give a particular method any name that you want. It is a good idea to give it a name that will help you (and the users of your code) remember what it does. Recall that Java
names are allowed to include alphanumeric and a few symbolic characters. The syntax of Java names sidebar in Chapter 3 lists the precise rules for legal Java names. By convention, the name of a method should start with a lower case letter.

4.2.2 Parameters and Parameter Types

Parameters are the things that a method needs in order to work. For example, to check a book back into a library, the CirculationDesk's checkIn method will need the BookID of that book. When you ask an object to perform one of its methods, you need to provide these things between the parentheses of the `()` syntax:

```java
circDesk.checkIn( mobyDickID )
```

The information that you supply to the method -- in this case, mobyDickID, the BookID corresponding to Moby Dick -- is called an argument. The user of the method needs to have, and to provide the method with, all of the necessary arguments. Once the user hands the arguments to the method, the method will need a way to keep track of them. The way that the method does this is with a special kind of name called a parameter. A parameter is a temporary name associated with an argument supplied to a method. [Footnote: In chapter Classes and Objects, we will see how a parameter can be used by the provider of the method behavior to refer to the arguments supplied when the method is invoked.] Because a method signature specifies what the method does, it needs to describe what kinds of arguments the method requires.

For example, the method signature for the CirculationDesk's checkIn method looks like this:

```java
boolean checkIn( BookID whichBook )
```

Just as arguments are supplied to a method request between parentheses, method signatures indicate what arguments are required in the same place. This specification of required arguments is called the method signature's parameter list.

When you are designing an interface, you will need to specify a type and a name for each parameter. (The type-of-thing name-of-thing rule (from the Chapter on Things, Types, and Names) strikes again.) The type can be any legal Java type (including both primitive and object types); the name can be any Java-legal name that you choose to give the parameter. It is advisable that you give your parameters names that make it easy for the users and implementors of your method to figure out what role the particular parameter plays in the method. Our convention is to use names that begin with a lower-case letter for parameters.

The list of parameters is separated by commas: type-of-thing name-of-thing, type-of-thing name-of-thing, and so on until the last type-of-thing name-of-thing which doesn't have a comma after it. The whole list is enclosed in parentheses. You can list your parameters in any order. Of course, some orders will naturally make more sense.
than others, and although the choice is arbitrary, once chosen the order is fixed. This means that users and implementors of the method will need to follow the order declared in the interface.

The `getValue` and `increment` methods of `Counting` don't have any parameters, i.e., they don't need any information to begin operation. Their parameter lists are empty: `()` as in `getValue()` and `increment()`. `CirculationDesk`'s `checkIn` method has one parameter, a `BookID`. We can call that `BookID` anything we want to; the name given a parameter in a method signature turns out not to matter at all. Of course, calling it something like `whichBook` makes it easier for the eventual user of the method signature to figure out what information to supply.  

A more complex `AlarmedCounting` interface might be mostly like our `Counting` interface but in addition have a `setAlarm` method that takes two parameters, one an `int` indicating the value at which the alarm should go off and the other a `String` that should be printed out when the alarm is supposed to be sounded.

```java
setAlarm( int whatValue, String alarmMessage )
```

When you are using a method, you need to pass the method a set of arguments that match the order and types of the parameter list. That is, between the parentheses after the name of the method you're invoking, you need to have an expression whose type matches the type of the first parameter, followed by a comma, followed by an expression whose type matches the type of the second parameter, and so on, until you run out of parameters: `increment()`, `transform( "a string to transform" )`, or `setAlarm( 1000, "capacity exceeded" )` You can tell what arguments you need to provide to a method by reading the parameter list in its signature.

### 4.2.3 Return Type

A parameter list specifies the information that a method needs in order to do its work. A method signature also needs to specify what -- if anything -- its users can expect to get back. In many cases, a method request to an object returns a value. The `Counting` method called `getValue` is, not surprisingly, an example of such a method. So is a `CardCatalog`'s `lookup` method -- which returns a `BookID` -- or `Console`'s `readln` method. Returning this value is part of the method's specification, so the method signature needs to include this information.

Specifically, the signature of a method indicates the method's **return type**: the type of the value returned. So, for example, `getValue`'s signature indicates that it returns an `int`, while `lookup`'s signature indicates that it returns a `BookID`:

```java
int getValue();
BookID lookup( String whatToLookUp );
```

In some cases, the method does not return a value. (`increment` is an example of such a method: it changes the value stored inside the counting object, but doesn't give anything back to the entity that invoked
The return type of such a method is a special Java keyword: `void`. The only purpose for void is as the return type of methods that don't return a value. The `Counting` interface's `increment` method doesn't return anything, so its return type is `void`.

When you use a method, you may or may not want to do something with the value returned. The return type of the method signature tells you what type of thing you can expect to get back, e.g., so that you can declare an appropriate name to store the result:

```java
int counterValue = myCounting.getValue()
```

where `myCounting` is something that implements the `Counting` interface, i.e., satisfies the `Counting` contract (and therefore has an `int`-returning `getValue` method). After this statement, `counterValue` is a name that refers to whatever `int` `myCounting`'s `getValue` method returned.

### 4.2.4 Putting It All Together: Abstract Method Declaration Syntax

Now you know about all of the components of a method signature. All you need to know is how to put them together. The `type-of-thing name-of-thing` rule comes into play here as well. The type of a method is its return type, so a method specification is:

```java
returnType methodName ( paramType1 paramName1, ... paramTypeN paramNameN ) ;
```

For example,

```java
int getValue();
```

or

```java
void increment();
```

or

```java
void setAlarm( int whatValue, String alarmMessage );
```

Note that these declarations end with a semi-colon (`;`). This means that the method signature is being used here as a specification -- a contract. It doesn't say anything about how the method -- say `increment` -- ought to work. That is, it doesn't even have a space for how to perform this method, just the method specification.

This form -- method signature followed by a semi-colon -- is called an **abstract method**. There is even a Java keyword -- `abstract` -- to describe such methods. It is OK, if sometimes redundant, to say

```java
abstract void increment();
```

instead of the form given above. Method signatures are used as abstract methods in creating interfaces. But method signatures are also used in other parts of a Java program as part of actually creating behavior. We will see how to use method signatures in that way in the chapter on Classes and Objects.
Since interfaces always specify only method signatures, interface method declarations are always abstract. If you don't say so explicitly, Java will still act like the word abstract is there. However, if your method definition does not end with a semi-colon, your Java interface will not compile.

4.2.5 What a Signature Doesn't Say

The properties of a method that are documented by its signature are its name, its parameters, and its return type. \[\text{Footnote: In addition, method signatures may include visibility and other modifiers and any exceptions that the method may throw.}] \] That leaves a whole lot open.

For example, for each parameter:

- What is that parameter intended to represent?
- What relationships, if any, are expected to exist among the parameters?
- Are there any restrictions on the legal values for a particular parameter?
- Will the object represented by a particular parameter be modified during the execution of the method?

For the return type:

- What is the relationship of the returned object to the parameters (or to anything else)?
- What may you do with the object returned? What may you not do?

Other questions not included in the method signature:

- What preconditions must be satisfied before you invoke this method?
- What expectations should you have after the method returns?
- How long can the method be expected to take?
- What other timing properties might be important?
- What else can or cannot happen while this method is executing?

Not all of these questions are relevant to every method. For example, the precise amount of time taken by the counting object's getValue method is probably not important; it is important that it return reasonably quickly, so that the value returned will reflect the state at the time that the request was made. However, it is important to recognize that these and other questions are not answered by your method signatures alone, so you must be careful to document your assumptions using Java comments.
4.3 Interface Declaration

Now that we know all about Java method signatures, it is very easy to declare a Java interface. A Java interface is simply a collection of method signatures.

4.3.1 Syntax

A Java interface is typically declared in its very own file. The file and the interfaces generally have the same name, except that the file name ends with .java. (For example, the Counting interface would be declared in a file called Counting.java.)

Like most other declarations, an interface follows the type-of-thing name-of-thing rule. The type-of-thing is, in this case, interface. The name is whatever name you're giving the interface, if you're declaring it:

interface Counting

Now comes an interface body: an open-brace followed by a set of method signatures followed by a close-brace. Note that it doesn't matter in which order the two methods are declared; the two possible orders are equivalent. The whole thing (including the interface Counting part) looks like this:

interface Counting
{
    abstract void increment();
    abstract int getValue();
}

That's all there is to it.

Q. In this definition of Counting, the word abstract appears twice. In the previous definition, above, it doesn't appear at all. Explain.

In fact, that was so easy, let's try another interface. This one is Resetable, and it is a very simple interface. (Good interfaces often are.) Resetable has a single method:
This interface is fine, but it could do with a little bit of documentation. After all, there are many things that an interface doesn’t specify.

Q. Can you identify some things that should be included in `Resetable`’s documentation?

For the precise specification of what may be included in an interface definition, in what order, and under what circumstances, see the Java Chart on Interfaces.

### 4.3.2 Method Footprints and Overloading

It might seem that each method in an interface would have a unique name. However, it turns out that this isn't the case -- at least, not exactly. Instead of a unique name, each method in an interface (or class) definition must have a unique footprint. The method’s footprint consists of its name plus its ordered list of parameter types. Only the ordered list of parameter types counts; the return type of the method, and the names given to the parameters, are not relevant to its footprint.

For example, a `reset()` method with no parameters (an empty parameter list, `()`) has a different footprint from a `reset( int newValue )` method (with the parameter list `(int)`), and both are different from `reset( String resetMessage )` (parameter list `(String)`). Only the parameter type matters, though, not the parameter names: `reset( String resetMessage )` is the same as `reset( String whatToSay )`.

As long as two methods have different footprints, they can share the same name. This is very common and even has its own name: overloading. Overloading allows an object to have two (or more) similar methods that do slightly different things. For example, there are two very similar mathematical rounding methods. One has the signature

```java
int round( float f );
```

while the other has the signature

```java
long round( double d );
```

Java’s `Math` object has both of these methods. If you ask `Math` to `round` a `float`, it will give you an `int`. If you ask `Math` to `round` a `double`, it will give you a `long`. This is very convenient: in both cases, a floating point number is converted to an integer, but in either case the more appropriate size is used. Parameter type pairing like this -- `floats` with `ints`, `doubles` with `longs` -- is one reason for method overloading.

Another kind of method overloading involves optional parameters. For example, our `AlarmedCounting` interface has a
void setAlarm( int whatValue, String alarmMessage )

method. It might also be useful to give it a second method that allows you to specify the alarm message, without changing the value at which it is set:

void setAlarm( String alarmMessage )

If both of these method signatures are part of the AlarmedCounting specification, the request

yourAlarm.setAlarm( 1000, "Capacity reached" )

sets the alarm message to trigger at 1000, printing the message "Capacity reached", while

yourAlarm.setAlarm( "Oops, all full" )

simply changes the warning to be issued when the AlarmedCounting reaches capacity.

Overloading method names is the choice of the interface builder. The interface user simply makes use of the interface as it is given.

4.3.3 Interfaces are Types: Behavior Promises

Now that we have these interfaces, what good do they do? Interfaces are kinds of Things: they are Java types.

In Java, every interface name is automatically a type name. That is, when you are declaring a (label) name, you can declare it suitable for labeling things that implement a specific interface. In the chapter on Classes and Objects, we will see how to declare Java classes and how to indicate what interface(s) the class implements.

So, for example, the declared type of myCounting, above, was Counting:

    Counting myCounting;

In this example, myCounting is declared to be of type Counting, i.e., something that satisfies the Counting contract (interface) that we declared in the preceding sections. For example, we might have an interface called Game that includes a getScoreCounter() method that returns a Counting:

    interface Game
    {
        abstract Counting getScoreCounter();
        // maybe some other method signatures....
    }

If theWorldCupFinal is a Game, then we might say

    Counting myCounting = theWorldCupFinal.getScoreCounter();

In this case, we don't know anything more about the type of myCounting; we just know that it is a Counting.

Often, as users of other people's code, interfaces are the only types we need to know about.
4.3.4 Interfaces are Not Implementations

We have seen that an interface can be used as the type of an object. You can use names associated with that type to label the object. You can pass objects satisfying that interface to methods whose parameter types are that interface type, and you can return objects satisfying that interface from a method whose return type is that interface. The Counting in the previous paragraph was an example of the power of interfaces.

However, there are certain things that you cannot do with an interface.

Of course, when we're manipulating that Counting object, we don't know anything about how it works inside. We don't know, for example, whether it has a touchdown part and a field goal part, or is represented in decimal or in binary, or is likely to keep going up while we're thinking about it (since players might keep scoring). To figure this out, we'd need to know more than just the interface -- the contract -- that it satisfies; we'd need to know how it is implemented.

Interfaces are about contracts, promises. They don't, for example, tell you how to create objects that satisfy those promises. In the next several chapters, we'll learn about building implementations that satisfy these promises and about creating brand new objects that meet these specifications. To do that will require additional machinery beyond the contract/promise of an interface.

Chapter Summary

- An interface is a contract that a particular kind of object promises to keep.
- Java interfaces are Java types.
- Every (public) interface must be declared in a Java file with the same name as the interface.
Java interfaces contain method signatures.

A method signature is also called an **abstract** method.

A method signature specifies a method's name, parameter types, and return type. It does not say anything about how the method actually works.

Arguments are things supplied to a method; parameters are what the method calls them.

One interface may have multiple methods with the same name, as long as they have different ordered lists of parameter types. Method name plus ordered parameter type list is called the method's footprint. Having two methods with the same name but different footprints is called overloading the method name.

An interface does not contain enough information to create a new object, though it can be used as a type for an existing object (that implements the interface's promises).

Many important properties of a method specification or interface are not specified by the method or interface declaration. Good documentation describes these additional assumptions.

### Exercises

1. StringTransformer has a transform method. Declare an interface, Transformer, that contains this single method specification, so that StringTransformer might be an implementation of this interface.

2. A Clock is an object that needs a method to read the time (say, `getTime`) and one to set the time (say `setTime`). Assuming that you have a type Time already, write the interface for a Clock.

3. Extend the interface of Clock (from the previous exercise) to include a setAlarm method (that should specify the Time at which the alarm should go off).

4. Extend the Clock interface further so that there is a second setAlarm method that takes a Time and a boolean specifying whether the alarm should be turned on.

5. Write the interface AlarmedCounting.

6. Consider the following interface:
interface Game
{
    /* returns the Counting that keeps track of the team's score */
    abstract Counting getScoreCounter(Team team);

    /* returns the Counting that keeps track of how many fouls */
    /* each player has committed */
    abstract Counting getFoulCounter(Team team, int playerNumber);

    /* returns the counting that keeps track of how much time */
    /* has passed in the period so far */
    abstract AlarmedCounting getTimeCounter();

    /* returns the length of a period */
    abstract int getPeriodLength();
}

Assume that theWorldCup is a particular Game, according to this interface.

a. Write a type declaration for the name theWorldCup. Don’t worry about where its value comes from.

b. Write a type declaration suitable for holding the result of theWorldCup.getTimeCounter().

c. Write an expression that returns the object that counts the fouls of player 5 on Team manchesterUnited.

d. Write an expression that returns the current score of Team juventus in theWorldCup

e. Write a method invocation that sets up theWorldCup (and its internal representation) so that it will print "Period over!" when the elapsed time reaches the length of the period.

7. Write the interfaces for CirculationDesk and CardCatalog (from Chapter 2). Can you write the interface for BookShelf?

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT
AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Chapter 5.
Expressions: Doing Things With Things

Chapter Overview

- How do I use the Things I have to get new (or other) Things?

This chapter and the next introduce the mechanics of executable code, the building blocks for individual sequences of instruction-following. Chapter 3's Things each come with a type, which specifies how that Thing can interact. **An expression is a piece of code that can be evaluated to yield a Thing with a value and a type.**

Simple expressions include literals -- Things that Java literally understands as you write them -- and names, which stand in for the Things that they refer to. More complex expressions are formed by combining other Things according to their types, or promised interactions.

To understand a complex expression, you must understand its parts (a basic form of "what goes inside") and how they are combined (a basic "how they interact"). Sometimes, you have to understand this without knowing all of the details of what's inside.

This chapter introduces several new kinds of expressions. Instance creation expressions, field access, and type membership expressions all involve objects. Most operations involve primitive types. And some other expressions -- assignment, equality testing, and explicit casting -- can be used with either objects or primitive Things. For each of these kinds of expressions, this chapter explains its syntax as well as the type and value of the resulting Thing.

Sidebars in this chapter cover details of various Java operators, including casts and coercion rules. A separate supplementary reference chart -- **Java Charts/Expressions** -- summarizes the syntax and semantics of Java expressions.
Objectives of the Chapter

1. To understand that an expression is a piece of Java code with a type and a value.
2. To become familiar with the rules of evaluation for basic Java expressions.
3. To learn how to understand complex expressions as combinations of simpler expressions.
4. To be able to evaluate both simple and complex expressions, giving the resulting type and value.

5.1 Evaluating an Expression Produces a Thing

In the preceding chapters, we have learned that Things have types as well as values. We have seen how a type tells you what an instance of that type can do. We have also seen three different ways to get our hands on a Thing:

- A literal, like 6, is a Thing that Java understands directly. Rules for interpreting literals (from the sidebar on Java Primitive Types) specify the literal's type.
- A name, like theClock, refers to the Thing to which the name is bound. A name is declared to have a specific type.
- A request, like theClock.getTime(), returns the Thing that is the (return) value provided by theClock's getTime() service. A method signature specifies the type of Thing that a service returns.

Each of these ways of obtaining a Thing is an example of an expression, a piece of code that can be evaluated to produce a value (or Thing) of a particular type. To evaluate an expression simply means to calculate the value of the expression. In a computer program, every value is produced by an expression. This chapter investigates many of the different kinds of expressions found in Java.

An expression is the simplest piece of Java code. An instruction-follower—an execution of Java code—evaluates the expression to obtain its value, which will always be of the expression's type. There are many kinds of expressions, and each has its own rules of evaluation that determine what it means for an instruction-follower to evaluate that expression. Each kind of expression is distinguished by the way that its value and type are produced, and these -- together with the syntax of the expression -- are the important features of expressions that you will need to learn.

In addition to the three kinds of expressions mentioned above—literals, names, and service requests—there are expressions that do arithmetic, assign values to names, and create or compare objects. Legitimate Java
expressions include 2 + 2, "Hi, there", and this.out.writeOutput( this.in.readInput() ). The last of these is an expression whose evaluation involves inter-object (and inter-entity) communication.

In this chapter, we will examine many different kinds of expressions and learn what values and types they produce. We will begin with the kinds of expressions that we have already seen.

### 5.1.1 A Literal Is What It Looks Like

Expressions are ways to get your hands on Things. A simple way to get your hand on a Thing is to spell out the Thing that you want. So the very simplest Java expression is a literal: an expression whose value is interpreted literally, such as 25 or 32e-65 or "How about that?". Java literals include the various kinds of numbers, characters, Strings, and booleans. For a more complete enumeration of literal expressions and rules regarding their syntax (i.e., how you write them), see the sidebar on Java Primitive Types in chapter 3.

are the ints, doubles, chars, booleans, or Strings that they appear to be.

Every expression has a value and a type, obtained by evaluating the expression. The value of a literal is its prima facie value, i.e., what it appears to be. The type of an expression is the type of its value. Integer literals are always of type int unless an explicit long type suffix (l or L) is included in the literal. Non-integral numeric literals are always of type double unless explicitly specified to be of type float (using the f or F suffix).

Of course, not all Java Things can be spelled out explicitly. In fact, except for String objects, only Things with primitive types can be literals.

### 5.1.2 A Name Has An Associated Value (and Type)

Another simple way to get your hand on a Thing is to have it around in the first place. Names -- labels for objects, dials for primitives -- are also Java expressions. A name is only a legitimate expression within its legitimate lifetime, i.e., within its scope. For present purposes, it suffices to think of a scope as beginning when the name is declared. [[ Footnote: Strictly speaking, the scope of a variable -- a name with no special properties beyond being a name -- begins at its declaration and extends to the end of the enclosing block. (See the section on blocks in the chapter on Statements.) Later, we will see three other kinds of names: classes, fields and parameters. Class names have scope throughout a program or package; they may be used anywhere. Field names have scope anywhere in their enclosing class, including textually prior to their declaration. Parameter names have scope throughout their method bodies only. ]] has its declared type and most recently assigned value.

The value of a name is the value currently associated with it, i.e., labeled by it, if it is a label name, or recorded on the dial, if it is a dial name. The type of a name expression is always the type associated with that name at the time of its definition. [[ Footnote: Note that the type of a name expression is the declared type of the name rather than the type of the value... ]]

associated with the name. That is, even where there is disagreement between the declared type of a name and its value, the type of a name expression is always its declared type. ]]

For example, if we are within the scope of a declaration that says

```java
int myFavoriteNumber = 4;
```

and nothing has occurred to change the value associated with (recorded on the dial called) `myFavoriteNumber`, then the value of the expression

```java
myFavoriteNumber
```

is 4 and its type is int. That is, the int 4 is the result of evaluating `myFavoriteNumber`.

### 5.1.3 Method Invocation Asks an Object to Do Something

Literals and names are simple Java expressions. In each case, a single Thing is involved. We have also seen expressions in which an object is asked to do something or to provide us with a value. In the previous chapter, we learned that the way that an object performs the service is called a method. This kind of expression -- asking an object to provide a service -- is formally called **method invocation**. Method invocation is the primary way in which one object asks another to do something. It is the primary basis for inter-entity communication and interaction, because it is the main way in which objects talk to one another.

In Java, a method invocation involves:

- An expression whose value is the object to whom the request is directed, followed by
- A period (or "dot"), followed by
- The name of the method to be invoked, followed by
- Parentheses, within which any information needed by the method request must be supplied.

An example method invocation might be

```java
"a test string".toUpperCase()
```

This example consists of a String literal expression -- "a test string" -- and a request to that object to perform its `toUpperCase()` method. A String's `toUpperCase()` method doesn't require any additional information, so the parentheses are empty.

(They can't be omitted, though!) This invocation matches the method signature of the String `toUpperCase()` method as described in the Selected String Behavior sidebar of Chapter 3.

The value of a String's `toUpperCase()` method is a new String that resembles the old one, but contains no lower case letters. Its type is the return type declared in the `toUpperCase()` method signature: `String` So
the value and type of this expression are the same as the value and type of the literal expression "A TEST STRING".

Another example of method invocation is

```java
Console.println( "Hello" )
```

This asks the object named by the name expression Console to print the line supplied to it. It requires that a String -- the line to be printed -- be supplied inside the parentheses. This is "necessary information" is called an **argument** to the method. It corresponds to the parameter specified by Console's `println` method signature.

What is the value of this method invocation expression? `Console.println( "Hello" )` is a method invocation whose primary use, like that of assignment, is for its side effect, not its value. We use this method to make something appear on the user's screen. Good style dictates that we wouldn't use this expression inside any other expression. It turns out that many methods have no real return values, so (as we saw in the previous chapter) there's a special Java type for use on just such occasions. This type is called **void**. It is only used for method return types, and it means that the method doesn't return anything.

Method invocation differs from literals and names in that method invocation expressions can **contain other** expressions. In both of these examples -- "a test string".toUpperCase() and `Console.println( "Hello" )` -- expressions such as "a test string", "Hello", and even `Console` had to be evaluated before the method invocation expression itself could be evaluated. This makes method invocation a **compound expression**; one that can contain other expressions within itself.

The complete evaluation rule for a method invocation expression is as follows:

1. Evaluate the object expression to determine whose method is to be invoked.
2. Evaluate any argument subexpressions.
3. Evaluate the method invocation by asking the object to perform the method using the information provided as arguments.
4. The value of the expression is the value returned by the method invocation. The type of the method invocation expression is the declared return type of the method invoked.

In order for step 3 to work, the object must know how to perform the method, i.e., it must have instructions that can be followed in order to produce the return value needed in step 4. We have already seen how an interface can describe an object's commitment to provide such behavior. We will see in the next chapters how this commitment may be accomplished in detail.

From the perspective of the method invoker, however, the transition from step 3 to step 4 happens by magic (or by the good graces of the object whose method is invoked). The object offers the service of providing a particular method requiring certain arguments and returning a value of a particular type. For example, if we look at the documentation (or code) for String, we will see that it has a `toUpperCase()` method.
that requires no arguments and returns something of type String. The println method of Console requires a String as an argument, and println's return type is void. We will learn more about the methods that objects provide in the chapters on Classes and Objects and Designing with Objects.

5.2 Combining Expressions

Since expressions are Things -- with types and values -- expressions can be combined to build more complicated expressions. For example, the expression "serendipitous".toUpperCase() has the type String and the same value as the literal "SERENDIPITOUS". That is, you can use it anywhere that you could use the expression "SERENDIPITOUS". So, for example, you could get an adverbial form of this adjective by using "serendipitous".toUpperCase() + "LY", producing the same Thing as "SERENDIPITOUSLY" (see the sidebar on String Concatenation), or extract the word "REND" using "serendipitous".toUpperCase().substring(2,6) (see the Selected String Behaviors sidebar in Chapter 3).

In general, since every expression has a type, you can use the expression wherever a value of that type would be appropriate. The exception to this rule about reuse of expressions is that some expressions are constant -- their value is fixed -- while other expressions are not. A few specific Java contexts require a constant expression. In these cases, you cannot use a non-constant expression of the same type. (For example, "to" + "get" + "her" is a constant expression, but str + "ether" is (in general) not, even if str happens to have the value "tog". [[Footnote: The expression str + "ether" would be constant if str were declared final, though. Names declared to be final cannot be assigned new values.]] There are a few places where Java requires a constant value. These will be noted when they arise.

The evaluation rule for a compound expression is essentially the same as the evaluation rules for the expressions that make it up: Evaluate the subexpressions that make up this expression, then combine the values of these subexpressions according to the evaluation rule for this expression. For example, when we evaluate "serendipitous".toUpperCase(), we are actually evaluating the simpler (literal) expression "serendipitous", then evaluating the method invocation expression involving "serendipitous"'s toUpperCase() method. Similarly, str + "ether" evaluates the (name) expression str and the (literal) expression "ether", and then combine these values using the rules for + expressions, detailed below. In this case, str and "ether" are subexpressions of str + "ether". There are two additional details: 1) Evaluating the subexpressions may itself involve several evaluations, depending on how complex these expressions are and 2) it may not always be clear which operation should be performed first.
Method invocation, like other expressions, can be used to form increasingly complex expressions. For example, we can combine two method invocations we used above to cause the value of "A TEST STRING" to appear on the user's screen:

```java
Console.println( "a test string".toUpperCase() )
```

In this case, the value of the `toUpperCase()` invocation is used as an argument to `println`. We can also cascade other kinds of expressions, such as

```
"This is " + "a test string".toUpperCase()
```

or

```java
Console.readLine().toUpperCase()
```

### 5.3 Assignments and Side-Effecting Expressions

Literals, names, and most method invocations are used to produce Things: values with types. However, some expressions are evaluated not for the Thing that result but for some other effect that happens when the expression is evaluated. This thing-that-happens is called a **side effect**. We have actually seen examples of this already: an assignment is a prototypical side-effecting expression.

In previous chapters, we have seen some simple assignments including some that were mixed with declarations and buried inside definitions. In its simplest form, an assignment involves a name and an expression whose value is to be assigned to that name:

```java
aString = "Something to say...."
```

** rent is used for its side effect, not its type and value.**

The point of the assignment expression is to associate the name -- `aString`, in this case -- with the value of the Thing after the `=`. It is this side effect -- the binding of a name to a value -- that is generally why one uses an assignment statement.

In its most general form, an assignment expression is

```java
name = someExpression
```

The right-hand side of the assignment expression -- what comes after the `=` -- can be any expression. The **left-hand side** of the assignment expression -- what comes before the `=` **must** be a name or another expression that can refer to a label or a dial. In this context, and in this context **only**, the name expression refers to the label

---

**Style Notes**

**Don't Embed Side-Effecting Expressions**

When you use a side-effecting expression, it is best if this expression is not a subexpression of any other expression. So, for example, while assignments--as expressions--can be used inside other expressions, it is generally considered bad style to do so. Embedding side-effecting expressions inside other expressions can make the logic of your code very difficult to follow. Side effects are also important and often difficult to catch. By highlighting the side-effecting expression by making it the outermost expression, you are increasing the likelihood that it will be read and understood.
or dial itself, not to the particular value currently associated with the name. Of course, the type of the name on the left-hand side must be compatible with the type resulting from the evaluation of the expression on the right-hand side.

[[ Footnote: Odd as it may seem, left-hand side -- or its shorthand, lvalue -- is actually a technical computer science term. It refers to the name in an assignment expression, which is treated as a location -- a label or dial -- rather than as a true sub-expression of the assignment. The term lvalue is used even when the programming language in which the assignment is written does not position the name on the left-hand side! ]]

Although assignment is generally used for its side effect, it is an expression. This means that evaluation of an assignment must produce a Thing -- a value with a type. The type of an assignment is the type of its left-hand side. The value of an assignment expression is the value assigned to the left-hand side, i.e., the value produced by evaluating its right-hand side.

For example, assuming that aString has previously been declared to be a String, the type of the expression

\[
aString = "Something to say...."
\]

is string because that is the type associated with the name aString. The value of the expression is "Something to say...." because that is the value assigned to aString.

An assignment is a side-effecting expression: although its evaluation produces a type and a value, it has another effect that is often -- if not always -- more important. Some method invocations -- service requests -- are similarly side-effecting. For example,

```java
Console.println( "^*&$%" )
```

is a side-effecting request that causes something to be printed on the user's computer screen. Side-effecting methods often -- though not always -- have return type void -- no return value. This helps to make it clear that such a method is to be used for its side effect, since it does not provide a useful return value. All void methods should be side-effecting -- otherwise they do nothing at all! -- but not all side-effecting methods are necessarily void methods.

Assignment statements and void methods are among the most common expressions used for their side effects. We will see several other expressions with important side effects in the remainder of this chapter. In the next chapter, we will explore the use to which such side-effecting expressions can be put.

## 5.4 Other Expressions that Use Objects

Some kinds of expressions work only with objects. We have already seen how to request that an object perform a service through method invocation, perhaps the most common object expression. In this section, we cover three additional expressions that use objects: field access, instance creation, and type membership. Each of these kinds of expressions will be discussed further when we explore how objects are actually created.
beginning in the chapter on Classes and Objects. In this chapter, it is sufficient to recognize these kinds of expressions and to understand their associated types and values.

### 5.4.1 Fields

We have seen that an object may provide services in the form of methods. Some objects also provide data in the form of a name. For example, the object called Math provides a double (dial) called PI, which has (is set to) a value that is a little more than 3.14159.

Like a method, a field is accessed using the dot syntax, but without following parentheses. So, for example,

```java
Math.PI
```

is the double dial belonging to the object called Math, with a value approximating a real number whose most significant digits are 3.14159.

A field access expression is essentially a name expression, though a more complex one than the simple names described above. The value of a field access expression is -- as for a simple name -- the value associated with the label or dial. The type of a field access expression is -- also like a simple name -- the field's declared type.

We can use field invocations in compound expressions, too. If `myWindow` is a Window with a `getSize()` method that returns a Dimension,

```java
myWindow.getSize().height
```

first asks `myWindow` to perform its `getSize()` method, resulting in a particular Dimension object, then asks the Dimension object for its height field. This compound expression is the same as first creating a name for the Dimension and assigning it the result of the method invocation:

```java
Dimension mySize = myWindow.getSize();
```

and then asking the newly named Dimension object, called `mySize`, for its height field.

Because field access expressions are actually name expressions, they also have special behavior on the left-hand side of an assignment statement. That is, you can assign to a field access expression just as you would to a simple name, and the field access expression behaves like the label or dial to which it refers. For example, if height is an int dial owned by `mySize`, the expression

```java
mySize.height = mySize.height / 2
```

halves the value contained in the height dial of `mySize`, which might shrink `mySize` vertically by half.
5.4.2 Instance Creation

In the previous chapter, we looked at some object types called interfaces. An interface specifies an object's contract, but not its implementation. In chapter 7, we will learn about another object type, called a class. A class is an object type that has an associated implementation. A particular object -- an instance -- is often manipulated using an interface in order to separate the object's user from its implementation details. But when you want to create a new instance, you need to have an implementation as well as a contract. This means that you need a class.

Instance creation is a kind of expression that uses a class to create a new instance of that class. The details of this expression type are covered in the chapter on Classes and Objects; for now it is enough to recognize it.

An instance creation expression has three parts: the keyword new, the class (type) name, and a (possibly empty) list of arguments, enclosed in parentheses. This description of how to write an expression is called its syntax, and we can abbreviate it as:

\[
\text{new ClassName ( argumentList )}
\]

For example,

\[
\text{new File ( "myData" )}
\]

creates a new File object with external (outside of Java) name myData. Like all other expressions, this one has a type -- ClassName, the kind of object created, in this case File -- and a value -- the new object created. The instance creation expression is typically used inside an assignment or method invocation.

The rules of evaluation for instance creation expressions are similar to the rules of evaluation for method invocation. The return value is always a new instance of the type (or class) whose instance creation expression is invoked (in this case, File). The return type is always the type whose instance creation is invoked. Instance creation is a side effecting expression (since it creates a new object).

5.4.3 Type Membership

There is one last operator that is usable only with objects. This is an operator called instanceof, which checks whether an object has (or can have) a certain type. For example, is the Thing in my pocket a String?

When would an object of one type be an instance of another? There are at least three different ways that this could happen, all of which are covered in more detail elsewhere in this book. First, one type may be a specialization of another. For example, a collie is a kind of a dog, so an object can simultaneously be a collie and a dog. Second, one type-contract (interface) can have multiple implementations (classes), like a telephone that may use wires or cellular transmission or satellite to send its signal. A cell phone, land line, and satellite
telephone might each implement the Telephone interface but each have its own specific type (e.g. CellularTelephone). Finally, one (instance) object may implement many interface contracts: A Person may be a Parent, an Employee, and a SoccerGoalie. A daughter, an employer, and a soccerForward would each access this object through a different interface. In each of these cases, it is reasonable to ask, "Is this Dog (or Telephone or Person) a Collie (or CellularTelephone or SoccerGoalie)?"

A type membership expression looks like this:

\[ \text{anObjectExpression} \ \text{instanceof} \ \text{ObjectTypeName} \]

For example,

\[ \text{myDog} \ \text{instanceof} \ \text{Collie} \]
\[ \text{aTelephone} \ \text{instanceof} \ \text{CellularTelephone} \]
\[ \text{mobyDick.author}() \ \text{instanceof} \ \text{SoccerGoalie} \]

The first operand, which precedes the keyword `instanceof`, can be any expression whose value is of any object (non-primitive) type. The second operand, which follows the keyword `instanceof`, must be the name of an object type. As we shall see in the next few chapters, this name may be the name of any class or any interface.

The `instanceof` operator is used to determine whether it is appropriate to treat its first operand according to the rules of the type named by its second operand. The value of an `instanceof` expression is a boolean, true if it is appropriate to treat the object according to this type, false otherwise. So, for example,

\[ \"a String\" \ \text{instanceof} \ \text{String} \]

has the value true (because "a String" is a (literal) instance of the type String), while

\[ \text{new Object()} \ \text{instanceof} \ \text{String} \]

has the value false (because the new Object created by the instance creation expression `new Object()` is not a String.

### 5.5 Expressions Involving Objects or Primitives

Method invocation and field access, instance creation and type membership are all expressions that involve objects. These kinds of expressions cannot be used with primitive Things, although -- except for instance creation -- their resulting values may have primitive types. For example, a method must be invoked on an object but may require an argument that is a `char` or return an `int`. A field belongs to an object, but may have type `double`. And `instanceof` expressions always return boolean values.
In contrast, literals -- except for string literals -- are always primitives. In the next section (5.6), we will look at other kinds of expressions that operate only on primitives and not on objects. These include various arithmetic and logical operations as well as numeric and logical comparators. All of these operations take advantage of the known structure of the primitive types to provide basic calculations.

There are a few kinds of expressions that operate equally on either primitives or object types. These expressions are the focus of this section. We have already seen assignment; in this section we will also introduce explicit cast expressions and equality testing.

Assignment expressions involve a name on the left-hand side and an expression of any type suitable for binding to that name on the right-hand side. This means that if the name on the left-hand side is of an object type -- a label name -- the right-hand side must be an expression with an (appropriate) object type; if the name on the left-hand side is a primitive -- dial -- name, the expression on the right-hand side must have a (corresponding) primitive type. The use of an assignment expression is identical whether it is operating over object or primitive types, although the mechanics of assigning to a label name differ from the mechanics of assigning to a dial name. (Two label names may be stuck on one object, while dial assignment involves copying the value of one dial onto another. These distinctions are described in detail in Chapter 3: Things, Types and Names.)

There is also one kind of expression that is primarily used with primitive Things, but that has one object application. This is +, whose use as the arithmetic addition operator is described below. The sidebar on String concatenation, above, explains how + can be used to glue two Things together. In this case, it is important to distinguish the use of + as addition from + as String concatenation. They are really two separate operations that simply happen to be spelled in the same way.

[[ Footnote: There is a name for this phenomenon, the use of one operation for multiple purposes. It is called operator overloading. This is very much like method overloading -- described briefly in Chapter 4: Interfaces and in somewhat more detail in later chapters. -- except that methods always belong to objects, not primitive Things. In some programming languages, the programmer can overload operators, giving them additional behavior. Java does not allow the programmer to overload operators, but a few operators -- like + -- have multiple built-in meanings. ]]

5.5.1 Casting and Coercion: Changing the Type of a Thing

Sometimes Things don't have the types we might wish. For example, we might have the int 3 when we really want the double 3.0, or a Dog who is really a Collie that can herd sheep. Coercion is the process of viewing a Thing of one type as though it had a different type: for example, treating an int (like 3) as though it were a double (3.0) or a Dog as

Java Notes

Coection and Casting in Java

Generally, coercion happens automatically whenever it is information-preserving. Java only makes certain automatic -- implicit -- coercions. For example, Java knows how to make byte into short, short into int, int into long, long into float, and float into double. This works because each type spans at least the magnitude range of the ones appearing before it in the list. (A few of these coercions -- such as long to float -- may lose precision.) These coercions -- which are, in general, information-preserving -- are called widening. We will see in the chapter on Inheritance that there are also widening coercions on reference types.

Coercions in the opposite direction are called narrowing. Java does not generally perform narrowing coercions automatically.
though it were a Collie. Coercion does not change the Thing; it merely provides a different view.

5.5.1.1 Widening and Narrowing

Some coercions, such as treating an int or a float as a double or a Collie as a generic Dog, are information-preserving. That is, every int (and every float) has an accurate representation as a double. These are called widening coercions, and Java can do them automatically when types demand it. For example, the (stylistically ugly) definition

\[\text{double } d = 3;\]

implicitly coerces the int 3 into the double 3.0 before making the assignment to d. Similarly, if you've made the appropriate definitions, Java can figure out that it's OK to treat a CellTelephone as a Telephone or a Collie as a Dog.

Other coercions, such as treating the double 3.0 as though it were an int or a Dog as a Collie, are not necessarily information-preserving. (The double 3.001 might also be treated as the int 3, so this type transformation in general loses information.) You may know that a particular Dog is really a Collie, but Java won't automatically treat it as such since some Dog might not be a Collie. These coercions, called narrowing, are not performed by Java automatically. Instead, you need to indicate a narrowing conversion to Java by using an explicit cast expression as described below.

Coercion is a way of looking at a Thing; it does not actually change the Thing itself. A coercion simply provides a different version (with a different type). For dial types, this version is essentially a copy. For label types, it is another "view" of the same object.

Java's specific treatment of coercion is described more fully in the sidebar on Coercion and Casting in Java.

5.5.1.2 Explicit Cast Expressions

Sometimes, you need to change the type of Thing even when Java will not do so automatically. This is accomplished by means of an explicit cast expression. Like automatic coercion, a cast expression gives you a view of the Thing cast as a different type.

For example, if you have a Dog and know that it is actually a Collie, you can tell Java to treat it as a Collie. It would be an error to ask

\[\text{myDog.herdSheep();}\]
sheep herding is not an ability of most Dogs. But if you know that myDog is a Collie, you can tell Java to treat it as a Collie and then ask it to herd sheep. The syntax of this request -- which we will examine below -- is

\[(\text{Collie} > \text{myDog}).\text{herdSheep}()\]

or, more verbosely,

\[
\begin{align*}
\text{Collie } & \ a\text{Collie;} \\
\text{aCollie} & = (\text{Collie} > \text{myDog}; \\
\text{aCollie}.\text{herdSheep}() \\
\end{align*}
\]

[[ Footnote: The more succinct -- and preferred -- form of the Collie cast actually begins with two parentheses. One -- the pair around Collie -- is the cast expression. The other -- surrounding (Collie) myDog -- is used for grouping as described in the final section of this chapter. ]]

When you use an explicit cast expression, you must be sure that the thing you're casting really can be coerced to its new type: for example, that the Dog you are casting really is a Collie. The explicit use of a cast expression tells Java that you really do mean to do something that Java can't tell is OK, like make a lossy coercion. It is your responsibility to be sure that the request you are making really is OK. An `instanceof` expression can be used to verify type membership before a cast expression.

The syntax of a cast expression is

\[
(\text{type-name}) \ \text{expression to be cast}
\]

That is, you put the name of the type that you wish the Thing to have in parentheses before the (expression representing the) Thing.

For example, if `myInt` is an `int`-sized dial displaying the value 3 -- perhaps from

\[
\text{int } \text{myInt} = 3;
\]

-- then

\[
(\text{long}) \ \text{myInt}
\]

is a `long`-sized dial displaying 3 and

\[
(\text{double}) \ \text{myInt}
\]

is an expression with the same type and value as the literal expression 3.0. Throughout this, `myInt` itself remains an `int`-sized dial displaying the value 3. Casting, like implicit coercion, does not actually modify the castee.

If explicit cast:

\[
\text{pe } \ old\text{Thing}
\]

\[
old\text{Thing}; \ \text{type is NewType}
\]

Evaluating a cast expression yields the value of the cast operand (in this case, `myInt`), but with the type of the explicit cast (in this case, `long`). A cast expression does not alter its operand in any way; it simply yields a new view of an existing value with a different type. Some casts are straightforward and
appropriate; some risk losing information; and most are simply not allowed. For example, in Java you cannot cast an int to boolean.

Explicit cast expressions are also allowed from one Java object type to another under certain circumstances. Specifically, a Java object may be cast to a type if the object is an instanceof that type.

For further details on explicit cast expressions, see the sidebar on Coercion and Casting in Java.

5.5.2 Equality Testing and Identity

The final primitive- and object-Thing operation is equality testing. There are actually two equality operators that can be used with either primitive or object Things: ==, which tests whether two Things are the same, and !=, which tests whether two Things are different.

**Beware:** == tests for equality; = is the assignment operator.

```java
expression1 == expression2
```

is an expression with boolean type; its value is true if `expression2` are the same and false otherwise.

```java
expression1 != expression2
```

also has boolean type, but its value is false if `expression2` are the same and true otherwise.

---

**Java Notes**

**Java Operators**

Java operators include:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left-shift sign-extended</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right-shift zero-extended</td>
</tr>
<tr>
<td>+=</td>
<td>Assignment addition</td>
</tr>
<tr>
<td>-=</td>
<td>Assignment subtraction</td>
</tr>
<tr>
<td>*=</td>
<td>Assignment multiplication</td>
</tr>
<tr>
<td>/=</td>
<td>Assignment division</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Assignment bitwise and</td>
</tr>
<tr>
<td>^=</td>
<td>Assignment bitwise negation</td>
</tr>
<tr>
<td>%=</td>
<td>Assignment modulus</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td>! &amp;</td>
<td>Logical NOT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>Pre-increment</td>
</tr>
<tr>
<td>--</td>
<td>Pre-decrement</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
</tr>
<tr>
<td>? :</td>
<td>Ternary operator</td>
</tr>
</tbody>
</table>

The arithmetic operators and bit-wise logical operators in the first row are, respectively, addition, subtraction, multiplication, division, bitwise or, bitwise and, bitwise negation, modulus, left-shift sign-extended right-shift, and zero-extended right-shift. The + operator is also used for String concatenation when at least one of its arguments is a String. The - operator can also be used as unary (one-argument) negation.

The operators in the second row are operator-assignment operators that combine their correlate in the first row with an assignment operation. Thus `x + = 2` has the same effect as `x = x + 2`; the difference is that the left-hand side of the combined operator is evaluated only once. The value of an operator assignment expression is the new value of the left-hand side. All assignment expressions modify the name that is their left-hand side.

The third row above lists the six comparison operators, each of which returns a boolean. The final comparison is not-equal.

The fourth row lists the logical operators: logical negation, logical conjunction (and), and logical disjunction (or). Each of these takes boolean arguments -- one in the case of negation, two in the case of conjunction and disjunction -- and returns a boolean.

The operators in the fifth row are autoincrement and autodecrement. These can be used as either prefix or postfix operators. Both `++x` and `x++` modify `x`, leaving it incremented. However, `++x` returns the incremented value of `x`, while `x++` return the unincremented value. The `- -` operator works similarly.

The final two operators are simple assignment (which work like the compound assignments, above) and the ternary (three-operand) expression conditional:

```java
x > y ? a : b
```
evaluates to `a` if `x > y` and to `b` otherwise. (Any boolean-valued expression may be used instead of `x > y` and any expression may be used in place of `a` or `b`.)
* `ion1 == expression2` Type is boolean; true iff `expression2` are the same. `expression1 != expression2` Type is boolean; true iff `expression2` are different.

* For primitive types, Things are the same whenever they "look" the same, i.e., when their (types are compatible and) values are indistinguishable. [[Footnote: Java actually performs any automatic coercions available to compare primitive objects, so `(int) 3 == (long) 3` is true.]]

* For object types, values are the same exactly when the two expressions refer to the same object.

This last point is actually rather subtle. Two objects may look different but actually be the same, or they may appear similar but actually be different. Consider, for example, identical twins `x` and `y`. Although they may look exactly the same, they are still two different people. If one gets a haircut, the other's hair doesn't automatically get shorter. If one takes a bath, the other doesn't get clean. Thus they are different: `x == y` is false.

For Java equality testing, it is not sufficient for two objects to look alike, as in the case of identical twins; they must actually be the same object, so that modifications to one will necessarily be reflected in the other. On the other hand, two object Things can be the same even if they happen to be viewed through different types: my `CellTelephone` and your `Telephone` might actually be the identical object.

Equality of primitive Things is different. The `int 3` has no internal structure that can be changed (the way that one twin's hair can be cut). If you change `3`, you don't have `3` any more. If dial name `a` and dial name `b` are each of type `int` and each has `3` as its value, then `a == b` is true.

5.6 Complex Expressions on Primitive Types: Operations

Perhaps the most common kind of expression on primitive types is made up of two expressions combined with an `operator`. Java operators are described in the sidebar on Java Operators. They include most of the common arithmetic operators as well as facilities for comparisons, logical operations, and other useful functions. Assignment and equality testing, described above, are also Java operators. Of special note are `+` for String concatenation and unary `-` for negation.

Each operation takes arguments of specified types and produces a result with a particular value and type. For example, if `x` and `y` are both of type `int`, so is `x + y`. The `+` `operator` can be used to combine any two numeric types. The two Things combined with the operator are called the `operands`. In the expression `x + y`, `+` is the operator and `x` and `y` are the operands. Some operators take two operands. These are called `binary` operations. Other operators take only one operand; these are the `unary`
operations. One operator -- \( ? : \)-- takes three operands. The operands of this operator can be any expressions, not just primitives or objects.

### 5.6.1 Arithmetic Operation Expressions

The operator \(+\) is an example of a kind of operator called an **arithmetic operator**. The rules for evaluation of the binary arithmetic operators \(+\), \(-\), \(*\), \(/\), and \(\%\) are simple: compute the appropriate mathematical function (addition, subtraction, multiplication, division, and modulus, respectively), preserving the types of the operands. As explained in the sidebar on **Arithmetic Expressions**, an expression of the form

\[
\text{operator type has type type} \quad \text{type operator type}
\]

has type \(\text{type} \) for all of the basic arithmetic operations on most of the primitive types. That is, for these arithmetic operators, if the types of the two operands are the same, the result -- the value of the complete expression -- will generally also be of that type. For example, the expressions

\[
3 + 7 \\
2.0 \times 5.6 \\
5 \div 2
\]

evaluate to the \text{int} 10, the \text{double} 11.2, and -- perhaps surprisingly -- the \text{int} 2, not 2.5 (or 2.0), respectively.

Sometimes, an operator needs to treat one of its operands as though it were of a different type. For example, if you try to add \(7.4\) (a \text{double}) and \(3\) (an \text{int}), Java will automatically treat the \text{int} 3 as though it were the equivalent \text{double} \(3.0\). This way, Java can add the two numbers using rules for adding two numbers of the same type. This kind of treating numbers -- or other Things -- as though they had different type is called **coercion**. Coercion does not actually change the Thing, it simply provides a different version (with a different type). For dial types, this version is essentially a copy. For label types, it is another "view" of the same object. Coercion is described more fully in the sidebar on **Coercion and Casting in Java**.

Other arithmetic operators work in much the same way as \(+\). Additional information on arithmetic

---

**Java Notes**

**Types of Arithmetic Expressions**

Arithmetic expressions include the binary operators for addition \(\ast\), subtraction \(-\), multiplication \(\times\), division \(\div\), and the modulus or remainder operation \(\%\). In addition, there are two unary arithmetic operators, \(+\) and \(-\).

Arithmetic operations work only with values of type \text{int}, \text{long}, \text{float}, or \text{double}. When a (unary or binary) arithmetic expression is invoked with a value of type \text{short}, \text{byte}, or \text{char}, Java automatically widens that operand to \text{int} (or to a wider type if the other operand so requires). For further details on widening, see the sidebar on **Coercion and Casting in Java**.

When the operands of a binary arithmetic expression are of the same type, the complete expression also has that type, except that no binary arithmetic expression has type \text{short}, \text{byte}, or \text{char}. This is because operands of these types are automatically widened.

When the operands are of different types, Java will automatically widen one to the other.

The values of the expressions involving the binary operator \(\ast\), \(-\), \(\times\), and \(\div\) are the sum, difference, product, and quotient of their (possibly widened) operands, respectively.

The value of \(\times \div \%\) is the (appropriately widened) remainder when \(\div\) is divided by \(\%\).
The value of a unary `-` expression is the additive inverse of its (possibly widened) operand; a unary `+` expression has the value of its (possibly widened) operand.

expressions is summarized in the sidebar below. Note in particular that `/` (the division operator) obeys the same \textit{type op type is type} rule. This means that

\[ 7 / 2 \]

has type \texttt{int} (and the value 3). If you want a more precise answer -- \texttt{3.5} -- you can make sure that at least one operand is a floating point number:

\[ 7.0 / 2 \]

has type \texttt{double}, as does

\[ 7 / 2.0 \]

and (best style)

\[ 7.0 / 2.0 \]

In addition to the \textbf{binary} (two-argument) arithmetic operators described above, Java includes a \textbf{unary} minus operator that takes one argument and negates it. So \texttt{-5} is a (literal) \texttt{int}, while \texttt{-5} is an arithmetic expression that has value \texttt{-5} and type \texttt{int}. (Subtle, no?)

\section*{5.6.2 Comparator Expressions}

Not all operators are arithmetic. There is a set of boolean-yielding operators, sometimes called \textbf{comparators}, that operate on numeric types. These include \texttt{==} and \texttt{!=}, which have already been described, as well as more specialized numeric comparators such as \texttt{<} and \texttt{<=}. The sidebar on Java Operators contains a complete list.

Each of these operations takes two numbers, coerces them appropriately, and then returns a boolean indicating whether the relationship holds of the two numbers in the order specified. For example,

\[ 6 > 3.0 \]

is \texttt{true}, but

\[ 5 <= 3 \]

is \texttt{false}.

Evaluating one of these expressions is much like evaluating an arithmetic expression. The values of the operands are compared using a rule specific to the operator -- such as \texttt{>} or \texttt{<=} -- and the resulting boolean value is the value of the expression.
5.6.4 Logical Operator Expressions

Another set of operators combines booleans directly. These include && (conjunction, or "and") and || (disjunction, or "or"). For example, the expression true || false is true. While this is not very interesting by itself, these boolean operators can be used with names (of type boolean, of course) or in complex expressions to great effect. For example, rainy || snowy might be a reasonable way to express bad weather; it will (presumably) have the value true exactly when it is precipitating. There is also a unary boolean negation operator, denoted !.

The Java fragment

!(rainy || snowy || overcast)

might be a good expression for sunshine.

The rule for evaluating negation is simply to invert the boolean value of its operand. The rules for evaluating conjunction and disjunction are a bit more complex. First, the left operand is evaluated. If the value of the expression can be determined at this point (i.e., if the first operand to a conjunction is false or the first operand of a disjunction is true), evaluation terminates with this value. Otherwise, the second operand is evaluated and the resulting value computed. The type of each of these expressions is boolean.

These odd-seeming rules are actually quite useful. You can exploit them to insert tests. For example, you might want to compute whether (x / y) > z, but it might be the case that y is 0. By testing whether (y == 0) || ((x / y) > z), you can eliminate the potential divide-by-zero error. (If y is 0, the first operand to the disjunction -- (y == 0) -- will be true, so evaluation will stop and the value of the whole will be true. (A comparable formula can be written to return false if either y is 0 or (x / y) > z.)

5.6.4 Logical Operator Expressions

Java Notes

Other Assignment Operators

Compound Assignment

Java has several variants on the simple assignment statement. If we have already declared total as an int, we can say:

```java
total = 6
```
or

```java
total = total + 1
```

(The second expression uses the fact that total + 1 is an expression with type int and value one greater than total to form an assignment expression whose second operand is an arithmetic expression.) This last expression -- adding to a name -- is pretty common, and so it has a convenient shorthand:

```java
total += 1
```

The += operator is one of a class of compound assignment operators. It works by computing the value of its first operand, then adding its second operand to that value and assigning the result to the name represented by the first operand. In other words, the expression above is exactly the same as saying total = total + 1

This kind of compound assignment can be used with any number-or other appropriate expression -- as the second operand, of course. There are also other compound assignment operators in Java, including -=, *=, /=, and %=.

Like the * operator, the += operator works for both numeric addition and String concatenation. Like their longhand forms -- the simple assignment equivalents -- these expressions have type and value of their left-hand side (after the assignment).

5.6.3 AutoIncrement and AutoDecrement

There is another family of side-effecting operators that are related to assignment. These operators are autoincrement and autodecrement. The postfix autoincrement expression

```java
++total
```
is similar to total = total + 1 (or total += 1), but it has the value of total before the assignment. The prefix autoincrement expression

```java
++total
```
also adds one to total, but has the value of total after the assignment. (Remember: ++var first increments, then produces a value; var++ produces the value first.) The two (prefix and postfix) autodecrement operators work similarly.
5.7 Parenthetical Expressions and Precedence

A parenthetical expression is simply an expression wrapped in a pair of parentheses. The value of a parenthetical expression is the value of its content expression, i.e., the value of the expression between the ( and the ). The type of a parenthetical expression is the same as the type of the expression between the parentheses.

Parenthetical expressions are extremely useful when combining expressions. For example, suppose that the name \( x \) has the value 6 and consider the following expression:

"I have " + x + 3 + " monkeys"

The person who wrote this expression might well have meant

"I have " + (x + 3) + " monkeys"

which evaluates to "I have 9 monkeys". However, Java evaluates the expression by grouping subexpressions from the left, more like

\[
( ( "I have " + x ) + 3 ) + " monkeys"
\]

This evaluates to "I have 63 monkeys"! Isolating \( x + 3 \) with parentheses makes the \(+\) in \( x + 3 \) behave as addition, not String concatenation.

Note that, in giving the evaluation rules for expressions, white space doesn't matter --

\[
x >= 2 + 3
\]

is identical to

\[
x >=2 + 3
\]

-- but punctuation does matter. For example,

\[
2 + 3 * 2
\]

doesn't have the same value as \( 5 * 2 \)--

\[
2 + 3 * 2
\]

is 8. We can use parentheses to fix this, though:

\[
( 2 + 3 ) * 2
\]

is 10 again. In this case, parentheses change the order of evaluation of subexpressions (or, equivalently, how the expression is divided into subexpressions.) In the case of

\[
2 + 3 * 2
\]
if you evaluate the + first, then the *, you get 5 * 2, while if you evaluate the * first, you get 2 + 6. (Java evaluates the * first.)

How do you know which way an expression will be evaluated? In these situations, where one order of operation would produce a different answer from another, we fall back on the rules of precedence of expression evaluation. In Java, just as in traditional mathematics, * and / take precedence over + and -, so 

2 + 3 * 2

really is 8. (Another way of saying this is that the * is more powerful than the +, so the * grabs the 3 and combines it with the 2 before the + has a chance to do anything. This is what we mean when we say that * has higher precedence than +: it claims its operands first.)

A full listing of the order of precedence in Java is included in the sidebar on Java Operator Precedence. Parentheses have higher precedence than anything else, so it is always a good idea to use parentheses liberally to punctuate your expressions. This makes it far easier for someone to read your code as well.

Chapter Summary

- Evaluating an expression produces a Thing with a type and a value. Each kind of expression has its own rules of evaluation that determine the type and value of the Thing it produces.

- Simple expressions include literals and names.
  - A literal has its apparent type and value.
  - A name has its declared type and assigned value.

- Assignment expressions are generally used for their effects -- modifying the value associated with a (label or dial) name -- but, as expressions, also have type and value. The value of an assignment expression is the value assigned; the type is the type of the value assigned.

- Several kinds of expressions operate only on objects:
  - A method invocation expression asks an object to perform an action. It has the type and value returned by the method. Methods may be side-effecting.
  - A field access expression is like an ordinary name expression: its type is the field's declared type and its value is the field's current assigned value, except in the context of assignment expressions.
  - An instance creation expression's value is a brand new object whose type is the (class) type with which the constructor expression is invoked.
  - An type membership -- instanceof -- expression tells you whether an object can be treated as a member of a particular (class) type. Its type is boolean.

- Some expressions can operate on either primitive Things or objects:
• An explicit cast expression's value is the same as the Thing it is given, but with a different type.
• Equality testing expressions produce boolean values.
  • Two primitive Things are the same if they appear to be the same.
  • Two objects are the same only if they are actually the same object.
• Operator expressions combine or produce modifications of simpler expressions.
  • Arithmetic operators compute mathematical functions; the type of an arithmetic operation expression is typically the wider of its operand types.
  • Logical operators compute binary logical functions; the type of a logical operation expression is boolean.
  • Explicit cast expressions have the type of the cast operation and the same value as the cast operand.
  • None of these operations actually modifies any of its operands. However, autoincrement, autodecrement, and the shift operators do modify their operands.
• A compound expression contains multiple sub-expressions. It is evaluated by evaluating each of its constituent parts, then combining the resulting Things.
• Parentheses can be used to group expressions. Otherwise, expressions are evaluated in the order of precedence established by Java.

Exercises

1. In Java, every expression has a type. Assume that the following declarations apply:

```java
int i, j, c;
double d;
short s;
long l;
float f;
boolean b;
```

For each expression below, if it is syntactically legal Java, indicate its type (not its value). If it is not syntactically valid, indicate why.

a. 6
b. 24L

c. +3.5

d. 3.5f

c. 2e-16

f. -25b

g. i

h. i+3

i. i+3.0

j. i+s

k. l+d

l. f+s

m. i / 0

n. 4 * 3.2

o. i = 0

p. i == 0

q. b = 0

r. b == 0

s. 'c'

t. "An expression in double-quotes"

u. "An expression in double-quotes" + "another one"

v. "6" + 3

w. !b
2. Give examples of three expressions with side effects.

3. For this question, you may wish to consult the sidebar on Java Operator Precedence. Assume the following definitions:

```java
int i = 93;
boolean b = true;
```

What is the value of each of the following expressions? Which ones produce errors in evaluation?

a. 2.0 + 3.5 * 7

b. ("top " + "to " + "bottom").toUpperCase()

c. "the answer is " + 6 * 7

d. 4 + 6 + "is " + 10

e. i > 0 && i < 100
4. Assume that $x$ and $b$ are previously defined names for an `int` and a `boolean`, respectively. Give examples of each of the following:

a. An expression whose type is `int` and whose value is more than a previously defined $x$.

b. An expression whose type is `boolean` and whose value is true when $x$ is between 5 and 15.

c. An expression whose type is `double` and whose value is half of $x$'s.

d. An expression whose type is `long` and whose value is the remainder when $x$ is divided by 7.

e. An expression whose type is `boolean` and whose value is the opposite of the boolean $b$.

f. An expression whose type is `boolean` and whose value is true exactly when the int $x$ is evenly divisible by 5.

g. An expression whose type is `String` and whose value is read from the user's keyboard.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:

<webmaster@cs101.org>
Statements and Rules

Chapter Summary

- How do I tell the computer how to do something?

This chapter introduces **statements**, the simplest forms of complete executable instructions. Statements are fragments of Java code that have neither value nor type; instead, they have effects. Statements can be combined to form rules, or services that one object can provide to another. Statements and rules form the backbone of the peanut-butter and jelly model of programming.

Statements can be built out of expressions. However, unlike expressions, which have both type and value, statements are used for their effect -- to get something done. Examples of this are asking a thing to do something or assigning a name to keep track of a value. In addition to declarations, assignments, and method invocation, this chapter introduces simple control flow statements. More advanced statement types are introduced later in the book.

The chapter ends with a discussion of methods, the rules implementing behavior. Method invocation provides the basis for virtually all inter-object interaction.

This chapter is supplemented by a reference chart on the syntax and semantics of java statements.

Objectives of this Chapter

1. To appreciate the difference between evaluating an expression and executing a statement.
2. To be able to read and understand basic statements including assignments, method invocations, declarations, blocks, conditionals, and loops.
3. To learn how to combine statements to construct rules that implement method behavior.

Statements and Instruction-Followers

In the first chapter of this book, we saw that computations are made of communities of interacting entities. Each of these entities may be a community of smaller entities, until eventually an entity can be subdivided no more. At that point, an entity is a simple instruction-follower that provides behavior -- often in the form of ongoing services -- to the other members of its community. This chapter is about how those instructions work. Towards the end of the chapter, we will begin to see how instructions can be combined to form special sequences that articulate how service requests can be fulfilled.
In the previous chapter, we saw how to create Java expressions. An expression is a piece of Java code with a value and a type. The process of producing the value from an expression is called evaluating that expression. The purpose of evaluating an expression is generally to produce its value.

In contrast, statements are all about their side effects. A statement is a piece of executable Java code without either a type or a value. That is, a statement does something (changes something, produces some visible behavior, etc.). It has an effect. It does not have a value. A statement is executed (producing an effect), not evaluated (producing a value).

In order to evaluate an expression, you must evaluate its subexpressions, then use the evaluation rule for that kind of expression to produce an appropriate value of an appropriate type. If you understand the evaluation rules for each type of expression, you understand how expressions work.

Understanding how to execute a statement is similar. A statement is not defined by a type and a value (it doesn't have either!), but by its effects and by what happens next. That is, statements do things; they change the values associated with names. And statements can also cause you to skip around in the instructions that you are following. This is called flow of control: what instruction to follow next. Some of these control flow statements involve conditions (if it's raining, do this) or loops (keep doing this until the light changes color). And many statements involve either subexpressions--which must be evaluated--or substatements--which must be executed in order to execute the superstatement.

**Simple Statements**

Perhaps the simplest kind of statement is one built directly out of an expression, such as

```java
this.who = name;
```

or

```java
Console.println( "Hello" );
```

Note the trailing semicolon following the ends of these expressions. It is this semicolon that converts these expressions into statements.

What kinds of expressions can be used to form statements? Only side-effecting expressions. Many expressions are useful solely because of the value that they compute. But a statement doesn't have a value; it has effects on state and control flow. So an expression whose primary purpose is the value it produces doesn't make a very good basis for a statement on its own.[Footnote: These expressions may find use in other, more complex statements, though.] In fact, it is not legal in Java to make an expression-semicolon statement out of a non-side-effecting expression. (For example, `x + 3;` is not a legal statement.)

However, some expressions do more than just produce values when they are evaluated. For example, an expression like `x = 3` has the value `3` (and the type `int`, assuming that `x` is an `int`). It also (and more importantly) has the effect of storing the value `3` in the shoebox named `x`. This effect (of evaluating the expression) is called a **side effect**. All assignment expressions (including compound assignments) are side effecting. Autoincrement and autodecrement are also side-effecting expressions. Method invocation expressions are also side-effecting, although not every method invocation actually has a side effect. Instance creations -- `new` expressions -- are also side-effecting.
So, for example, a simple assignment statement can be made by adding a semicolon to the end of the assignment expression $x = 3$

$x = 3;$

The semicolon turns this into a statement. It no longer has a value or a type; it just does its work.

To execute an expression-semicolon statement, simply evaluate the expression. Of course, this expression may have complicated subexpressions that must be evaluated according to the rules described in the previous chapter. Since the expression is a side-effecting one, something will happen -- an effect will be produced -- during the evaluation.

After executing a side-effecting-expression-plus-semicolon statement, execution proceeds at the following statement.

**Declarations and Definitions**

We have also already seen declarations in Chapter 3. A declaration creates a new name that can be used to store (in the case of primitive types) or label (in the case of reference types) a value. A declaration follows the `type-of-thing name-of-thing` rule: It consists of a Java type followed by a Java name, then a semicolon. For example,

```
int i;
Object thing;
```

A declaration (or definition) statement creates a kind of name called a **local variable**.

You can actually declare multiple names of a single type with one declaration statement. The syntax for this is `type-of-thing name-of-thing1, name-of-thing2, and so on, with commas between the names and a semicolon at the end:

```
int i, j, k;
Object thingOne, thingTwo;
```

The same type is associated to each of the comma-separated names, so the declarations above are identical to

```
int i;
int j;
int k;
```

and

```
Object thingOne; Object thingTwo;
```

respectively.

**Style Sidebar**
Formatting Declaration Statements

Remember that Java doesn't care how much white space you leave between things, so there is no difference in meaning between putting the multiple declarations on one line or many. It is definitely easier to read on multiple lines, though, so the convention is to put each declaration on its own line.

When one declaration statement is used to declare many names, you can put the names on one line or on several. It's good style to indent all of the names on subsequent lines of a single declaration so that they line up with the first name declared:

```
Object thingWithALongName,
    anotherThingWithALongName;
```

This way, it's easy to see that `anotherThingWithALongName` is involved in the same declaration statement as `thingWithALongName`.

Although it is technically correct to mix declarations and definitions of a single type using the comma-separated multiple declaration notation, this is not good style. It is too easy to miss a definition among the declarations; mixing the two makes your code unnecessarily harder to read.

A declaration makes it legal to use the name to hold/label appropriately typed values. But the declaration, by itself, doesn't explicitly assign a value to the name. In fact, for the most generic kind of name--a local variable--it is illegal to use a name without first assigning it a value. [Footnote: It is, however, legal to assign a label-name local variable the special non-value `null`. Assigning `null` to a name means that the name doesn't refer to anything. Not assigning forces the computer to guess. The rule is that you just can't leave the computer to guess.] You can assign this value directly in the declaration (making it a definition), or you can assign it before the first time that you try to use the name's associated value.

A variant on a declaration statement is a definition. A definition is a declaration statement with `=` `expr` between the `name-of-thing` and the semicolon (or comma). This statement declares the name, but it also assigns it the value of `expr`. For example:

```
int i = 2;
String who = "Pat";
double pi = 3.14159,
    ninetyDegrees = pi / 2;
```

Note that the final statement here assigns the value 1.570795 to the name `ninetyDegrees`. First 3.14159 is put into the shoebox named `pi`. Next, the expression `pi / 2` is evaluated: its value is the value inside the `pi` shoebox divided by 2. Finally, this value is assigned to (stored in) the (newly created) shoebox named `ninetyDegrees`.

It is legal to mix declarations and definitions in a single statement -- assigning initial values to only some of the names -- but this can make your code hard to read. It is usually better to use multiple statements in this case.

Executing a declaration statement creates a shoebox or label associated with the name declared. Executing a definition is the same as declaring a name, plus immediately afterwards executing an assignment.
statement. Note that this assignment is an expression and may have subexpressions, causing a significant amount of evaluation before execution is complete.

After executing a declaration or definition statement, execution proceeds at the immediately following statement.

**Sequence Statements**

You can also make a bigger statement out of a collection of statements. You do this by enclosing them in braces:

```
{ 
    int i = 3; 
    Console.println( "i is " + i ); 
    int j = i + 1; 
    j = i + 5; 
}
```

This statement-made-of-statements is a block, and it mostly serves to organize your code. Some other statements -- such as `if`, described below -- are often used together with blocks.

Any statement can be used at any point inside a block. In particular, declarations and definitions may appear anywhere in a block. This is useful as it allows you to declare a name immediately before you need it. Doing so makes it easier to read your code as the reader is less likely to have forgotten what you mean by that name.

Blocks also have implications for scoping of names: a variable has scope (its name can be used) from the point in the code where it is declared until the end of the first enclosing block. [Footnote: Remember, not all names are variables. We will learn more about parameters and fields in subsequent chapters. Type names have scope everywhere that they are visible.] So if we declare a name at the top of the block, it has scope for the whole block, as `i` does in the example above. But `j` is not declared until after the call to `println`, so the definition of `i` and the call to `println` are outside of `j`'s scope:

```
{ 
    int i = 3; 
    Console.println( "i is " + i ); 
    int j = i + 1; 
    j = i + 5; 
}
```

This means, for example, that it would be illegal to use `j` in `i`'s definition:

```
{ 
    int i = j;  // illegal use of j outside its scope! 
    Console.println( "i is " + i ); 
    int j = i + 1; 
    j = i + 5; 
}
```
**Beware:** The scope of a local variable only persists until the end of the enclosing block. This means that a local variable must be declared at the same level as (or at a level enclosing) each of its uses.

```java
{
    // A variable declared here...
    String name;
}
// ...is invisible here, making this reference
name = "Pat";
// illegal!
```

// ...and so on.

The rules for executing a block statement are: execute each substatement in turn, from the top (beginning) of the block to the bottom (end) of the block.

After a block, execution continues at the next statement.

**Style Sidebar**

**Formatting Blocks**

The open brace of a block should generally appear on its own line. If the block is part of a compound statement (such as an if), its opening brace can appear as the last character on a line. However, studies have found code using this convention harder for programmers to scan than code in which the open brace appears alone on a line.

Text within a block should always be indented (typically by two or four characters). This makes the left-hand margin of code in a block line up. The text -- but not the braces -- of an interior block is indented further; the original indent is resumed when the interior block is closed, i.e., after the closing brace.

The closing brace of a block should always begin its own line. If the closing brace completes the statement, as in a simple block, it should appear alone on that line.

```java
// Some statements...
{
    // Statements in a block
    // all line up.
    {
        // Interior block statements
        // are indented further.
    }
    // Close brace exits the block
    // and restores earlier indent.
}
// ...and so on.
```
Flow of Control

So far, we have seen declarations, definitions, and a few executable statements made out of side-effecting expressions such as method invocation and assignment. You can write some interesting programs using only these constructs, but typical programs involve more complex structures. One of the most important features is the ability to control which code is executed when. This is called flow of control. These statements have execution rules that do not always cause the next statement to be executed in turn. Instead, a statement may be executed more than once or not at all.

Simple Conditionals

One of the simplest forms of control flow is conditional execution. Conditional execution refers to a situation in which a block of code may or may not be executed, depending on the value of an expression. It is analogous to a set of instructions that says

Step 1. If your gizmo is not already assembled, you must assemble it before going on to step 2. To assemble your gizmo, first....

Step 2. Now that your gizmo is fully assembled, ...

In Java, conditional execution is most often and most generally embodied in the `if` statement. For example:

```java
if ( theLight.isOn() )
{
    theRoom.isLit = true;
}
```

Let's dissect this statement. It begins with the java keyword if. After the if is a boolean expression that must be enclosed in parentheses. The closing parentheses are followed by a block statement.[Footnote: There are other kinds of statements that can appear in place of this block, but in this book we will restrict ourselves to the cases in which the if body is a block.] This block is sometimes called the if statement's body or the consequent; the boolean expression is called the if statement's test or condition.

Execution of the if statement proceeds as follows. First, the boolean condition expression is evaluated. If the value of this expression is true, the if's body block is executed. If the value of the boolean condition expression is false, the if's body block is skipped.

In either case, execution proceeds at the next statement following the if's body.

The if statement, as defined, is very useful when you want to do something or skip it. But often you want to do one of two things. We can express this using two if statements with inverse conditions:

```java
if ( theLight.isOn() )
{
    theRoom.isLit = true;
}
if ( ! (theLight.isOn() ) )
```
theRoom.isLit = false;

This is poor code in three ways. The first is that it invokes the same method -- theLight.isOn() -- twice, but the code would not work as we want if the value returned were different in the two invocations. (Imagine that the light were off the first time you asked and on the second time. The value of theRoom.isLit would never get set!)

We could fix this problem by temporarily assigning this value to a boolean name, and then testing the name twice:

```java
boolean itIsLight = theLight.isOn();
if ( itIsLight )
{
    theRoom.isLit = true;
}
if ( ! itIsLight )
{
    theRoom.isLit = false;
}
```

But this makes a second problem with the code even more apparent. This code is testing a boolean expression (theLight.isOn() or itIsLight, depending on which version) in order to set another boolean expression. It would be cleaner just to write

```java
theRoom.isLit = theLight.isOn();
```

This statement is equivalent to the whole previous example (using itIsLight), and much easier to read. For more on this stylistic point, see the sidebar on Using Booleans.

Of course, we can write other code that's not subject to these two problems. For example, we could use this idea to write code to compute absolute value of a given int, x.

```java
int absValue;
if ( x > 0 )
{
    absValue = x;
}
if ( x < 0 )
{
    absValue = - x;
}
if ( x == 0 )
{
    absValue = 0;
}
```
This code has neither of the previous problems -- x doesn't change, so we can test it repeatedly, and the value assigned is an int, not a boolean, so we can't write the shorter assignment statement. But this code doesn't make it clear that these are really three cases of the same test. There is a form of an if statement that allows us to make this clearer. It uses the Java keyword else to denote a situation in which we know that these conditions are mutually exclusive, i.e., at most one of them can hold.

So, for example, we could rewrite our light-tester (verbosely) as:

```java
boolean itIsLight = theLight.isOn();
if ( itIsLight )
{
    theRoom.isLit = true;
}
else
{
    theRoom.isLit = false;
}
```

This still isn't as nice as the one-line version, but it gives us the opportunity to illustrate control flow in an if/else statement. To execute an if/else statement:

1. Evaluate the boolean condition expression.
2. If the value of the condition is true, execute the if body block, then skip to the end of the entire if/else statement (i.e., to step 4).
3. Else (the value of the condition statement is false, so) execute the else body block. An else body is sometimes called an alternative.
4. Execution continues at the following statement.

Since there might be more than two mutually exclusive conditions -- as in the absolute value code -- else is allowed to have its own condition. An else with a condition is like an if, except that you only execute that part of the statement if all previous conditions in this if/else statement have been false. An else with no condition is always executed if no previous condition in this if/else statement has been true.

```java
if ( x > 0 )
{
    absValue = x;
}
else if ( x < 0 )
{
    absValue = - x;
}
else
{
    absValue = 0;
}
```

Note that this is all one statement, not three as in the previous version. Exactly one of the assignment statements will be executed, no matter what the value of x at the beginning of the if statement.
Even now, this is not the most elegant absolute value code we could write; for example, the final case is redundant and could be folded into the first case using $\geq$ instead of $>$. It does, however, illustrate the syntax of cascaded if statements. We will return to examine if statements, and other conditionals, in the chapter on Dispatch.

**Style Sidebar**

**Using Booleans**

There are only two boolean values, **true** and **false**. There can be lots of boolean labels, but each label is attached to either **true** or **false**; there is nothing else. This means that testing whether a boolean is the same as **true**--

```java
    (boolVal == true)
```

-- is redundant. You can just use `boolVal`, since it's either **true** or **false**. Similarly, you don't need to use an if statement to test a boolean if you're generating a boolean value. For example,

```java
    if (boolVal) {
        return true;
    } else {
        return false;
    }
```

is also redundant: just `return boolVal;`. The same thing applies if you're assigning to a variable instead of returning: `otherBoolVal = boolVal;` (or `otherBoolVal = ! boolVal;` if you want to reverse its sense).

**Simple Loops**

Another flow-of-control construct is **while**. **While** takes a condition and a block, just like the simple form of **if**. Execution of a while statement first evaluates its boolean condition expression. If the condition is true, the while body block is executed. When execution of each statement in the body is complete, the while's condition is checked again. Again, if the condition is true, the body is executed. This continues until the evaluation of the condition expression yields false; at this point, execution continues at the next statement after the while body.

There are several uses of a while loop. One is to continually test something until it becomes true:

```java
    int i = 1;

    while (i < 100) {
        Console.println("I'm up to " + i);
        i = i + 1;
    }
```

This loop prints the numbers from 1 to 99. (Why doesn't it print 100?)
Another use is for a loop that keeps going essentially forever. (It will stop when something stops the program, but not before:

```java
while (true) {
    myOutput.writeOutput( myInput.readInput() );
}
```

This loop continually passes whatever input it gets to its output. Since the value of true doesn't change, this loop won't end until something nasty happens to it. Writing loops like this one -- that go on essentially forever -- is much easier than writing loops like the counting loop, above, because in the counting loop you have to keep track of what's true each time you go around the loop. For example, the value of i when you exit the loop above will always be one more than the last value printed.

Here's an even more tricky one:

```java
while (x < 25) {
    x = x + 3;
    x = x - 2;
}
```

If x's value is 20 when we reach the beginning of this loop, what will its value be when we exit? Remember that the test expression is only checked at the beginning of each pass through the loop, not in the middle.

There is another looping construct in Java, called do/while statement or just a do loop. It is much like the while loop, except that the loop body is always executed once before the condition is tested:

```java
int i = 1;

do {
    Console.println( "I'm up to " + i );
    i = i + 1;
} while (i < 100)
```

As with a while loop, once the loop exits, execution proceeds at the statement following the entire do statement.

**Statements and Rules**

Programs are not simply sequences of instructions to be executed. Instead, the instruction-followers executing these statements are embedded in a community of other instruction-followers. A program is a community of interacting entities providing ongoing behavior and services. In this section, we look at how those interactions too rely on statements.

When one Thing needs to communicate with another, this is commonly accomplished through method invocation. Method invocation is an expression in which one object supplies another with information (in the form of arguments), and the second supplies the first with other information (in the form of the return
value). These mechanisms are the major means of inter-object communication and coordination. Of course, method invocation can also be used within an object, allowing one part of the object to communicate with another.

We have previously seen how interfaces specify methods that an object provides. Now, we turn to the question of how method behavior is actually implemented. Statements provide the key. Performing a method amounts to following the instructions associated with that method, i.e., stepping through the instructions for that rule. Statements are the steps of those instructions. By sequencing statements, you can build a rule that the computer can follow to accomplish a desired task. Some rules require information in order to accomplish their tasks. (For example, a rule that doubles a number needs the number to be doubled.) Some rules produce results. (For example, the doubling rule might produce the doubled number.) Some rules behave differently under different circumstances. (This uses a conditional statement).

In order to use a rule -- to interact with it -- you need to know whose rule it is, what information you need to supply in order for the rule to do its work, and what the rule will give you in return. This prefigures the idea of method signature. There are other things you'd like to know about a rule -- such as the relationship between the rule's input and its output -- and these form the basis of the rule's documentation.

For example, here is a rule for printing a brief form letter:

```
1. print "Dear 
2. if ( title isn't null ) print title + lastName else print firstName
3. println ":\nWe are tremendously pleased to inform you that "
4. println "you have won!".toUpperCase()
5. println "Not much, but what did you expect?"
6. println " Sincerely,\n me"
```

It's just a short hop from this pseudocode rule to real Java:

```
void printFormLetter( String title, 
 String firstName, 
 String lastName )
{
    if ( title != null )
    {
        Console.print( title + lastName );
    }
    else
    {
        Console.print( firstName );
    }
    Console.println( ":\nWe are tremendously pleased " 
 + "to inform you that ");
    Console.println( "you have won!".toUpperCase() );
}
```
Method Invocation Execution Sequence

Method invocation is, as we have seen, an expression. To invoke the printFormLetter, we need to know whose method it is. We follow this object expression with a dot, then the name of the method, then the parentheses-enclosed parameter list:

```java
theWidgetCompany.printFormLetter( "Prof.", "Pat", "Smith" )
```

To evaluate this expression, we need to invoke the `theWidgetCompany.printFormLetter` method (using the rule, or instructions, or method body, provided above) with the arguments "Prof.", "Pat", and "Smith".

The first step in method invocation is parameter binding. In this step, each parameter name (title, firstName, and lastName) is treated as though it were newly declared and it is given the value of the corresponding argument. (Recall that parameters are the names in the method declaration, while arguments are the values supplied in the method invocation expression.) In order for this to work, each value must be assignable to the corresponding parameter's declared type.

After parameter binding, method invocation proceeds as though the method body were a simple block. The block is, however, within the scope of the parameter bindings, so that inside the block the parameter names can be used to refer to the provided argument values. For example, in the body of the printFormLetter, title is bound to "Prof", firstName is bound to "Pat", and lastName is bound to "Smith".

Now the body statements are executed in turn. In this case, the first statement is an if, so its test expression is evaluated to determine whether to execute the consequent block or the alternative block. When the test expression

```java
title != null
```

is evaluated, title is bound to "Prof", so it is not null, causing the consequent to execute.

This argument-value-providing is one way in which method invocation implements inter-entity communication: the value is communicated from the method-invoker to the method owner.

**Return**

This special statement can only be used inside method bodies. It is used to terminate the execution of the method body. It is also what is responsible for making a method body -- which is essentially a block statement -- return a value -- which is a necessary property of a method invocation expression (unless the method's return type is void).

The need for this statement arises when the sequence of instructions that you are writing is turned into a method body. In this case, you need to say what the method *returns*. This return value becomes the value
produced by evaluating a method invocation expression. This is accomplished using a return statement. The syntax of a return statement is

\[
\text{return expression;}
\]

where expression can be any arbitrary Java expression. Remember: the return statement -- a statement -- does not have a value, but the method invocation -- an expression -- does.

To execute a return statement, evaluate the expression. Then, exit the enclosing method, providing the value of the expression as the return value of the method invocation expression.Exiting the enclosing method means both exiting from the block that is the method body and also exiting the scope of the parameter/argument bindings.

After a return statement, execution proceeds at the method invocation whose method body contained the return statement; evaluation of this expression is complete (with its value the value supplied by the return statement) and execution of the statement containing the method invocation continues.

For example, if we execute

\[
\text{String transformed = this.transform( "Knock, knock" );}
\]

and the transform method of this object ends with the line

\[
\text{return "Who's there?";}
\]

then the value of the invocation this.transform( "Knock, knock" ) is "Who's there?". Execution continues by assigning the value of the invocation ("Who's there?") to the name transformed.

Another example is the doDouble( int ) method mentioned above. The code for doDouble might read:

\[
\begin{align*}
\text{int doDouble( int whatToDouble )} \\
\{ \\
\hspace{1em} \text{return whatToDouble * 2;}
\}
\end{align*}
\]

To evaluate the application of doDouble to 7,

1. The parameter name whatToDouble is bound to 7.
2. Within the scope of this binding, the body block of doDouble is executed.
   a. Each statement in the block is executed in turn. Since there is only one statement, it is executed.
      i. The expression whose value is to be returned is evaluated. This requires evaluating the subexpressions (name whatToDouble and literal 2) and then applying the operator to these values.
   2. The value produced by the operator expression (14) is returned by the method
3. This exits both the method body block and the parameter scope, providing the value (14) as the value of the method invocation expression.

There is also an alternate form of return that does not take an expression. This form is used in methods whose return type is void. In this case, a return statement executes by exiting the method (and, with it, the scope of the parameter names). Since the simple return statement is used only in methods whose return type is void, there is no value for it to supply.

This return statement can also be left implicit certain methods. For example, in the printFormLetter method that we saw above, there was no explicit return statement. In Java, a method without a return statement is presumed to have a return statement as its final statement. This return statement is a simple return; -- it is the form that does not return a value. So the end of that method body was equivalent to saying

```java
//...
Console.println( "                Sincerely,
+ " Sincerely,\n" + " me" );
return;
}
```

In a method whose return type is not void, an explicit return statement must always be executed in order to provide the method's return value. Value-returning is another example of inter-object communication.

**Chapter Summary**

- Statements combine expressions to produce useful behavior.
- A statement does not have a value or a type.
- A statement is executed to produce an effect.
- A side-effecting expression followed by a semicolon is a simple statement.
- Declarations and definitions are also simple statements.
- A sequence of statements can be grouped into a block by surrounding the sequence with braces { }
- Conditional statements allow you to write code containing alternative execution sequences. The execution sequence of a conditional statement depends on the result of evaluating a boolean expression.
- A loop allows the same block of code to be executed repeatedly, until an exit condition -- a boolean expression -- is true.
- A return statement is used to exit from a method, with or without a value.
- Method bodies, or rules, use sequenced statements -- including loops and conditionals -- to produce chunks of executable behavior. A method is specified by its name, the information it needs, and the value (if any) that it produces.

**Exercises**
1. Using Java's if statement, write instructions for determining which team returns an out-of-bounds ball to play in a soccer game. In soccer, the team that did not last touch the ball receives possession of the ball and returns it to play.

a. You may presume that you have a method, lastTouch(), that returns either homeTeam or visitTeam, and that the goal of your code is to assign the correct team value (either homeTeam or visitTeam) to the already-defined name possessingTeam.

b. In addition, make your code determine whether returnBallToPlayMethod is sideThrow, cornerKick, or goalKick. You may make use of the ballOutLine() method to determine whether the ball exited via the sideLine, the homeEndLine, or the visitEndLine. [Footnote: If the ball has exited via the side line, the return is by side throw. If the ball exits via the home end line and is last touched by the home team, the visitors return the ball to play by means of a corner kick. A ball that is pushed beyond the home end line by the visiting team is returned by the home team via a goal kick. The situation at the visitor's end line is the opposite.]

2. Using Java's while statement, give instructions for building a tall tower of blocks.

3. Using Java's while statement, give instructions for blowing up a balloon.

4. Which of the following are expressions, which statements, and which illegal? For the expressions, indicate the type and value. For the statements, indicate the effect (if known) and the execution sequence. You may assume that x is an int, b a boolean.

   a. int x = 5
   2. boolean b;
   3. x + 3
   4. x = x + 3
   5. x = x + 3;
   6. x == 3
   7. x == 3;
   8. b = x == 3;
   9. {
      Console.print( "What is your name? " );
      String name = Console.readln();
      String cap = name.toUpperCase();
   }

5. What will the value of d be after each of the following statements? Also, indicate any other changes that may occur as a result of executing the statement. You may assume that they are executed in the order given.

   a. double d = 3.5;
   2. d = d * 3;
   3. if ( d < 8 )
4. d = 2.0
5. while ( d < 30 )
   {
   d = d * 2;
   }

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments: <webmaster@cs101.org>
Interlude: Expressions and Statements

Overview

This interlude explores what you've learned so far about Java expressions and statements. There is a supporting executable, distributed as a part of the on-line supplement to this book, which allows you to experiment with the ideas described here. The goals of this interlude are to show you how expressions and statements can be used in context and simultaneously to give you an opportunity to explore interactions among entities and how these interactions can be used to generate a variety of basic behaviors.

Objectives of this Interlude

1. To increase familiarity with expressions and statements, including return statements and conditionals.
2. To be able to read and write simple sequences of instructions.
3. To appreciate how multiple independent instruction followers can produce behavior through their interactions.
4. To begin to appreciate the differences between parameters, local variables, and fields.

The Problem

This interlude is inspired by a simple child's toy called an Etch A Sketch®. In case you may not be familiar with an Etch A Sketch®, here is a brief description:[Footnote: The Etch A Sketch® product name and the configuration of the Etch A Sketch® product are registered trademarks owned by the The Ohio Art Company. Used by permission.]

An Etch A Sketch® is a rectangular frame (generally red) with a silver screen in the center and two white knobs, one in each of the lower corners. Inside the silver screen is a point of darker grey:

Figure 1: A simple drawing tool.
An Etch A Sketch® is a simple drawing tool. Turning the knobs moves the darker grey point around the screen. As the point moves, it leaves a darker grey trail behind it, showing where it has been. By coordinating the motion of the knobs, you can draw pictures.

On an Etch A Sketch®, each knob controls one direction of motion of the darker grey dot. Rotating the left knob moves the dot from side to side. Rotating the right knob moves the dot up and down. Keeping a knob still prevents the dot from moving in the corresponding direction. So the position of the knob determines the position of the dot on the screen (in one dimension) and changing the knob position moves the dot (in that dimension).

By rotating just one knob -- by leaving the position of the other knob fixed, or constant -- you can draw a straight (horizontal or vertical) line, as in figure 2. By rotating both knobs at appropriately coupled velocities, you can draw diagonal lines of varying slope. Proficient Etch A Sketch® users can draw complex pictures by coordinating the knob position changes.

![Figure 2: Drawing a straight line by rotating the appropriate knob.](image)

In this interlude, we will explore the instructions required to perform similar operations on a similar (though less brightly colored) display. In our application, we will supply a rule that describes the current position of each knob. For example, the position might be constant -- always the same -- in which case, the dot wouldn't move. Or it might be steadily increasing, in which case the dot would move steadily across the screen.

Each rule will be read (and executed) repeatedly. It is as if, behind the scenes, an instruction-follower were to continually check the position of each knob and update the position of the dot correspondingly. Each time that the instruction follower wants to know what to do with the dot, it will ask our rule. If the rule always gives the same value, it will be as though that knob is stuck in one position. If the rule changes the value, the same change will be made to the knob's position over and over again. So, for example, if the rule says "increase the current position", the dot will move across the screen until it reaches the edge.

In fact, there will be two instruction-followers, one for each knob. In addition, they're not guaranteed to run at the same speed, or even to take fair turns. So, even if both knobs were to have the same rules, we might discover that our horizontal knob was moving twice as fast as our vertical knob.

Q. What would the resulting picture look like?

Ans. It would slope upwards gradually. @@supply pic.
Or the horizontal knob instruction follower might check its rule three times, then the vertical knob once, then the horizontal knob once, then the vertical knob three times, then both at once.

Since the knobs are being checked by independent instruction-followers, any schedule is possible in principle. By observing the actual behavior of the system, we can try to write rules to explicitly coordinate the behavior of the two knobs. To begin with, we'll just assume that they run at about the same rate.

An interesting feature of our program is that the knob-rules don't have any way to tell which knob they're controlling. The rules just say things like "turn the knob to a higher value" or "turn the knob lower" or "set the knob to the middle" (or "...halfway to the edge").

But on to the details....

**Representation**

In our application, instead of actually rotating knobs, we will represent a knob position as a number. We will use a standard Cartesian coordinate frame, with (0,0) in the center, increasing horizontal coordinates to the right, and increasing vertical coordinates at the top. [Footnote: In a later chapter, we'll see computer graphics that use a different coordinate system, sometimes known as "screen coordinates".]

![Figure 3: Cartesian coordinates.](image)

The value of the knob position can range between $-\text{maxPos}$ and $\text{maxPos}$. Since (0,0) is dead center on the screen, $(0, \text{maxPos})$ is the center top of the screen, while $(-\text{maxPos}/2, 0)$ is middle height, halfway to the left edge. The actual size of the Etch A Sketch® window (and therefore the value of $\text{maxPos}$) will vary as the window is resized. [Footnote: In particular, $\text{maxPos}$ may have different values in the horizontal and vertical rules. $\text{maxPos}$ is simply the largest coordinate in whichever dimension the rule controls.]

**Q.** What four points represent the four corners of the Etch A Sketch® window?

**ans.** $(-\text{maxPos}, -\text{maxPos})$, $(-\text{maxPos}, \text{maxPos})$, $(\text{maxPos}, -\text{maxPos})$, $(\text{maxPos}, \text{maxPos})$. These are lower left, upper left, lower right, and upper right in order. But note that $\text{maxPos}$ in the vertical dimension may not have the same value as $\text{maxPos}$ in the horizontal dimension, and also that the value of $\text{maxPos}$ may change as the window is resized.

Our job will be to write the instructions for the next knob position: a rule that returns the desired dot position. We'll need one rule for the horizontal knob and one for the vertical, of course.

The form of a control rule is a sequence of Java statements ending in a statement of the form
return double;

where double is some Java expression with type double. The value returned by your control rule will be used as the new position of the dot. So, for example,

    return 0;

is a rule that holds the knob in the center, i.e., keeps the dot in the middle of the screen. (Why is it ok to return 0 rather than 0.0?)

Remember, though, that this is just one rule. The other rule might be moving the dot along steadily, or holding it at the edge of the screen, or doing something else entirely. If return 0; is the vertical rule, then the dot will remain halfway up the screen, but it could be anywhere along that halfway line.

Q. What rule would keep the dot centered horizontally (halfway between the left and right edges) on the screen? Is this a horizontal- or a vertical-knob control rule?

ans. return 0; as a horizontal-knob control rule. Using this rule, the dot may move up and down (depending on the vertical control rule), but will always be halfway between the two sides of the screen. Its horizontal position is fixed, i.e., constant.

**horizontal rule**

    return 0;

If you use this rule as a vertical control rule, it will keep the dot halfway up the screen. Its side-to-side (horizontal) position would be determined by the horizontal control rule.

**vertical rule**

    return 0;

Q. How would you position the dot almost in the upper right-hand corner? The answer should involve a horizontal rule and a vertical rule.

ans. The horizontal rule should position it at the far right of the screen. Examining the coordinate system (figure ??), we see that this is at maxPos. So return maxPos; as a horizontal-knob control rule. The vertical control rule should position the dot at the top of the screen, which is maxPos in that dimension. So the vertical rule is also return maxPos;

    horizontal rule          vertical rule
    return maxPos;           return maxPos;

Q. How about the upper left-hand corner? Again, the answer should involve a horizontal rule and a vertical rule.

    horizontal rule          vertical rule
    return maxPos;           return maxPos;
ans. In this case, the horizontal rule should put the dot at the far left, which (according to the coordinate frame in figure 3) is at \(-\text{maxPos}\). So the horizontal rule should be 
\[
\text{return } \neg\text{maxPos};
\]
The vertical rule should put the dot at the top, which is still \text{maxPos}, so the horizontal rule should be 
\[
\text{return maxPos};
\]
as in the previous question.

<table>
<thead>
<tr>
<th>horizontal rule</th>
<th>vertical rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>return (-\text{maxPos});</td>
<td>return \text{maxPos};</td>
</tr>
</tbody>
</table>

The vertical rule has not changed. But because it is interacting with a different horizontal rule, the on-screen behavior is different.

Q. Assume that you have rules that position the dot in the upper left-hand corner. Now suppose that you swap the horizontal and vertical rules. Where is the dot now? What if you use the horizontal rule for both horizontal and vertical behavior?

ans. This question involves swapping the horizontal and vertical rules from the previous question:

<table>
<thead>
<tr>
<th>horizontal rule</th>
<th>vertical rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>return \text{maxPos};</td>
<td>return (-\text{maxPos});</td>
</tr>
</tbody>
</table>

Note that it is the vertical rule that now returns a negative value. Examining the coordinate frame in figure 3, we see that the horizontal rule puts the dot on the right-hand edge of the frame, while the vertical rule puts the dot at the bottom. So the result will be a dot in the lower right-hand corner of the frame. Swapping the rules -- so that the same two rules play different roles -- changes the behavior of the system, too.

A rule is a sequence of statements including a value returning statement. A rule is executed by following the instructions until a return statement is reached. This kind of rule is essentially the body of a Java method. This application focuses attention on the sequence of statements -- the method body -- but its ideas apply to Java methods as well.

**Interacting With The Rules**

In introducing this system, we said, "Each rule will be read (and executed) repeatedly." But nothing about the observable behavior of the system has really made this clear. In fact, in the examples above, each rule might be followed only once and the same results would still be produced. Each of the rules that we have used so far has been like setting the Etch A Sketch® knob to a certain position. If you set the knob to the center, it looks the same whether you do it once (and then just leave it there) or do it over and over, repeatedly making sure that the knob is set to the center position. So how can we tell what the Etch A Sketch® is actually doing?

If we had a way to change the position of the dot (without using the rules), we could see that the rules are repeatedly executed. If they
In order to see that the rules are executed repeatedly, we need another (non-rule) way to change the position of the dot. If we could move the dot this way, we would see that continued execution of the rules forces it back to the "stuck knob" position described by the rules. Imagine that both rules say

```
return 0;
```

So each instruction follower will keep putting the dot right in the middle of the screen. This won't look like much until we move the dot. But if we get the dot to jump to the top of the screen, the next time that the instruction follower executes the rule it will still put the knob back into the position mentioned in the rule - - the center of the screen, in our example. We can see that the rules for our Etch A Sketch® are invoked repeatedly, rather than just run once, by interacting with them.

In fact, in our Etch A Sketch® application, you can move the dot using your mouse. By clicking the mouse at a particular point on the Etch A Sketch® screen, you move the dot there. Almost immediately, the instruction followers go back to their knob-rule-checking. This means that if you position the dot with the mouse, it may not stay there for very long. Fortunately, you can see where the dot has moved because every time that the instruction follower moves the dot, it leaves a (black) trail behind it. Indeed, we will use this feature to draw pictures. So, for example, you can make a line from the upper lefthand portion of the Etch A Sketch® to the center by using the rules

```
horizontal rule
return 0;
vertical rule
return 0;
```

and clicking the mouse in the upper lefthand portion of the window:

```
@@add pic
```

The mouse click moves the dot momentarily, then the execution of the rules brings the dot back to the center, leaving a trail. You -- the user -- provide a third independent control to this application. By interacting with the dot, you, too, affect its behavior.

Q. Combining these two observations -- leaving trails and "jumping" the dot around using the mouse -- can you figure out how to create an asterisk (a bunch of line segments intersecting in the center)?

ans. If you run the Etch A Sketch® with both rules set at return 0; the dot will return to the middle, no matter where you move it with the mouse:

```
horizontal rule
return 0;
vertical rule
return 0;
```

Now, with the mouse, click anywhere. The dot will jump there, then immediately (well, as soon as the horizontal and vertical rules are sampled) return to the center. This should draw a line from wherever you clicked to the center. Repeat this, clicking somewhere else. Eventually, you’ll have a lovely asterisk.

```
@@add a picture.
```
Actually, you can make an asterisk around any point in the screen, not just the center of the screen. Try setting the rules to:

```
horizontal rule  vertical rule
return maxPos/2;  return 0;
```

Then repeat the clicking around process. Now you should get an asterisk centered at \((\text{maxPos}/2, 0)\), i.e., centered vertically but in the right half of the screen.

Rule bodies as we have presented them here are really just very simple method bodies. The return statement supplies the return value of its containing method. In the rule form that we are using here, the enclosing method declaration and braces are omitted. We will see more of how to write and use methods in the next chapter.

By manipulating the behavior of each method, the role played by each, and how we and the system interact, we can generate a variety of different behaviors even with extremely simple code.

### Paying Attention to the World

So far, the rules that we have written return the same value, no matter where the dot starts out. This corresponds to a rule that drives the knob to a certain fixed position, regardless of where it starts out. When you don't move the dot with the mouse, the rule causes the dot to sit still. When you do use the mouse to move the dot, the rule causes the dot to jump back to the same place that it has been. In this section, we will see how our rules can respond to information that they are given.

Each time that an instruction follower goes to execute a rule, the name \(\text{pos}\) is been pre-defined for it to hold the current position of the dot (along the relevant dimension). So, if the dot is half-way between the left side of the screen and the center, \(\text{pos}\) will be \(-1/2 \times \text{maxPos}\) when the horizontal rule is invoked, while if the dot is all the way at the right side of the screen, \(\text{pos}\) will be \(\text{maxPos}\).

**Q.** Using this information, write a rule that causes the dot to stay where it is.

**ans.** "Where it is" is always pos -- pos is the current position of the dot when the rule is about to be executed. So, no matter where the dot is, we can make it stay there using the rule

```
return pos;
```

If the dot is at 36, pos will be 36 and this rule will return 36. If we click the mouse and move the dot to 78, pos will be 78 the next time that the rule is executed, and we will return pos, or 78. Since pos is always where the dot is, returning pos will keep the dot there.

Note that pos is defined anew each time that the rule is executed. It is, in effect, a parameter to the rule.

**Q.** What happens when the dot is moved, using the mouse?
ans. When the dot is moved, the rule still says "stay where you are." So each time you click the mouse, the rule adjusts the knob to keep the dot where you've put it.

Note that the horizontal rule and the vertical rule each have their own version of pos. So pos in the horizontal rule has nothing to do with pos in the vertical rule; each gets its own proper position.

Now, using the information that pos is where the dot is, we can cause the dot to move. Each time the knob is checked -- each time the rule is invoked -- the knob should turn just a little bit.

Q. What would such a rule look like?

ans. We could use a rule that says

```
return pos+1;
```

This rule checks where the dot is, then instead of setting the knob there, it moves the knob slightly. The next time the rule is executed, it will move the dot a little bit further over. This will continue to happen until the dot reaches the edge of the screen.

[Why not pos=pos+1? Pos is a local name that this piece of code has for the current position of the dot. It is NOT the "control" for the current position of the dot. When the instruction follower is about to execute the instructions, it creates a new dial -- called pos -- and sets it to the value that represents the current position of the dot. After this happens, there's no additional connection between the value of the pos dial and the position of the dot. Reading the pos dial tells you what the current position of the dot is. But changing the pos dial doesn't move the dot. Returning a value does.

What would happen, then, if we ran with the horizontal rule pos=pos+1? First, we'd get an error. Remember, a rule has to end with return double; So now consider pos=pos+1; return 0; This would be exactly the same as just return 0; i.e., it would drive the dot to the center. How about pos=pos+1; return pos; ? This works, but it isn't as "nice" as return pos+1; The reason it works is that it first modifies the pos dial to have the value we want to return (pos+1), then returns the value on the dial. There's really no reason to modify the dial; we can just return the value directly.]

Pos is a parameter -- a name whose value is defined before the rule (method) body begins to execute. We can create names (dials) of our own as well. For example, a much more long-winded way of writing the previous rule might be:

```java
double velocity = 1;   // how much the position changes by.
double newPos;         // what the new position will be.

newPos = pos + velocity;   // compute next position...
return newPos;             // ...and return it.
```

The names velocity and newPos here are new local variables we create. Their declarations last until the return statement. Each time that this set of instructions is executed, the declaration line `double velocity = 1;` is executed again, and a new dial called velocity is created. (Yes, that's a lot of wasted dials. Don't worry; Java has facilities to make sure they are recycled.) In a later section, we will see a different kind of name that persists from one rule execution to the next.
Q. What happens when the dot reaches the edge of the screen?

ans. At this point, pos will continue to be increased. But values greater than maxPos aren't allowed. (The application is written so that values greater than maxpos are treated just like maxPos, so the dot will sit at the edge of the screen.)

Q. How would you make the dot move in the other direction?

ans. With a rule that says

return pos-1;

This rule makes the knob turn a little bit in the opposite direction. Remember, returning a value is the way to move the dot.

Fancy Dot Tricks

Q. What would happen if you used the rules:

\[
\begin{align*}
\text{horizontal rule} & \quad \text{vertical rule} \\
return pos+1; & \quad return 0; \\
\end{align*}
\]

Assume that the dot starts in the center of the screen.

ans. You would get a horizontal line from the center of the screen to the right hand edge of the screen.

Q. How would this be different if the rules were

\[
\begin{align*}
\text{horizontal rule} & \quad \text{vertical rule} \\
return pos+1; & \quad return pos; \\
\end{align*}
\]

ans. If the dot starts in the center of the screen and you don't click the mouse, you wouldn't be able to tell whether the vertical rule said return 0; or return pos; The value of pos (for the vertical rule) would start out as 0, and since nothing changes it, it would remain 0. The only way to see a difference is to move the dot (using the mouse). If you let the dot move across the screen until it's halfway to the right edge, then click the mouse in the lower left (at the X), here's what you'll see:

@@add picture

Q. What does the dot do if you start in the lower left hand corner of the screen and use the rules

\[
\begin{align*}
\text{horizontal rule} & \quad \text{vertical rule} \\
return pos+1; & \quad return pos+1; \\
\end{align*}
\]
ans. This rule pair would draw a diagonal line from the lower left hand corner of the screen towards the upper right hand corner. [Footnote: Actually, the line would only go towards the corner if the screen were relatively square. This diagonal line has a slope of 1.]

Q. Can you make the dot move from the maxPos edge of the screen to the other edge when it gets there?

ans. In order to do this, we need to check whether we've gotten to the maxPos edge. We can do this using an if statement:

```java
if (pos < maxPos)
{
    return pos + 1;
}
else
{
    return -maxPos;
}
```

Q. Can you make the dot move more quickly across the window?

ans. In the previous rules, we've increased pos by 1 each time. If we increase pos by a larger number, it will move more quickly. In fact, this increase to pos is the velocity -- the speed -- of the dot.

We can use this rule to create a sort of barber-shop pole effect -- a slowly climbing spiral around the window. To do this, we use a horizontal wrap-around rule and a vertical wrap-around rule. By setting the horizontal rule to move more quickly than the vertical rule, we get a line with a gradual slope. Since we're using wrap-around rules, the line repeats over and over again as it moves up the screen.

So starting in the lower left hand corner of the screen and executing

```java
if (pos < maxPos)
{
    return pos + 5;
}
else
{
    return -maxPos;
}
```

produces something like:

`@`@add pic. a sequence would be better.

For each Etch A Sketch® rule, pos is a name whose value is fresh each time the rule is executed. There is no connection between the value of pos from one invocation of the horizontal rule to the next. The value of pos for the horizontal rule is unrelated to the value of pos for the vertical rule. This behavior is essentially the behavior of a parameter to a Java method. In the next section, we will see a different kind of name.
Remembering State

In the previous section, we saw how to prevent the dot from getting stuck at one edge of the screen by jumping it to the other edge. It might have been nice to have the dot bounce back from the edge -- turning to move in the opposite direction -- instead. It turns out that that behavior requires an additional idea and a corresponding bit of machinery.

Suppose that we wanted to get the dot to turn around. We might start with a rule that looks like the "jump to the other edge rule, trying to detect when we've bumped into the $maxPos$ edge:

```java
if (pos < maxPos)
{
    return pos + 1;
}
```

This rule seems reasonable enough. It will cause the dot to move along until it reaches $maxPos$. But what then? When we reach $maxPos$, the if test will fail and we'll drop through to the else clause. It goes through $maxPos - 2, maxPos - 1, maxPos$. Now, it needs to go to $maxPos - 1, (and then to maxPos - 2, maxPos - 3$ and so on). So we might try

```java
else
{
    return pos - 1;
}
```

Sure enough, the dot's positions will be $maxPos - 2, maxPos - 1, maxPos, maxPos - 1$. But then what? The problem is that when the dot is at $maxPos - 1$ and this rule is executed again, the if test will succeed! The next position of the dot will be $((maxPos - 1) + 1), or maxPos! Then the if test will fail, triggering the else clause: $maxPos - 1$. At this point, the dot will oscillate between $maxPos - 1$ and $maxPos$ forever.

What went wrong? As always, our errors are informative. The problem is that the condition we're testing -- whether our position is $< maxPos$ -- doesn't really tell us what we need to know -- which direction to move in. Our position might be $maxPos - 1$ because we're heading towards $maxPos$, or it might be $maxPos - 1$ because we're heading back towards $maxPos - 2$. The if test doesn't give us any way to tell the difference. In fact, nothing about the current rule execution or our current position can answer this question for us. Instead, we need to know something about the previous execution, or about where we've been.

Fields

At this point, we need to introduce some new machinery. In our application, there is a special box (for each dimension) where we can enter names that persist from one execution of a rule to the next. These names correspond to fields of instance objects. They are like airport lockers, places that you can leave things when you're executing the rule and find them the next time you come back into town.

There are several different ways we can use airport lockers to solve this problem. The simplest is probably just to remember which direction we're going in. We can do this using a boolean name. In this case, we'll call the boolean increasing. We start with increasing true. So the declaration should say:
fields

boolean increasing = true;

(Recall that a declaration follows the type-of-thing name-of-thing rule. So this declaration says we have a boolean -- a true-or-false kind of dial -- that is called increasing. Because this is a definition, not just a declaration, it also sets the dial to read true.)

Now, we can write an if statement that says what to do if we're increasing: increase pos, unless we've hit the edge.

```java
if (increasing)
{
    if (pos < maxPos)
    {
        //keep going higher -- return the next position
        return pos + 1;
    }
    else
    {
        //we've hit the edge -- turn around
        increasing = false;
        return pos;
    }
}
else
{
    //not (increasing)
    if (pos > - maxPos)
    {
        //keep going lower -- return the next position
        return pos - 1;
    }
    else
    {
        //we've hit the edge -- turn around
        increasing = true;
        return pos;
    }
}
```

The else condition is similar, but with the signs reversed:

```java
else
{
    //not (increasing)
    if (pos > - maxPos)
    {
        //keep going lower -- return the next position
        return pos - 1;
    }
    else
    {
        //we've hit the edge -- turn around
        increasing = true;
        return pos;
    }
}
```

Q. Can you write a similar rule that relies on a numeric piece of state -- double previouspos -- instead? What does the declaration of persistent state look like? What happens the first time the rule is executed?

ans. First, declare a field (in the special box):

```
fields
```
double previousPos;

Note that the code starts by checking the boundary cases. If we're at the edge, we need to go inwards. Otherwise, we remember where we were this time (previousPos=pos;) and return a number that continues moving the dot in the appropriate direction.

```
if ( pos >= maxPos )
{
    //we're at the higher edge.
    previousPos = maxPos;
    return maxPos - 1;
}
else if ( pos <= -maxPos )
{
    //we're at the lower edge.
    previousPos = -maxPos;
    return maxPos + 1;
}
else if (pos > previousPos)
{
    //we're moving up: return a higher number.
    previousPos = pos;
    return pos + 1;
}
else // (pos <= previousPos)
{
    //we're moving down; return a lower number.
    previousPos = pos;
    return pos - 1;
}
```

**Fields vs. Variables**

Consider the previous example: using previousPos to keep track of which direction we were going. Why do we need previousPos here? Why can't we just use pos? There are two reasons. First, pos is already being used for something -- the current position of the dot. But the other reason is that pos gets a new value each time this set of instructions is executed. (Actually, pos gets re-created each time this set of instructions is executed.) So, if we put something we want to remember into pos, it won't be there the next time that these instructions are executed. We need to create a special value -- a field -- to hold things that we want to remember from one execution of these instructions to the next.

Field declarations must be made in the special box. Declarations in the regular code box are allowed, but they do not carry over from one execution to the next. Instead, a name declared in running code is a temporary scratch space. The corresponding dial or label is created each time that the declaration is executed (as a part of following those instructions) and discarded when the return statement is reached. Such local scratch space is called a local variable.

Contrast this with the use of velocity and newPos in an earlier section:
double velocity = 1;   // how much the position changes by.
double newPos;         // what the new position will be.

newPos = pos + velocity;   // compute next position...
return newPos;             // ...and return it.

Velocity and newPos here are local variables. They are not fields. That is, they are new dials that are created each time the rule is executed -- local scratch space that only exists during a single rule execution -- and they go away when the rule execution is done. Next time the rule is executed, they will be recreated.
In contrast, a field -- like previousPos in the rule above -- sticks around from one rule execution to another.

Summary

In this chapter, we have seen simple pieces of Java code that produce behavior. Each short set of instructions is in effect the body of a Java method; a value is returned at the end. The behavior of the system as a whole depends on the particular methods written. In addition, system behavior depends on how those rules are coupled together and how you as a user interact with them.

There are three different kinds of names that can be used in your code. First, you can use names that have been pre-defined for you, like pos. These are called parameters. In other chapters, we will see that in a Java method, all parameter names are included in the method declaration.

There are also two kinds of names that are declared using standard declarations or definitions. One kind is a temporary name that can be used during a single application of your rule. These names can be declared anywhere in your code. They are called variables. newPosition and velocity are examples of variables. Variables can be declared inside a method.

The last kind of name sticks around from one use of your rule to another. These names must be declared in a special box, separate from your rule code, but can be used freely in your rule code. These names are called fields. In this chapter, increasing is an example of a field. In the Etch A Sketch®, fields are declared in a separate box. In Java code generally, fields are declared outside of methods (but within an enclosing class).

Suggested Problems

See the text for things marked with a Q. Also:

1. Implement constant acceleration. Velocity is the change in position over time. For example, the rule return pos + 1; has a velocity of 1, while the rule return pos - 5; has a velocity of 5 in the opposite direction. Acceleration is the change in velocity over time. For example, if we return pos+1; when the rule is executed the first time and then the next time we return pos+3; when the rule is executed the second time, the change in velocity (i.e., the acceleration) is 2. To implement a constant acceleration, you need to
change the velocity by the same amount each time. This means that the rule can't return pos+a constant; instead, it has to return pos + an amount that changes each time. (Hint: Use a field.)

2. Can you make the dot go in a parabolic path? (Hint: what accelerations does it need?)

3. We have given you a parameter named \texttt{otherPos}. Each time that a rule is followed, otherPos begins with the position of the dot along the \textit{other} axis. Using this information, implement a function plotter. Write the code to plot the following:

- $y = x^2$
- $y = \sin(x)$
- $y = 1/x$

You may want to look at the \texttt{Math} library.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of \textit{Introduction to Interactive Programming In Java}, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:  
<w ebmaster@cs101.org>
Building New Things: Classes and Objects

Chapter Overview

- How do I group together related rules?
- How do I build a computational object?
- What are Java programs really made of?

In this chapter, you will learn to put together the pieces you've already seen -- things, names, expressions, statements, rules, and interfaces -- to create computational objects that can populate your communities.

In order to create an individual object, you first have to describe what kind of object it is. This includes specifying what things you can do with it -- as in its interface(s) -- but also how it will actually work. This description of the "kind of object" is like building a recipe for the object, but not like the object itself. (You can't eat the recipe for chocolate chip cookies.) These object-recipes are called classes.

For each thing that your object can do, your class needs to give a rule-recipe. This is called a method. Your objects may also have (named) pieces. These are called fields, and they are special java names that are always a part of any object made from this recipe.

When you actually use your class (recipe) to create a new object, there may be things that you need to do to get it started off right. These startup instructions are called a constructor.

When you are building an object, you are bound by the interfaces it promises to meet. If the interface promises a behavior, you have to provide a rule (method) body for the object to use.

This chapter is supplemented by reference charts on the syntax and semantics of Java classes, methods, and fields. It includes style sidebars on good documentation practice.

Most of the syntax of this section is covered in the appendix Java Charts.

Objectives of this Chapter

1. To recognize the difference between classes and their instances.
2. To be able to read a class definition and project the behavior of its instances.
3. To be able to define a class, including its fields, methods, and constructors.

Classes are Object Factories
In a previous chapter, we saw how to build an interface, or specification, that described the contract a particular kind of object would fulfill. We also saw that an interface does not provide enough information to actually create an object of the appropriate kind. Interfaces do not say anything about how methods actually work. They do not talk about the information that an object needs to keep track of. And they do not say anything about the special things that need to happen when a new object is created.

In this chapter, we will learn how to create objects and how to describe the ways in which they work. The mechanism that Java provides for doing this is called a **class**. Like an interface, a class says something about what kind of thing an object is. Like an interface, a class defines a Java type. However, interfaces specify only contracts; classes also specify implementation. Class methods are full-fledged rules, with bodies telling how to accomplish the task of that rule (not just the rule specification, or method signature, of abstract interface methods). Classes also talk about data -- information to be kept track of by objects -- as well as methods, or behavior. And a special part of a class -- the constructor -- talks about how to go about creating an object of the type specified by that class.

**Classes and Instances**

Objects created from a class are called **instances** of that class. For example, the class `CheckBox` refers to the instructions for creating and manipulating a GUI widget that displays a selectable checkbox on your computer screen. `CheckBox` is the name of the class, i.e., of the instructions. Let's say we create two particular checkboxes:

```java
CheckBox yesCheckBox = new CheckBox();
CheckBox noCheckBox = new CheckBox();
```

![CheckBoxes](Figure 1. The actual CheckBoxes.)

The two objects labeled by the names `yesCheckBox` and `noCheckBox` are instances of the class `CheckBox`. That is, they are particular `CheckBoxes`. The instructions for how to create -- or be -- a `CheckBox`, on the other hand, aren't a `CheckBox` at all; the instructions are `instructions`, or a class. In fact, the instructions are an object, too, though a very different kind of object and not one as obviously useful as a `CheckBox` or a `Timer` or a `Counter`. The kind of object the instructions are is called a **Class**.

Because the class contains the instructions for how to make a new instance and for how to behave like an instance of that class, we sometimes say that a class is like a **factory** where instances are made. Both a factory and its product are objects, but factories and the widgets that they make are very different kinds of objects. The factory has all of the know-how about its instances. But the factory isn't one of its instances, just as the **class** `CheckBox` isn't a `CheckBox`. It's a factory!
Recipes Don't Taste Good

Another analogy for a class (as opposed to its instances) is that the class is like a recipe for how to make instances. The instances are like food cooked from the recipe (say, chocolate chip cookies). It isn't hard to tell the difference between these things. The cookies smell good. If you are hungry, the note-card with the recipe on it won't be very satisfying. (It probably tastes a lot like cardboard.) On the other hand, if you're going over to Grandma's to cook, you might want to take the recipe but you probably don't want to stick the chocolate chip cookie in your back pocket. Classes actually contain a lot of information other than just how to make an instance. (The recipe might, too. It might include information on how long it takes to make the cookies, whether they need to be refrigerated, how long it will take before they go stale, or even how many calories they contain.)

Figure 2. Two recipes (classes) and two platefuls of cookies (instances) made from the second recipe.

Classes are Types

Like interfaces, classes represent particular kinds of objects, or types. Once a class has been defined (see below), its name can be used to declare variables that hold objects of that type. So an instance of a class can be labeled using a name whose declared type is that class. For example, the CheckBoxes described above are labeled using names (yesCheckBox and noCheckBox) whose declared type is CheckBox. Note that the class CheckBox -- the CheckBox recipe -- can't be labeled using a name whose declared type is CheckBox. The type of the class CheckBox is Class, not CheckBox. (This is the recipe vs. cookie distinction again.)

If an object is an instance of a class -- such as yesCheckBox and the class CheckBox -- then the type membership expression (yesCheckBox instanceof CheckBox) has the value true. Of course, CheckBox instanceof CheckBox is false (since the class isn't a CheckBox), but CheckBox instanceof Class is true.

Style Sidebar

Class Declaration

It is conventional to declare the members of a class in the following order:

- static final fields (i.e., constants)
- static non-final fields
non-static fields  
constructors  
methods

This order is not necessary -- any class member can refer to any other class member, even if it is declared later -- but it makes your code easier to read and understand.

All non-private members of the class should be listed in the class's documentation.

Class Declaration

A class definition starts out looking just like an interface declaration, although it says that it is a class rather than an interface:

```java
class Cat {
    ....
}
```

A class definition tells you what type of thing it is -- a class -- what it is called -- its name -- and what it's made of -- its definition, between braces. This last part is called the class's body. The body of the class definition contains all of the information about how instances of that class behave. It also gives instructions on how to create instances of the class. These elements -- fields, methods, and constructors -- are called the class's members. [Footnote: Be careful not to confuse members, which are parts of the class, with instances, which are objects made from the class. If chocolate chip cookies are instances of the cookie class (recipe), the chocolate chips are members of the class.] Each member is declared inside the body of the class, but not inside any other structure within the class. Another way of saying this is that each member is declared at top level within the class. So members are all and only those things declared at top level within a class.

For example, each instance of Java's Rectangle class has a set of four coordinates describing the rectangle's position and extent, as well as methods including one which tells whether a particular x, y pair is inside the Rectangle.

```java
...class Rectangle {

    ...
    int height;
    int width;
    int x;
    int y;
    ...
    ...inside(...)

}
```

In this case, height, width, x, y, and inside are all members of the Rectangle class.

Members and instances are quite different:
● members are parts of a class
● instances are things created from the class.

We will return to each of the elements of this declaration later in this chapter.

Classes and Interfaces

A class may **implement** one or more interfaces. This means that the class subscribes to the promises made by those interfaces. Since an interface promises certain methods, a class implementing that interface will need to provide the methods specified by the interface. The methods of an interface are abstract -- they have no bodies. Generally, a class implementing an interface will not only match the method specifications of the interface, it will also provide bodies -- implementations -- for its methods.

For example, a **ScoreCounter** class might meet the contract specified by the **Counting** interface:

```java
interface Counting {
    abstract void increment();
    abstract int getValue();
}
```

So might a **Stopwatch**, although it might have a totally different internal representation. Both would have **increment()** and **getValue()** methods, but the bodies of these methods might look quite different. For example, a ScoreCounter for a basketball game might implement increment() so that it counts by 2 points each time, while a Stopwatch might call its own increment() method even if no one else does.

A class that implements a particular interface must declare this explicitly:

```java
class ScoreCounter implements Counting {
    ....
}
```

If a class implements an interface, an instance of that class can also be treated as though its type were that interface. For example, it can be labeled with a name whose declared type is that interface. For example, an instance of class **ScoreCounter** can be labeled with a name of type **Counting**. It will also answer true when asked whether it's an instance of that interface type: if **myScoreCounter** is a **ScoreCounter**, then **myScoreCounter instanceof Counting** is true. Similarly, you can pass or return a **ScoreCounter** whenever a **Counting** is required by a method signature.

The generality of interfaces and the inclusion of multiple implementations within a single (interface) type is an extremely powerful feature. For example, you can use a name of type **Counting** to label either an instance of **ScoreCounter** or an instance of **Stopwatch** (and use its **increment()** and **getValue()** methods) without even knowing which one you've got. This is the power of interfaces!

Data Members, or Fields

The **Rectangle** class, above, had certain things that were a part of each of its instances: **width**, **height**, etc. This is because part of what it is to be a **Rectangle** involves having these properties. A
Rectangle-factory (or Rectangle-recipe) needs to include these things. Of course, each Rectangle made from this class will have its own width, height, etc. -- it wouldn't do for every Rectangle to have the same width!

Many objects have properties such as these: information called state or data that each instance of a class needs to keep track of. This kind of information is stored in parts of the object called fields. A field is simply a name that is a part of an object. For the most common kind of field, each instance of a class is born with its own copy of the field -- its own label or shoebox, depending on the type of name the field is.

Declaring a field looks just like an ordinary name declaration or definition (depending on whether the field is explicitly initialized). Such a declaration is a field declaration if it takes place at top level in the class, i.e., if it is a class member. (A local variable declared inside a method body or other block is not at top level in the class.)

Consider the Rectangle class defined above and reproduced here:

```java
class Rectangle {
    int height;
    int width;
    int x;
    int y;
    ...
}
```

Each instance of this class will have four int-sized shoeboxes associated with it, corresponding to the height, width, horizontal and vertical coordinates of the Rectangle instance. These fields are declared at top level inside the class body.

These fields are declared here, but not initialized: none of these fields is explicitly assigned a value. Fields, unlike variables, are initialized by default. If you don't give a field a value explicitly, it will have a default value determined by its type. For example, int fields have a default value of 0. Contrast int local variables, which don't have a default value and cannot be used until they are initialized. For details on the default values for each type, see the sidebar on Default Initialization.

**Java Types and Default Initialization**

In Java, field names can be declared without assigning them an initial value. In this case, Java automatically provides the field with a default value. The value used by Java depends on the type of the field.

Fields with numeric types are initialized by default to the appropriate 0; that is, either 0 or 0.0 (using the appropriate number of bits).
Fields with type `char` default to the value of the character with ascii and unicode code 0 -- `\u000'. This character is sometimes called the **null character**, but should not be confused with the special Java value `null`, the non-pointer.

Fields with `boolean` type are by default assigned the value `false`.

Fields associated with reference types -- including `String` -- are by default not bound to any object, i.e., their default value is `null`.

If a declaration is combined with an assignment -- i.e., a definition -- the definition value is used and these default rules do not apply.

These rules apply to names of fields as well as to the components of arrays -- described in a later chapter. In contrast, local variables must be explicitly assigned values -- either in their declaration (definition) or in a subsequent assignment statement -- before they are used. There are also names called parameters, which appear in methods and `catch` expressions; they are initialized by their invoking expressions and are discussed in elsewhere in this book.

**Fields are not Variables**

The difference in default initialization is only one difference between fields and local variables. This section covers several other important differences after first reviewing some properties of local variables.

A local variable is a name declared inside a method body. The scope of a local variable -- the space within which its name has meaning -- is only the enclosing block. At most, this is the enclosing method, so the maximum lifetime of a variable name is as long as the method is running. Once the method exits, the variable goes away. (A similar variable will come into existence the next time the method is invoked, but any information stored in the variable during the previous method invocation is lost.)

**Hotel Rooms and Storage Rental**

Because a field is a part of an object, and because an object continues to exist even when you're not explicitly manipulating it, fields provide longer-term (persistent) storage. When you exit a block, any variables declared within that block are cleared away. If you reenter that block at some later point, when you execute the declaration statement, you will get a brand new variable. This is something like visiting a hotel room. If I visit Austin frequently, I may stay in similar (or even the same) hotel rooms on each trip. But even if I stay in the same hotel room on subsequent visits, I can't leave something for myself there. Every time that I check into the hotel, I get what is for all intents and purposes a brand new room.

Contrast this with a long-term storage rental. If I rent long-term storage space, I can leave something there on one visit and retrieve it the next time that I return. Even if I move, change my address, etc., I can go back to my storage locker and get back the things I left there. This is just like a field: the object and its fields continue to exist even
when your attention is (temporarily) elsewhere, i.e., even when none of the object's methods are being executed.

The storage locker story is actually somewhat more complex than that, and so is the field story. It might be useful for someone else to have a key to my storage locker, and it is possible for that person to go to Austin and change what's in the locker. So if I share this locker with someone else, what I leave there might not be what I find when I return. It is important to understand that this is still not the same as the hotel room. Between my visits, the hotel cleans out the room. If I leave something in my hotel room, it won't be there the next time I come back. Each time, my hotel room starts out "like new". In contrast, the contents of my storage locker might change, but that is because my locker partner might change it, not because I get a freshly cleaned locker each time that I visit.

The locker partner story corresponds closely to something that can happen with fields. It is possible for the value of a field to change between invocations of the owning object's methods, essentially through the same mechanism (sharing) as the storage locker. To minimize this (when it is not desired), fields are typically declared \texttt{private}. For more on this matter, see the discussion of \texttt{Public and Private} in the next chapter. We will return to the issue of shared state (e.g. when two or more people have access to the same airport locker) in the chapter on \texttt{Synchronization}.

\textbf{Whose Data Member is it?}

A second way in which fields differ from variables is that every field belongs to some object. For example, in the \texttt{Rectangle} code, there's no such thing as \texttt{width} in the abstract. Every \texttt{width} field belongs to some \textit{particular} \texttt{Rectangle} instance, i.e., some object made from the \texttt{Rectangle} class/factory/recipe.

Because a field belongs to an object, it isn't really appropriate to refer to it without saying \textit{whose} field you are referring to. Many times, this is easy: \texttt{myRectangle.width}, for example, if you happen to have a \texttt{Rectangle} named \texttt{myRectangle}. The syntax for a field access expression is (1) an object-identifying expression (often, but not always, a name associated with the object), followed by (2) a period, followed by (3) the name of the field. You can now use this as you would any other name:

\begin{verbatim}
myRectangle.width = myRectangle.width * 2;
\end{verbatim}

for example.

There is, however, a common case in which the answer to the question "whose field is it?" may be an object whose name you don't know. This occurs when you are in a class definition and you want to refer to the instance whose code you are now writing. (Since a class is the set of instructions for how to create an instance, it is common to say "the way to do this is to use my own \texttt{width} field,...")

In Java, the way to say "myself" is \texttt{this}. That is, \texttt{this} is a special name expression that is always bound to the current object, the object inside whose code the name \texttt{this} appears. That means that the way to say "my own \texttt{width} field,..." is \texttt{this.width}. (Note the period between \texttt{this} and \texttt{width} -- it is important!)

\textbf{Scoping of Fields}

http://www.cs101.org/ipij/objects.html
The final way in which fields differ from (local) variables is in their scoping. The scope of a name refers to the segment of code in which that name has meaning, i.e., is a legitimate shoebox or label. (If you refer to a name outside of its scope, your Java program will not compile because the compiler will not be able to figure out what you mean by that name.) A local variable only has scope from its declaration to the end of the enclosing block. (A method's parameter has scope throughout the body of that method.)

A field name has scope anywhere within the enclosing class body. That means that you can use the field name in any other field definition, method body, or constructor body throughout the class, including the part of the class body that is textually prior to the field declaration! For example, the following is legal, if lousy, Java code:

```java
class Square{
    int height = this.width;
    int width = 100;
    ...
}
```

(This isn't very good code because (a) it's convoluted and (b) it doesn't do what you think it does. Although `this.width` is a legal expression at the point where it's used, the value of `this.width` is not yet set to 100. The result of this code is to set `height` to 0 and `width` to 100. The rule is: all fields come into existence simultaneously, but their initialization is done in the order they appear in the class definition text.)

A cleaner version of this code would say

```java
class Square{
    int height = 100;
    int width = this.height;
    ...
}
```

### Comparison of Kinds of Names

<table>
<thead>
<tr>
<th>Class or Interface Name</th>
<th>Field (Data Member)</th>
<th>Parameter</th>
<th>(Local) Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everywhere within containing program or package.</td>
<td>Everywhere within containing class.</td>
<td>Everywhere within method body.</td>
<td>From declaration to end of enclosing block.</td>
</tr>
<tr>
<td>Lifetime</td>
<td>Lifetime of object whose field it is.</td>
<td>Until method invocation completes.</td>
<td>Until enclosing block exits.</td>
</tr>
<tr>
<td></td>
<td>Label names: null</td>
<td>Value of matching argument expression supplied to method invocation.</td>
<td>Illegal to use without explicit initialization.</td>
</tr>
</tbody>
</table>
Shoebox names:
value depends on
type.

[Footnote: The column for Class or Interface Name refers only to top-level (non-inner) classes or interfaces. The scope and lifetime of an inner class is determined by the context of its declaration.]

Static Members

So far, we've said that fields belong to instances made from classes and that each instance made from the class gets its own copy. Recall that the class itself is an object, albeit a fairly different kind of object. (The class is like a factory or a recipe; it is an instance of the class called `class`.) Sometimes, it is useful for the class object itself to have a field. For example, this field could keep track of how many instances of the class had been created. Every time a new instance was made, this field would be incremented. Such a field would certainly be a property of the class (i.e., of the factory), not of any particular instance of that class.

The declaration for a class object field looks almost like an instance field. The only difference is that class field declarations are preceded by the keyword `static`. [Footnote: The choice of the keyword `static`, while understandable in a historic context, strikes us as an unfortunate one as the common associations with the term don't really accord with its usage here. In Java, `static` means "belonging to the class object." ] For example:

```java
class Widget {
    static int numInstances;
    ...
}
```

In this case, individual `Widget` do not have `numInstances` fields. There is only one `numInstances` field, and it belongs to the factory, not the `Widgets`. To access it, you would say `Widget.numInstances`. In this case, `this.numInstances` is not legal code anywhere within the `Widget` class.

Style Sidebar

Field Documentation

In documenting a field, you need to indicate what that field represents conceptually to the object of which it is a part. In addition, you should answer these questions as appropriate:

- What range of values can this field take on?
- What other values are interdependent with this one? For example, must this field's value always be updated in concert with another field, or must its value remain somehow consistent with another field?
- Are there any "special" values of this field that carry hidden meaning?
Interactive Programming In Java

- What methods (or constructors) modify this field? Which read this field? What else relies on its value?
- Where does the value of this field come from?
- Can the value of this field change?

Methods

In a previous chapter, we saw how method signatures describe the name, parameters, and return type of a method. A method signature declared in an interface ends in a semi-colon; this method specifies a contract, but doesn't say anything about how it works. It is essentially a rule specification. This kind of method -- a specification without an implementation -- is called abstract.

Classes specify more than just a contract. Classes also specify how their instances work. In order for an instance to do be able to do something, its class must give more than the rule specification for its methods. An instance needs the rule body for its methods. Classes must supply bodies for any methods promised by the interfaces that they implement. They may also supply additional methods with their own signatures and bodies.

Methods can be identified by the fact that a method name is always followed by an open parenthesis. (There may then be some arguments or parameters, on which more below; there will always be a matching close parenthesis as well.)

Method Declaration

A method definition also follows the type-of-thing name-of-thing convention, but the type-of-thing is the type that is returned when the method is called. So, for example, the inside method in the definition of Rectangle, above, returns a boolean value:

```java
...boolean inside( int x, int y) { ... }
```

Inside the parentheses is the list of parameters to the method: calling pictureFrame.inside on a particular x and y value returns true or false depending on whether the point (x, y) is inside pictureFrame. (Remember that the inside method only exists with reference to a particular Rectangle -- it's always some object's method!) The list of parameters, like every other declaration, follows the type-of-thing name-of-thing convention. Note, though, that while a regular variable definition can declare multiple names with a single type, in a parameter list each name needs its own type.

A few more notes on methods: If there are no parameters, the method takes no arguments, but it must still be declared and invoked with parentheses: pictureFrame.isEmpty(), for example. If there is no return value, the return type of the method is void. Finally, inside the body of the method, the parameters may be referred to by the names they're given in the parameter declaration. It doesn't matter what other names they might have had outside of the method body, or what else those parameter names might refer to outside the method body. We'll return to the issue of scoping later.
Recall from previous chapters that the method definition as we've described it so far -- the return type and the parameter list -- is also called the signature of the method. It tells you what types of arguments need to be supplied when the method is called -- it must be possible to assign a value of the argument type to a variable of the parameter type -- and what type of thing will be returned when the method is invoked. It doesn't tell you much about the relationships between the method's inputs and its outputs, though. (The method's documentation ought to do that!)

**Method Implementation Documentation**

Documentation for methods in classes is much like the documentation for methods in interfaces. However, class/object methods have bodies as well as signatures. In addition to the usual documentation of the method signature (see the Style Sidebar on Method Documentation in the chapter on Interfaces), your method documentation here should include:

- ways in which this method implementation differs from or specializes the documented interface method (signature).
- information concerning the design rationale (why the method works the way that it does), just as you would for any piece of Java code. For more detail, see the Style Sidebar on Documentation in the chapter on Statements.

**Method Body and Behavior**

This relationship -- how to get from the information supplied as arguments to the result, or return value -- is the "how to do it" part of the method. Its details are contained in the method body, which -- like a class body -- goes between a pair of braces. What goes in here can be variable definitions or method invocations or any of the complex statements that you will learn about later. You cannot, however, declare other methods inside the body of a method. Instead, the method body simply contains a sequence of instructions that describe how to get from its inputs (if any) to its output (if any), or what else should happen in between.

The body of a method is inside the scope of its parameters. That is, the parameter names may be used anywhere within the method to refer to the corresponding arguments supplied at method invocation time. The body of an instance method is also within the scope of the special name `this`. Just as in fields, inside a method the name `this` refers to the particular instance whose method this is. Static methods -- methods belonging to the class -- are not within the scope of `this`, though. That is, you can't use the special name expression `this` in a static method.

In order to return a value from a method, you use a special statement: `return`. There are actually two forms of this statement: `return(...)`, `return a value from a method invocation. For example,
return (total + 1);

returns one more than the value of \texttt{total}, though it doesn't change the value of \texttt{total} at all. The parentheses around the expression whose value is to be returned are in fact optional, leading to the second form of return: \texttt{return;} is used to exit from a method whose return type is \texttt{void}, i.e., that does not return anything.

Remember (from the chapter on Expressions) that a method invocation is an expression whose type is the return type of the method and whose value is the value returned by the method. You make this happen (when you're describing the method rule) by using an explicit \texttt{return} statement in a method's body. In the chapter on Statements, we saw the execution rule for a method body and how it relates to the evaluation rule for method invocation. This process is summarized in the sidebar on Method Invocation and Execution.

### A Method ALWAYS Belongs to an Object

A method is a thing that can be \textit{done} (or \textit{invoked}, or \textit{called}). For example, a painting program can draw a line, so \texttt{drawLine} could be the name of a method. \textit{Every method belongs to a particular object}. For instance, each \texttt{increment} method belongs to a particular \texttt{ScoreCounter} (or \texttt{Stopwatch}, or...) object; there is no such thing as an independent \texttt{getValue} method. So, if \texttt{myScoreCounter} refers to a particular \texttt{ScoreCounter}, \texttt{myScoreCounter.getValue()} invokes \texttt{myScoreCounter}'s \texttt{int}-returning method. You can't just call \texttt{getValue()}. Whose \texttt{getValue()} method is it, anyway?

Each time that you refer to a method, you should ask yourself whose method it is. You can invoke a method by first referring to the object, then typing a period, then the method name, as in \texttt{myScoreCounter.getValue()}. Sometimes, the answer to "whose method is it?" will be "my own", that is, the method belongs to the object whose code is being executed. As with fields, the way to say "myself" is with the special name expression \texttt{this}, so the way to say "my \texttt{getValue()} method" is \texttt{this.getValue()}. (Note the period between \texttt{this} and \texttt{getValue()} -- it is important!)

Generally, methods belong to instances of the class in which they're defined. Occasionally, though, it may be useful to have a method that belongs to the class itself. This corresponds to a property of the factory (or recipe), rather than one belonging to the widgets (or cookies) produced. For example, a method that prints out the number of widgets produced by the factory so far would be a method belonging to the factory, not one belonging to any particular widget. Methods that belong to the class instead of to its instances look just like regular methods, except that they are prefaced with the keyword \texttt{static}. (This name is pretty unintuitive, though it makes some sense in its historical context. Remember: In Java, \texttt{static} means "belonging to the class/factory/recipe itself, not to its instances.") A static method can be addressed by first citing the object it belongs to, then period, then the method name: \texttt{Widget.howManyWidgets()}. A static method \textit{should not} be invoked using \texttt{this}, though, because it doesn't belong to an instance.

Inside the method body, the name \texttt{this} may be treated as any other name.

it is also possible to refer to the object whose method it is as (for example, if you want to pass it as an argument to another method).

### Method Overloading
Just as in an interface, it is possible for a class to have multiple methods with the same name. This is called method overloading, since the name of the method is overloaded -- it actually refers to two or more distinct methods -- belonging to that object. In this case, each method must have a different footprint, i.e., the ordered list of parameter types must differ for two methods of the same object with the same name.

When an object has an overloaded method, the particular method to be invoked is selected by comparing the types of the arguments supplied with the footprints of the methods. The method whose footprints best matches the (declared) types of the arguments supplied is the one that is invoked. This matching is done using the same type inclusion rules as the operator instanceof.

Method Invocation and Execution

Method invocation is an expression; it is evaluated, producing a value. Within this expression, the body of the method is treated as a block (sequence) statement to be executed. This sidebar summarizes this process.

1. Before the method invocation expression can be evaluated, the object expression describing whose method it is must be evaluated. This object is called the method's target.
2. Based on this object and the (declared) types of the argument expressions, the method body is selected.
3. The argument expressions are evaluated and the method parameter names are bound to the corresponding arguments. If the target is an instance (i.e., if the method is not static), the name this is bound to the target as well.
4. Within the scope of these name bindings, the body of the statement is executed as a normal block except for special rules concerning return statements.
   - If, at any point within the execution of the body, a return statement is encountered, its expression (if present) is evaluated and then the entire method body and the scope of parameter names and this are exited upon completion of the return statement.
   - If the method has a return type other than void, the return statement is mandatory and must include an expression whose type is consistent with the return type. A suitable return statement must be encountered on any normal execution path through the method body. In this case, the value of the return expression is the value returned by the method invocation expression.
   - If the return type of the method is void, the final closing brace of the method body is treated as an implicit return; statement, i.e., a return with no expression. This has the effect of exiting the method body and special name scope.

Constructors

So, how do objects get created? Each class has a special member, called a constructor, which gives the instructions needed to create a new instance of the class. (If you don't give your class a constructor, Java automatically uses a default constructor, which roughly speaking "just creates the instance" -- details below. So some of the classes that you see may not appear to have constructors -- but they all do.)
**Constructor are Not Methods**

A constructor is sort-of like a method.

1. It has a (possibly empty) parameter list enclosed in parentheses.
2. It has a body, enclosed in braces, consisting of statements to be executed.
3. Inside the constructor body, `this` expressions can be used to refer to methods and fields of the individual instance under construction.

There are several differences.

1. The name of a constructor always matches the name of the class whose instances it constructs.
2. A constructor has no return type.
3. A constructor does not return anything; `return` statements are not permitted in constructors.
4. A constructor cannot be invoked directly.

Instead, a constructor is invoked as a part of a `new` expression. The result of evaluating this `new` expression is a new instance of the type whose constructor is evoked.

For example:

```java
class Pie {
    Pie (Ingredients stuff) {
        stuff.bake();
    }
}
```

In other words, to create a `Pie`, bake its ingredients. Note that `stuff` is a parameter, just like in a method. Constructor parameters work exactly like method parameters, and constructors take arguments to match these parameters in the same way that methods take parameters.

But you don't invoke a constructor in the same way that you invoke a method. In order to invoke a method, you need to know whose method it is. In order to use a constructor, you only need to know the name (and parameter type list) of the constructor. You invoke a constructor with a new expression as follows:

```java
new Pie (myIngredients)
```

where `myIngredients` is of type `Ingredients`.

**Syntax**

The syntax of a constructor is similar to, but not identical to, the syntax of a method. A constructor may begin with a visibility modifier (i.e., `public`, `protected`, or `private`) or one of a handful of other modifiers. Next comes the name of the constructor, which is always identical to the name of the class.
name is followed by a comma-separated parameter list enclosed in parentheses. This parameter list, like the parameter list of a method, consists of type-of-thing name-of-thing pairs. As in a method, the constructor name plus the ordered list of parameter types forms the constructor's footprint. It is possible for a class to have multiple constructors as long as they have distinct footprints.

After the parameter list, a constructor has a body enclosed in braces. This body is identical to a method body -- an arbitrary sequence of statements -- except that it may not contain a return statement. This is because constructors are not methods that can be called and that return values of specified types; instead, a constructor is invoked using a new expression whose value is a new instance of the constructor class's type. The constructor body may contain any other kind of expression or statement, however, including declarations or definitions of local variables.

```
modifiers ClassName ( type_1 name_1, ... type_n name_n )
{
   // body statements go here
}
```

For example, the NameDropper StringTransformer class might begin as follows. Note that the constructor argument is used to initialize the private field, the particular name that *this* NameDropper will drop.

```
public class NameDropper extends StringTransformer
{
   private String who;

   public NameDropper ( String name )
   {
      this.who = name;
   }

   //etc.
```

Note the use of a this. expression to refer to the field of the particular NameDropper instance being created.

This constructor could be invoked using the expression new NameDropper( "Jean" ) or new NameDropper( "Terry" ).

```
[Pictures of NameDroppers Jean and Terry]
```

**Style Sidebar**

**Constructor Documentation**

Although a constructor is not a method, documentation for a constructor is almost identical to documentation for a method. Constructor documentation should include:

- specifics distinguishing this constructor from others
- preconditions for using this constructor
- parameters required and their role(s)
- relationship of the constructed object to parameters or other factors
- side effects of the constructor
- additional assumptions and design rationale as appropriate

**Execution Sequence**

Before a constructor is invoked, the instance is actually created. In particular, any shoeboxes or labels declared as fields of the instance are created before the execution of any constructor code. This permits access to these fields from within the constructor body. In addition, any initialization of these fields -- through definitions in their declarations -- is executed at this time as well. Fields are each created and then each initialized in textual order, but all fields -- even those declared after the constructor[Footnote: There should be no such fields, declared after the constructor, because this makes your code difficult to read and so is bad style. However, if any such declarations are made, they still executed prior to the constructor itself.] -- are created and initialized prior to the execution of the constructor. Once each of the instance fields is created, execution of the constructor itself can begin.

When a constructor is executed, its parameters are matched with the arguments supplied in the invocation (new) expression. For example, in the body of the NameDropper constructor, the name name is identified with the particular String supplied to the constructor invocation expression. So if the constructor were invoked with the expression new NameDropper( "Terry" ), the name name would be associated with the String "Terry" during the execution of the body of the NameDropper constructor. When the statement

```java
this.who = name;
```

is executed, the value of the expression name is the String "Terry".

Once each of the parameter names has been associated with the corresponding argument, the execution of the statements constituting the constructor's body proceeds in order (except where that order is modified by control-flow expressions such as if or while). These statements may include local variable declarations; in this case, the name declared has scope from its declaration to the end of the enclosing block, just as in a method. When the end of the constructor is reached, execution of the constructor invocation expression is complete and the value -- the new instance -- is produced.

Because a constructor body may not contain a return statement, it is not possible to exit normally from any part of the constructor body except the end. Judicious use of conditionals can simulate this effect, however.

**Multiple Constructors and the Implicit No-Arg Constructor**

A class may have more than one constructor as long as each constructor has a different footprint, i.e., as long as they have different ordered lists of parameter types. So, for example, NameDropper might also have a variant constructor that took a descriptive phrase as well as name:
public NameDropper ( String name, String adjective )
{
    this.who = adjective + " " + name;
}

In this case, new NameDropper( "Marilyn Monroe" ) would create a NameDropper that started every phrase with "Marilyn Monroe says..." while

    new NameDropper( "Norma Jean", "My dear friend" )

(i.e., NameDropper(String, String)) would attribute everything to "My dear friend Norma Jean..."

If -- and only if -- a class contains no constructors at all, a default constructor is assumed present. This default constructor takes no arguments and does nothing beyond creating the object (and initializing the fields if they are defined in their declarations).

If there is even one constructor, the implicit no-arg constructor is not assumed. This means that if you define a constructor such as the one for NameDropper, above, that takes a parameter, the class will not have a no-arg constructor (unless you define one).

Beware: This can cause a problem when extending a class, if you're not careful. See chapter on Inheritance.

Constructor Functions

 Often, one of the main functions of a constructor is to initialize the state of the instance you're creating. Some initializations don't require a constructor; they can happen when the field is declared, by using a definition instead of a simple declaration:

class LightSwitch {
    
    boolean isOn = false;

}

In this case, each LightSwitch instance is created in the off position. In this kind of initialization, each instance of the class has its field created with the same initial value.

Contrast this with the following example, in which the initial value of the name field isn't known until the particular Student instance is created.

class Student {

    String name;

    Student( String who ) {
        this.name = who;
    }

}
In this case, a constructor is used to explicitly initialize the field named \textit{name}. When the initial value of a field varies from instance to instance, it cannot be assigned in the field declaration. Instead, it must be assigned at the time that the particular instance is created: in the constructor.

A constructor (or a method body) can also refer to properties of the class object itself. Recall the Widget class, which kept track of how many instances had been created. When the constructor is invoked, it can increment the appropriate field:

```java
class Widget {
    static int numInstances;
    static int howManyWidgets(){
        return Widget.numInstances;
    }
    Widget(){
        Widget.numInstances = Widget.numInstances + 1;
    }
}
```

Note that the constructor is not declared \texttt{static} (Constructors don't properly belong to any object) but that it refers to a \texttt{static} field. Note also that the \texttt{static} field is referred to using the class name (\texttt{Widget}), \textit{not} using \texttt{this}. We've also filled in the static method referred to above.

Finally, note that there is no explicit return statement in a constructor. A constructor is not a method, and it cannot be invoked directly. Instead, it is used in a construction expression, with the keyword \texttt{new}: \texttt{new Widget()} is an expression whose type is \texttt{Widget} and whose value is a brand new instance of the \texttt{Widget} class, for example.

\textbf{Q.} Construct a \texttt{Counter} class which supports an increment (increase-by-one) method. Where does the \texttt{Counter}'s initial value come from?

\textit{Style Sidebar}

\textbf{Capitalization Conventions}

By convention, the first letters of all class and interface names are capitalized. Since constructor names match their classes, constructor names also begin with capital letters. Java file names also match the class (or interface) declared within, so Java file names begin with a capital letter.

All other names (except constants) begin with lower case letters. In particular, the names of Java primitive types begin with lower case letters, as do fields, methods, variables, and parameters.

After the first letter, mixed case is used, with subsequent capital letters indicating the beginnings of intermediate words: e.g., \texttt{ClassName} and \texttt{instanceName}. 
The exception to the above conventions is the capitalization of constants (i.e., static final fields; see below). The names of constants are entirely capitalized. Intermediate words are separated using underscores (_): \texttt{CONSTANT\_NAME}.

Summary

- A Java class is a Java type.
- Each (public, top level) class must be defined in a separate file whose name matches the class name.
- An instance of a class is an object whose type is that class.
- If a class implements an interface, its instances must satisfy the interface's promises.
- Classes have methods, fields, and constructors.
- In a class, methods typically have bodies specifying how to carry out the method. (Otherwise, the method is \texttt{abstract}, and so is the class.)
- Every method belongs to some object. Unless declared \texttt{static}, a method belongs to (each of) a class's instances, not to the class itself.
- A field declares (and perhaps also defines) a name whose scope is the class body (i.e., any methods, fields, or constructors in the class body) and whose lifetime is the lifetime of the instance it belongs to.
- Every field belongs to some object. Unless declared \texttt{static}, a field belongs to (each of) a class's instances. Each instance has its own copy of the field, i.e., its own unique label with that field's name and type.
- In Java, \texttt{this} is a special name, bound in any non-static member, that refers to the instance whose instructions are being followed. An instance can refer to its own methods and fields by saying \texttt{this.methodName(...)} or \texttt{this.fieldName}, or to itself by the name expression \texttt{this}.
- A constructor gives instructions for how to create an instance of the class.
- The class itself is an object. (It is an instance of the class \texttt{Class}.) Fields and methods declared \texttt{static} belong to the class object itself and are properly referred to using \texttt{ClassName.methodName(...)} or \texttt{ClassName.fieldName}.

Exercises

1. Consider the following definition:

```java
public class MeeterGreeter
{

    private String greeterName;

    public MeeterGreeter( String name )
    {
        this.greeterName = name;
    }

```
public void sayHello()
{
    Console.println( "Hello, I'm " + this.greeterName );
}

public void sayHello( String toWhom )
{
    Console.println( "Hello, " + toWhom \\
        + ", I'm " + this.greeterName );
}

public String getNameWithIntroduction( String toWhom )
{
    // ****
    this.sayHello( toWhom );
    return this.greeterName;
}

Now assume that the following definition is executed:

MeeterGreeter pat = new MeeterGreeter( "Pat" ),
    terry = new MeeterGreeter( "Terry" );

1. What is printed by pat.sayHello()? What is returned? Which method is invoked?
2. What is printed by new MeeterGreeter( "Chris" ).sayHello( "Terry" )? What is returned? Which method is invoked?
3. What is printed by terry.sayHello( "Pat" )? What is returned? Which method is invoked?
4. Assume that the expression pat.getNameWithIntroduction( "Chris" ) is being evaluated. What would the value be of each of the following expressions if they were to appear on the ****'d line:
   i. toWhom
   2. this.greeterName
   3. name
   4. this.sayHello()
   5. new MeeterGreeter( "Pat" )
   6. this.getNameWithIntroduction( toWhom );

2. Now consider the following modification of the MeeterGreeter code. Assume that we add the field definition

    private static String greeting = "Hello";

We will want to make several other modifications to the MeeterGreeter code.

1. Write a changeGreeting method that allows a user to change the greeting string.

i. What arguments should this take?
2. What should it return?
3. What should its body say?
4. To which object should this method belong?

ii. Write an expression that invokes the changeGreeting method that you have written.

3. Next, modify the sayHello methods to replace the fixed string "Hello" with the a reference to the greeting field. Whose greeting field is it?

3. Define a class whose instances each have one method, rememberAndReturnPrevious, that takes a String and returns the String it was previously given. Supply the first return value through the instance creation expression. Give an example of your code in use.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Designing With Objects

Chapter Overview

- How do I design using objects and entities?

In the preceding chapters, we have seen how interfaces specify contracts and how classes implement them. We have used expressions and statements to create instructions that describe the processes of performing actions, making up method and constructor bodies. And we have used names to retain an object's state even while none of the object's methods is executing. In this chapter, we turn to the question of how we design systems using these various tools.

The first part of this chapter looks at one simple example to illustrate how the fields and methods of an object can be identified and implemented. Although the example is small, the principles described here are general and will be used in the design of any object-oriented program. This example also provides an opportunity to look briefly at the question of privacy, or how an object separates internal information from information that it makes available to other objects.

The next section of this chapter turns to look at three important kinds of objects that appear in many systems. These kinds of objects -- data repositories, resource libraries, and traditional objects -- each play distinctly different roles in any system, and their designs reflect these roles. A fourth distinct kind of object -- animate objects -- is the topic of the next chapter.

The chapter concludes with a discussion of the ways in which different objects and types are interrelated.

Objectives of this Chapter

- To become familiar with the identification of objects, methods, fields, interfaces, and classes from a problem description.
- To recognize common kinds of objects and the roles that they play.
- To learn to identify opportunities to use these patterns in designing systems.

Object Oriented Design

So far, you've seen a lot of Java how-to: how to declare, define, assign, and invoke variables of primitive and object types, classes, object instances, methods, and control flow. Now that you have some fluency with the basic building blocks of Java, it is time to start looking at why each of these constructs is used and
how they are combined to build powerful programs. In this chapter, we'll look at objects and classes; in the next, we'll continue this discussion by focusing on instruction-followers and self-animating objects.

Objects are Nouns

When you are constructing a computational system, you need to build pieces of code to play various roles in the system you're constructing. To a first approximation, you can do this by writing down a description in English of the system and the interactions you want to have with it (and that you want its parts to have with one another), then mapping these things onto elements of Java. When you do this, you will find that Java objects correspond roughly to the nouns of your description.

To be a bit more precise, Java objects are things in your computational world, but not all of the things are Java objects. Some of the things will have primitive types -- numbers, for example, will probably be doubles or ints -- but most of the things that are important enough to represent and complex enough that Java doesn't have a built-in type for them will be objects in your world. This means that you will have to define a Java class which describes what this type of object is (more below).

For example:

- A counter has a number associated with it. When it starts out, the number is 0. You can increment the counter, and each time you do so, the number goes up by one. At any time, the counter can also be asked to provide the current value of its associated number.

The nouns in this paragraph are counter and number. (And you, but we'll assume that you either refers to the user, which we don't need to implement, or to some other component outside of the current system.) The counter will be a Java object; we can use an int for the number since it isn't asked to do anything, just to be there.

Methods are Verbs

When you write down your description, you will also find that there are lots of things that these objects do to/with/for one another, or that you want to do to/with/for them. These things correspond to the verbs in your English description, and they are the methods of your Java objects. Every verb has a noun associated with it -- its subject -- and every Java method belongs to some object.

In our basic counter example, the verbs are increment (and its alternate form, goes up by one) and provide (as in "provide the current value"). Increment is something you need to be able to do to the counter object. We could handle provide in either of two ways: we could give the counter someone or something to provide the value to, or we could ask it. We will adopt the second of these options, though we will return to the first option in the chapter on Communication Patterns. This means that the counter object is going to have to have (at least) these methods.

Interfaces are Adjectives

Interfaces and classes are both types. How do you know when to use which one? As a general rule of thumb, names (including parameters, fields, and local variables) should generally be declared using an interface type whenever possible. Constructor expressions, of course, require a class type.
Interfaces are good at capturing commonality. It is almost always useful to define an interface corresponding to the set of features of your objects that would hold for any implementation of them. For example, the need for any counter to have an increment method and a getValue method makes these good properties to encapsulate in an interface. No matter how we implement the counter, these method properties will hold. In contrast, the fact that most counters will keep track of their value using a field (perhaps even an int field is an implementation-specific detail that cannot be expressed in an interface. An interface talks about what an object can do, not about how it accomplishes these tasks.

A Counting interface might say:

```java
interface Counting {
    void increment();
    int getValue();
}
```

Why would we use this? By referring to any actual counters by their interface type -- Counting -- rather than their implementation types, we make it possible for the implementor to modify details of the implementation -- or, even, to change which underlying implementation we're using -- without changing the code that uses it. We also avoid committing to any specific aspects of the implementation -- such as the representation of the current value through a long or a double or even a String -- that really shouldn't matter to the user of the class.

The name of this interface is only moderately adjectival, but most interfaces are named using adjectives. For example, we have seen Resettable and will soon see Animatable, Runnable, and Cloneable. We could almost call Counting Incrementable instead.

**Classes are Object Factories**

So if the nouns are objects, the verbs are methods, and the interfaces are adjectives, what is left for the classes? Java classes are kinds of objects. They correspond, roughly, to machines (or factories) that tell you (or Java) how to make new objects, not (necessarily) to anything explicitly in your English description.

For example, the class BasicCounter is something that tells Java how to make a new BasicCounter. It doesn't appear explicitly in the English description, but parts of the description are about it and other parts imply things about what it must say. The phrase "When it starts out, the number is 0" talks about initial conditions for BasicCounter objects; the class is the thing responsible for establishing these (since it is the factory where Counters are made).

For that matter, the class is responsible for establishing what the parts of an object are. "Parts" here refers to methods and fields. What are the pieces of a BasicCounter object? In this case, its number (and maybe an associated display). What are the things a BasicCounter can do (or that we can do with/to a BasicCounter)? increment and provide its value, at least. So the class BasicCounter will most likely include a number field (which is going to be of type int), as well as methods corresponding to incrementing and value-providing. It will also initialize the number field to 0.
Q. Is this a static or dynamic initialization? Where does it take place?

[Pic of counters]

Style Sidebar

Class and Member Documentation

This list summarizes many of the main features that good documentation will capture about classes and their members. For more detail, see the specific documentation sidebars in the previous chapter.

- methods
  - parameters: type and role
  - return value: type and role
  - function: why you'd do it
  - "side effects": what else it does (esp. values changed)
- fields
  - type and role
  - how it changes & which methods use/change it
  - constraints and interdependencies
- constructors
  - parameters: type and role
  - relation of parameters to the particular instance produced
  - "side effects": what else it does (esp. values changed)
- class
  - its interface, especially key methods and fields & how they interact

Some Counter Code

Here is a very basic implementation of the counter class:

class BasicCounter implements Counting {
    int currentValue = 0;

    void increment() {
       this.currentValue = this.currentValue + 1;
    }

    int getValue() {
      return this.currentValue;
    }
}
Some notes on this code:

- The class is a factory for making BasicCounters. Its body talks about what each individual BasicCounter looks like, not about the factory itself.
- Each individual BasicCounter has its own currentValue field. Each one starts out with the value 0, but they can change independently: each currentValue field belongs to a specific BasicCounter.
- We haven't included a constructor because, in this case, Java's default constructor does what we want. This is in general true when there is no dynamic initialization (each instance starts out in the same state).
- The increment and getValue methods are methods that belong to each BasicCounter instance. In each case, they refer to the currentValue field of that BasicCounter instance. We note this by using the java keyword this.

Someone wanting to use a BasicCounter could now do so by invoking an instance creation expression with this BasicCounter factory:

```java
new BasicCounter()
```

This expression is probably more useful if we embed it inside another expression or statement, e.g.,

```java
Counting myCounter = new BasicCounter();
```

Note the use of the interface type when declaring the name, but the class type within the construction expression.

Now we can ask myCounter to increment itself or to give us its value:

```java
myCounter.increment();
Console.println( myCounter.getValue() );
// prints 1
myCounter.increment();
myCounter.increment();
myCounter.increment();
myCounter.increment();
Console.println( myCounter.getValue() );
// prints 4
```

**Final**

A name in Java may be declared with the modifier **final**. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, constant.
The most common use of this feature is in declaring final fields. These are object properties that represent constant values. Often, these field are static as well as final, i.e., they belong to the type object rather than to its instances. Making a constant static as well as final makes it easy for other objects to refer to this value. It is appropriate for static final fields to be declared public and to be accessed directly by other objects. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared final. A final parameter is one whose value may not be changed during the execution of the method. A final variable is one whose value is unchanged during its scope, i.e., until the end of the enclosing block.[Footnote: Final fields and parameters are unnecessary unless you plan to use inner classes. They may, however, allow additional efficiencies for the compiler, and in any case they cannot be detrimental.]

Java methods may also be declared final. In this case, the method cannot be overridden in a subclass. Such methods can be inlined by the compiler, i.e., the compiler can make these methods execute more efficiently than other non-final methods. A static method is implicitly final. An abstract method may not be declared final.

Java classes declared final cannot be extended (or subclassed).

Public and Private

When we defined the BasicCounter class, we intended that the rest of the world would interact with its instances (things produced by the class BasicCounter factory) only through increment() and getValue(). But there is nothing about the code we've written that prevents someone from defining a BasicCounter name and then changing the value of that BasicCounter instance's currentValue field. For example, it would be perfectly possible for another object to say

```java
BasicCounter anotherCounter = new BasicCounter();
anotherCounter.currentValue = anotherCounter.currentValue + 1;
```

instead of

```java
anotherCounter.increment();
```

This would be rather rude of it (and very bad style), but it is technically possible and unfortunately done all of the time. Using the interface type -- Counting -- rather than the class type -- BasicCounter -- is one way to avoid this, and this is yet another reason why it is generally better to use the interface type. But as the implementor of BasicCounter, we can't require that it always be treated as a Counting instead of as a BasicCounter. Further, coercion (such as (BasicCounter) myCounter) will get you around the interface-associated name.[Footnote: Specifically, it would be legal, if longwinded, to say

```java
((BasicCounter) myCounter).currentValue
```
= ((BasicCounter) myCounter).currentValue + 1;

] Class designers don't always get to choose how users of the class will interact with it or as what type they'll choose to treat it.

We can take a stronger position on the matter of direct field access, though. We can, in fact, prevent direct field access by protecting the currentValue field of each BasicCounter instance. We do this by changing the declaration of the field in class BasicCounter:

```java
class BasicCounter {
    private int currentValue = 0;
    void increment ...
}
```

By making currentValue private to class BasicCounter, only the instance of BasicCounter itself can access the currentValue field. Now, this rudeness on the part of the calling object would simply be impossible. (The compiler would complain that the calling object could not access BasicCounter's private field currentValue.)

In general, it's a good idea to define fields as private when you don't want them to be accessed directly by other objects. You can also define private methods, which are generally things an object uses for its internal computations but not intended to be used from outside the object. Private things are a part of the class's or its instances' own internal representations and machinations; they are not to be shared.

Any member, not just a field or a method, can be private. You can even define private constructors. Although this may seem like an odd thing to do, it actually isn't all that strange. It means that the class object (along with any instances it creates) maintains complete control over whether and when new instances can be created. The class can refuse to create any instances, or it can create just one instance and return this any time someone asks for a new one (using a special method the class defines for this purpose, such as getInstance(), not the (private) constructor), or it can ask for the secret password before creating an instance if it (or its designer) wants to.

The opposite of private is public. You should declare things public when you want them to be accessible from any part of anyone's code. You can also declare classes and interfaces to be public, in which case they must be defined in a file whose name is the same as the name of the class or interface, plus .java.
If you don't declare something **private** or **public**, it is in an intermediate state. There are actually two intermediate states, **protected** and the default state. These two are in fact equivalent to one another and to **public** unless you use packages, a Java feature that we will explore in the chapter on Abstraction. Until then -- until you are building complex enough code that you need to subdivide it at finer levels than all-or-none -- you should use **public** and **private** all of the time, i.e., everything in your code should be one or the other.

### Kinds of Objects

Objects are the nouns of programming: the people, places, and things. Nouns do a lot of different things in the world and, similarly, objects do a lot of different things in programs. In this section, we take a closer look at several kinds of objects, their typical construction, and why you might use them. The objects discussed here are all relatively passive; they do nothing until asked. In the next chapter, we go on to look at active objects, objects that have their own instruction followers.

### Data Repositories

A **data repository** is a very simple object that exists solely to hold a set of interrelated data. The data repository object simply glues these things together, providing a convenient way to deal with the grouped data as a single unit.

One example of a data object might be a postal address. This might consist of a street address, a city or town, a state or province, a postal code, and a country. There isn't really much that you would do with an address, other than pull out the individual pieces or maybe modify one or more of the pieces. (For example, the postal service just changed my postal code, so although my address object stayed the same, its postal code field needed to change.) The whole address is useful and meaningful in a way that the pieces individually are not, so it is often convenient to be able to package these pieces together and to pass the address object around as a single unit.

Here is some code for a very simple address object. Note that this code has some aesthetic problems, which we will address shortly.

```java
public class OversimplifiedAddress {
    public String streetAddress,
            city,
            state,
            postalCode;
}

// Problems with this class:
// Non-final fields ought not to be public.
// Fields ought to be initialized by (missing) constructor or default.
```

Like instances of this OversimplifiedAddress class, data repository objects exist to hold a collection of pieces together. Typically, each of these pieces is represented by a field of the object. The simplest form of data repository object is one -- like an instance of the Oversimplified Address class -- that has a set of public fields and nothing else. However, this form is not recommended.
One object should never access another object's fields directly. [Footnote: Actually, this should read "One object should never access another object's non-final fields directly." Final fields are in effect constants; the reasons for objecting to field access do not apply to read-only accesses to a constant.] Instead, an object should provide methods for accessing its fields. [Teacher's note: Where getter methods are simply long-winded ways of doing field access, a good compiler should be able to inline this code. In Java, this can be done when the getter method is declared final.]

In our simple address object, we violated this rule. To fix that class definition, we should instead make each of these fields internal to the object. So that other objects can access these fields, we need to provide getter and setter methods to access them. A **getter** method is a method that returns the value of a field. A **setter** method is one that has a single parameter, the new (desired) value of the field; evaluating this method modifies the state of the object to reflect this new value. Getter methods are sometimes called **selectors** and setter methods are sometimes called **mutators**. It is common to use the name of the field prefixed with get as the name of the getter method and the name of the field prefixed with set as the name of the setter method.

Note that getter and setter methods need not correspond one to one with fields. Instead, a setter method may change the value of more than one field; a getter value may return an object that encapsulates more than one field value. Alternately, a getter or setter may make reference to an apparent field that doesn't actually exist *per se*.

We can improve the address class by modifying it to use getter and setter methods. Only one pair of these methods is shown here, although the complete class definition would presumably contain four pairs of getter and setter methods.

```java
public class BetterAddress {
    private String streetAddress, city, state, postalCode;
    ....
    public void setPostalCode( String code )
    {
        this.postalCode = code;
    }
    public String getPostalCode()
    {
        return this.postalCode;
    }
}
// Remaining problems with this class:
// Fields should be initialized by (absent) constructor or default.
```

Why shouldn't one object access the fields of another directly? (Why should you use getter and setter methods?)

1. Methods separate use from actual (internal) representation. The user of a class shouldn't need to know (or care) how information is actually represented inside the class. For example, US postal codes are commonly written as five-digit numbers. A different implementation of addresses intended
for use only in the US might actually represent the postalCode field using an int instead of a String. The getter and setter methods of this USAddress object could do the conversion for the user:

```java
public String getPostalCode()
{
    return new String( this.postalCode );
}
```

We might have an interface (say, GeneralizedAddress) containing (an abstract version of) this method. Both USAddress and BetterAddress classes could implement the GeneralizedAddress interface, even though they use different internal representations.

Another variant of separating use from actual representation involves getter and/or setter methods for fields that don't actually exist. For example, it might be useful for these address objects to have a getAddressLabel field, which would return the multiline String containing the complete address suitable for printing on an envelope. This getter method would automatically calculate the appropriate value from the individual fields of the address object; there is no actual field corresponding to the information that this getter field provides.

```java
public String getAddressLabel()
{
    return new String( this.streetAddress + "\n"
                    + this.city + ", "
                    + this.state + "  
                    + this.postalCode + "\n"
                    + this.country );
}
```

Getter and/or setter methods like this one, which do not correspond to any actual field of the object, are sometimes called virtual fields. To the user of the object, it looks as though there's a field there. Whether that field actually exists or just looks like it is nobody's business but the implementing object's.

2. Methods can provide additional behavior, including access control and error checking. For example, BetterAddress could be augmented with an internal list of the states or provinces within each country. If the setter method were given an argument that didn't match one of the appropriate values, it could report an error. The most extreme case of this is a read-only field, one in which no non-private setter method is supplied. This prevents a user of the object from ever modifying the value of that field.[Footnote: Note that a read only field is different from a constant (final) field. A read-only field can be changed by its owning object, but not by anyone else. A final field's value, once set, cannot be changed. This is enforced by the Java compiler.]

Another example of augmenting the behavior of a setter might involve automatically filling in the city and state whenever a postal code is entered. The postal code's setter method could look up the appropriate city and state information based on the postal code supplied and propagate this information to these other fields as well, saving the user the work of providing this information separately. (Some mail order companies do this now: you give them your postal code, and they tell you what city and state you live in!)

There are other reasons why methods, rather than fields, are a good idea. Some of these involve issues that
class exists so that its instances can hold both (horizontal and vertical) coordinates, e.g., of a window size. This allows them to be simultaneously returned from a method such as Window's getSize() method. If getSize() weren't able to return a data repository type such as Dimension, you'd first have to invoke a method that returned the Window's horizontal dimension, then one that returned its vertical dimension. If the Window's size changed in between these two method invocations, your two individual dimension components would combine to produce a nonsensical value!

Pure data repository objects are actually quite rare in good object-oriented design. This is because most objects do more than hold some state. The extensions we've described above, including propagation of changes, virtual fields, and access control already begin to expand the data repository idea. In the next subsection, we look at objects that exist to provide behavior without state. In the following subsection, we will return to objects that contain both data and more interesting behavior.

Resource Libraries

We have seen objects that hold together an interrelated set of data. Sometimes, an object exists to hold together an interrelated set of methods. If these methods are not tied to any particular state of the world, they may usefully be grouped together within a (generally non-instantiable) class that exists solely for this purpose. Consider, for example, the square root function. It is a useful function, and it is often convenient to have it lying around. But, in Java, any function must be a method belonging to a particular object. Java has a square root method; but whose method is it?

The answer to this question is that sqrt() belongs to a special class called Math. Math is a class that exists precisely so that you can use its methods, like sqrt(). Math is a canonical function library; it has no use beyond being the place to find its member functions. It exists to provide the answer to the question, "Whose method is sqrt()?"

Because Math is a place to find these functions, it is not a class of which you would want to make instances. Instead, Math has only static methods and static fields. This means that you can use its methods and data members through the class object (Math) itself. For example, a typical method is Math.sqrt(double d), which takes a double and returns a double that is the square root of its argument. Without the Math class to collect it and other mathematical functions, it is hard to imagine to whom this sqrt function could belong. Math exists so that there is a place to collect sqrt and a number of other abstract mathematical functions.

The Math class has static methods for the trigonometric functions, logarithms and exponentiation, various flavors of rounding, and very simple randomization. Math also has two (static final, i.e., constant) fields: E and PI, doubles representing the corresponding mathematical constants. See the sidebar on Math for details.

Q. Since it's not instantiable, why couldn't Math be an interface?

Math -- the class, with its static methods and fields -- is a very useful class. However, it wouldn't make sense to create any instances of it. In fact, Math has no publicly available constructor. This is a common way to prevent a class from being instantiated: give it only a private constructor. In general, a resource collection is the kind of object of which wouldn't have any use for multiple copies.

Another resource collection class is cs101.util.Console. Console -- documented in a sidebar in the chapter
Other classes that provide static collections of resources (whether functions or otherwise) include java.lang.System, cs101.util.MoreMath, and cs101.util.Coerce.

**class Math**

The built-in Java class Math may be the canonical resource library. It contains two (static) fields, Math.E and Math.PI, both doubles, corresponding to the mathematical constants e and pi, respectively.

Math also contains a host of useful mathematical functions, again all static. Each of the following methods takes a double as an argument and returns a double:

- cos: cosine of its argument
- sin: sine of its argument
- tan: tangent of its argument
- exp: Math.E raised to the power of its argument
- sqrt: square root of its argument
- floor: largest double corresponding to an integer value that is smaller than its argument
- ceil: smallest double corresponding to an integer value that is larger than its argument
- rint: double corresponding to the integer value nearest its argument
- acos: arc cosine of its argument
- asin: arc sine of its argument
- atan: arc tangent of its argument
- log: Logarithm base Math.E of its argument
- ceil: smallest double corresponding to an integer value that is larger than its argument
- floor: largest double corresponding to an integer value that is smaller than its argument
- rint: double corresponding to the integer value nearest its argument

Math.abs takes a double, a float, a long, or an int, and produces a value of the same type as its argument that is guaranteed to be non-negative.

Math.max and Math.min each take two arguments of the same type (both double, float, long, or int). max returns the larger of its arguments; min the smaller.

Math.round takes a double and returns the long closest in value to its argument.

Math.pow takes two doubles and yields the value of the first raised to the power of the second.

\[
\text{Math.pow( base, expt )} = \text{base}^{\text{expt}}.
\]

Math.random takes no arguments and returns a double equal to or larger than 0.0 and strictly smaller than 1.0.

There are a few other Math methods not included here. In addition, there are extra mathematical functions (including more flexible and powerful randomization) available in the package java.math. For these additional methods, see the Java API documentation on the Javasoft web site.

**Traditional Objects**

Some objects, like data repositories, exist primarily to bundle together certain pieces of data. Other objects exist primarily to hold stateless, general-purpose functional behavior. Most objects fall into neither of these categories. Instead, most objects represent things with both state -- what happens to be true of them Right
Now -- and behavior -- how that object can change over time. Some of these objects, like Windows, Buttons, and Menus, have visual manifestations. Other objects, like the ones that represent Strings or URLs, are more obviously internal to programs. Many of the objects that you create will be of this kind.

A String is an object that keeps track of the sequence of characters of which it is composed, so somewhere inside the String object must be data that corresponds to those characters. But a String is not simply a data repository; it has a diverse set of methods. What kinds of things might you want to do with a String? Certainly look at some of the characters, which you can do using the String's `charAt(int index)` method. Java's String class provides additional methods, though, which allow you to do more than simply look at parts of the String. For example, there is `toUpperCase()`, which returns a String just like the one whose method you invoke, but with all letters in upper case. (For example, "Hi there".toUpperCase() returns a String that would print out as "HI THERE".) String's `toUpperCase()` method is neither a selector nor a mutator. More complete descriptions of the String class and its methods are included in the sidebar on the String class in the first Interlude.

Another kind of traditional object that we've seen is a counter. This object has internal state (whatever the current count is set to) and methods providing access to this state (e.g., `increment()` and `getValue()`). The methods can't work without the state; the state isn't directly accessible, but provides the basis for method behavior. This is an extremely typical kind of object.

Here is some code implementing a slightly more sophisticated Counter class than the one described at the beginning of this chapter. In addition to the functionality provided by that BasicCounter class, this class implements the `Resettable` interface, i.e., provides a `reset()` method.

```java
public class Counter implements Counting, Resettable {
    private int currentValue;

    public Counter() {
        this.reset();
    }

    public void increment() {
        this.currentValue = this.currentValue + 1;
    }

    public void reset() {
        this.currentValue = 0;
    }

    public int getValue() {
        return this.currentValue;
    }
}
```
The two methods -- `increment()` and `reset()` -- rely on the current state (count) of the individual instance whose methods they are. Two different counters can have two different states (e.g., one can have count 4 and the other count 27). Incrementing the first will have a different effect (producing 5, etc.) from incrementing the second (which produces 28). Resetting one will not reset the other. `Increment()` and `reset()` make no sense without reference to the particular counter instance they're incrementing or resetting. This relationship between state (data members) and methods is typical of "traditional" objects.

![Picture of multiple counters.]

Traditional objects are exemplified by the following properties:

- Each instance has its own state.
- This state is not directly accessible. Instead, it provides the basis for method behavior.
- Method behavior is dependent on the internal state of a particular instance.
- State plus behavior, packaged together, provide a single logical unit.

**Types and Objects**

**Declared Type and Actual Type**

What happens when we take an object of one type and treat it as though it had another type? One common example of this that we've seen is using an interface-type name to hold an object. The object is an instance of some class. The name says that it's in instance of some interface. The interface provides a much more limited view of the object than the actual implementation. Does this change the object? What happens when we ask whether the object is an instance of its class, for example.

The answer is that the object is the same object no matter what its declared type (e.g. the declared type of the name that may be holding it, or of the method that may return it, or wherever else its type may be declared). It can do all of the same things regardless of its declared type. And it responds the same way when asked whether it is an instance of its class, regardless of whether its declared type is some more specialized interface.

For example, if we take an instance of the Counter class defined above, with its `reset()`, `increment()` and `getValue()` methods, and assign it to a name of type Counting (an interface with only `increment()` and `getValue()` methods), we haven't actually changed the Counter instance:

```
Counting count = new Counter();
```

If we ask whether

```
count instanceof Counter
```

this is true. Of course

```
count instanceof Counting
```

is also true. But
count instanceof BasicCounter

is false, given the definitions earlier in this chapter.

Using a Counting name instead of a Counter name does have some effect, though. First, we may not know about the Counter type. In this case, we are limited to treating count as though it were a Counting, not a Counter. For example, we couldn't call its reset() method, because Countings don't have reset() methods. Even if we did know about Counters, we'd have to explicitly cast count to be a Counter before we could use its Counter-specific properties:

    ( (Counter) count ).reset();

So an interface provides a limited view without limiting the actual object.

**Use Interface Types**

When declaring names and otherwise using objects, you should generally use interface types rather than class types. This allows the implementation of objects to vary independently of their use. It also allows different versions of the object to be used without dependence on unnecessary or possibly mutable properties. An interface allows common behavior to be abstracted and relied on. An interface can also be used to allow for future abstraction and variation, such as the Counting interface that allowed for the creation of a Timer.

For example, suppose that we are building a video game. The outer window of the video game is likely to be the same whether the game is Pong or Battleship or SpaceInvaders. It has controls such as start, stop, reset, and pause. What exactly happens when these controls are invoked depends on the particular game that is displayed in this window. But we want to build a generic DefaultGameFrame window that doesn't have to rely on the particular type of game that it will hold. We can accomplish this using an interface.

    public interface GameControllable
    {
        public void start();
        public void stop();
        public void reset();
        public void pause();
        public void unpause();
    }

Now, the DefaultGameFrame can refer to the game using the type GameControllable. As long as Pong or Battleship or SpaceInvaders implements GameControllable, any of these games can be used inside the DefaultGameFrame. When the DefaultGameFrame's reset control is invoked, DefaultGameFrame simply calls its GameControllable's reset() method. If the GameControllable happens to be Pong, it will bring the paddles back to rest and set the scores to 0. If the GameControllable is space invaders, the player will begin again with a full set of ammunition and plenty of aliens to shoot.

**Use Contained Objects to Implement Behavior**

One object can use another to provide behavior on the first object's behalf. For example, we might have a Clock object that provides a getTime() method and a setTIme() method. We might also have a VCR object
that includes among its functionality getTime() and setTime(). Should the VCR implement its own
getTime() and setTime() methods? This seems awfully inefficient. Or should the VCR reuse the Clock's
g getTime() and setTime() methods directly? (We will see a mechanism by which this can be accomplished
in the chapter on Inheritance.) The problem with this solution is that the VCR isn't really a Clock (or a kind
of Clock). Instead, the VCR can provide these methods by having a Clock inside it.

For example, the code for the VCR might say (in part):

```java
public class VCR
{
    private Clock clock;

    public Time getTime()
    {
        return this.clock.getTime();
    }

    public void setTime( Time t )
    {
        this.clock.setTime( t );
    }

    // etc.
}
```

In this way, the VCR provides access to the Clock's methods indirectly. This reuse of behavior by
inclusion is a very powerful mechanism. In this case, the VCR might be providing access to the full set of
Clock's methods. In another case, the including class might only provide a subset of the included class's
methods, or it might provide a superset by combining those methods in different ways. The including class
and the included class can even implement a common interface (such as TimeStorer) so that code that uses
one or the other can't really tell the difference so long as it only uses the interface's methods.

The DefaultGameFrame and GameControllable described above are similar. When the DefaultGameFrame
is asked to perform a reset (or a start or a stop or...), it passes this request along to the GameControllable.
In that case, the use of an interface type -- GameControllable -- for the included object increases the
flexibility and usability of the including class.

**The Power of Interfaces**

Why are interfaces so good at providing this flexibility? Because and interface is all about the contract an
object makes and not about implementation. By relying on an interface, you defer any dependence on
implementation details that might not be true of another implementation. This independence from
implementation-specific details is enforced by the compiler, which will not let you rely on properties of an
object specified by its interface type beyond those explicitly declared in the interface.

An object can also implement many different interfaces. In this case, it can be "seen" by other objects
through each of these different interface types. Each interface type provides a different view of the object.
By controlling these interfaces, a programmer controls the view that the object's users have of that object.
Reliance on interface types doesn't work perfectly, though. For example, a resource library such as Console or Math doesn't have an interface type. This is because resource libraries are typically non-instantiable classes. Only instances can have interface types.

Chapter Summary

- In an informal description of the program, nouns generally correspond to objects or to fields, methods to verbs, and interfaces to adjectives.
- Classes are the factories from which objects are created.
- Interface types provide a valuable layer of abstraction, allowing the implementation to vary without affecting the use.
- Members, classes, and instance marked public are accessible from anywhere within a program. Members marked private are only accessible within their defining class or instance.
- A data repository object exists to glue together a set of interdependent data. It has fields corresponding to this data and methods that allow you to read and modify this data.
- A resource library exists to hold a collection of methods or system-wide resources. Generally, a resource library supplies these methods and resources statically, i.e., it is not a class that is ever instantiated.
- Traditional objects mix both data and methods. These objects provide the kind of integrated state-dependent behavior that we expect of real world objects.

Exercises

1. Design and implement a class called Time that keeps track of the hour and minute together. Give it a
   nextMinute method that returns another Time, a minute later. How do you access the fields of Time
   objects?

2. Design and implement a class that provides IntegerArithmetic functions add( int, int ), sub( int, int ),
   mul( int, int ), and div( int, int ). You can give it any other methods you think might be useful. What does
   its constructor do? Why do you think that Java doesn't have such a class?

3. Design and implement a 2DVector class representing vectors in the plane. Include sum, difference, and
   product methods.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming

  textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101
Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and
formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology.

Questions or comments:

  <webmaster@cs101.org>

http://www.cs101.org/ipij/oo-design.html
Animate Objects

Chapter Overview

- How do I create an object that can act by itself?

This chapter builds on the previous ones to create an object capable of acting without an external request. Such an object has its own instruction follower, in Java called a Thread. In addition, an object with its own instruction-follower must specify what instructions are to be followed. This is accomplished by implementing a certain interface -- meeting a particular contract specification -- that indicates which instructions the Thread is to execute.

The remainder of this chapter deals with examples of how Threads and animate objects can be used to create communities of autonomously interacting entities.

Objectives of this Chapter

- To understand that Threads are Java's instruction-followers.
- To appreciate the relationship between a Thread and the instructions that it executes.
- To be able to construct an animate object using AnimatorThread and Animate.

Animate Objects

In previous chapters, we saw how objects group together state and behavior. Some objects exist primarily to hold together constituent pieces of a single complex state. Other objects exist to hold a static collection of primarily functional or system-specific resources. Most objects contain both local state and methods that rely on and interact with this state in complex ways. Many of these objects wait for something to happen or for someone else to ask them to act. That is, nothing happens until something outside the object invokes a method of the object. In this chapter, we look at objects that are capable of taking action on their own, without being asked to do so from outside. These objects have their own instruction-followers, making them full-blown entities.

Consider, for example, the Counter. This is a relatively traditional object. It has both state and methods that depend on that state. An individual counter object encapsulates this state-dependent behavior, wrapping it up into a neat package. But a counter doesn't do anything unless someone asks it to, using its increment() or reset() method. By itself, a counter can't do much.

Contrast this with a timer. A timer is very similar to a counter in having a method that advances it to the next state (paralleling the counter's increment() method) and one that sets the state back to its default
condition (such as reset()). A timer differs from a counter, however in that a timer counts merrily along whether someone asks it to or not. The timer's reset() method is a traditional (passive) method; the timer resets only when asked to. But the timer's increment() method is called by the timer itself on a regular basis.

This kind of object -- one that is capable of acting without being explicitly asked to do so -- is called an animate object. Such an object has its own instruction-follower, or actor, associated with it. While traditional objects are roles that an actor may take on and then leave, an animate object is a role that is almost always inhabited by an actor and tightly associated with it. Often, animate objects will use traditional objects (as well as data repositories, resource libraries, and other kinds of objects) to perform their tasks, temporarily executing instructions contained in these objects. But the animate object is where it begins and ends.

What makes an animate object different from other (passive) objects? Recall that on the first page of the first chapter of this book, we learned about the two prerequisites for a computation: The instructions for the computation must be present, and those instructions must be executed. Every method of every object is a set of instructions -- a rule -- that can be executed. When a method is invoked, its body is executed. (The method body is executed by the instruction-follower that invoked the method; this is how a method invocation expression is evaluated.)

An animate object differs from other objects because it also has its instruction follower. It does not need to wait for another instruction-follower to invoke one of its methods (although this may also happen). Instead, it has a way to start execution on its own.

In Java, an instruction-follower is called a Thread. No object can act except a Thread. A Thread is a special object that "breathes life" into other objects. It is the thing that causes other objects to move. An animate object is simply an object that is "born" with its own Thread. (Typically, this means that it creates its own Thread in its constructor and starts its Thread running either in its constructor or as soon as otherwise possible.)

**Animacies are Execution Sequences**

In every method of every object, execution of that method follows a well-defined set of rules. When the method is invoked, its formal parameters are associated with the arguments supplied to the method call. For example, recall the UpperCaser StringTransformer:

```java
public class UpperCaser extends StringTransformer {
    public String transform( String what )
    {
        return what.toUpperCase();
    }
}
```

If we have UpperCaser cap = new UpperCaser(); then evaluating the expression cap.transform( "Who's there?" ) has the effect of associating the value of the String "Who's there?" with the name what during the execution of the body of the transform method.
Now, the first statement of the method body is executed. In the case of the method invocation expression 
`cap.transform( "Who's there?")`, there is only one statement in the method body. This is the return statement, which first evaluates the expression following the return, then exits the method invocation, returning the value of that expression. To evaluate the method invocation expression `what.toUpperCase()` involves first evaluating the name expression `what` and then invoking the `toUpperCase()` method of the object associated with the name `what`.

No matter how complex the method body, its execution is accomplished by following the instructions that constitute it. Each statement has an associated execution pattern. A simple statement like an assignment expression followed by a semicolon is executed by evaluating the assignment expression. Expressions have rules of evaluation; in the case of an assignment, the right-hand side expression is evaluated, then that value is assigned to the left-hand side (shoebox or label). Evaluating the right-hand side expression may itself be complicated, but by following the evaluation rules for each constituent expression, the value of the right-hand side is obtained and used in the assignment.

A more complex statement, such as a conditional, has execution rules that involve the evaluation of the test expression, then execution of one but not both of the following substatements (the "if-block" or the "else-block"). Loops and other more complex statements also have rules of execution. Declarations set up name-value associations; return statements exit the method currently being executed.

At any given time, execution of a particular method is at a particular point and in a particular context (i.e., with a particular set of name-value associations in force). If we could keep track of what we're in the middle of doing and what we know about while we're doing it, we could temporarily suspend and resume execution of this task at any time. Imagine that you're following an instruction booklet to assemble a complex mechanism. This problem is a lot like placing a bookmark into your instructions while you go off to do something else for a while. All you need to know is where you were, what you had around you, and what you were supposed to do next; the rest of the instructions will carry you forward.

Inside the computer, there are things that keep track of where you are in an execution sequence. These are special Java objects called Threads. The trick is that there can be more than one Thread in any program. In fact, there are exactly as many things going on at once as there are Threads executing in your program. A Thread keeps track of where it is in its own execution sequence. Each Thread works on its own assembly project using its own instruction booklet, just like multiple people can work side by side in a restaurant or a factory.

In this book, we will make extensive use of a special kind of Thread called an AnimatorThread. An AnimatorThread is an instruction follower that does the same thing over and over again. It also has some other nice properties: it can be started and stopped, suspended and resumed. These last two mean that it is possible to ask your instruction follower to take a break for a while, then ask it later to continue working. AnimatorThreads provide a nice abstraction for the kinds of activities commonly conducted by the animate objects that are often entities in our communities.

**Being Animate-able**

In order for a Thread to animate an object, the Thread needs to know where to begin. A Thread needs to know that it can rely on the object to have a suitable beginning place. There must be special contract between the Thread and the object whose instructions this Thread is to execute. The object promises to supply instructions; the Thread promises to execute them. (In the case of the AnimatorThread, it promises
to execute these instructions over and over again.) As we know, such a contract is specified using a Java interface. This interface defines a method containing the instructions that the Thread will execute. The Thread will begin its execution at the instructions defined by this method.

**Implementing Animate**

If we use an AnimatorThread to animate our object, our object must fulfill the specific contract on which AnimatorThread begins. This contract is specified by the interface Animate:

```java
public interface Animate
{
    public abstract void act();
}
```

The Animate interface defines only a single method, void act(). A class implementing Animate will need to provide a body for its act() method, a set of instructions for how that particular kind of object act()s. An AnimatorThread will call this act() method over and over again, repeatedly asking the Animate object to act().

For example, the Timer that we described above could be implemented just as the Counter, but with the addition of an act() method:

```java
public void act()
{
    this.increment();
}
```

Of course, we'd also have to declare that Timer implements the Animate interface. It isn't enough for Timer to have an act() method; we also have to specify that it does so as a commitment to the Animate interface. Here is a complete Timer implementation:

```java
public class Timer implements Animate
{

    private int currentValue;

    public Timer()
    {
        this.reset();
    }

    public void increment()
    {
        this.currentValue = this.currentValue + 1;
    }

    public void reset()
    {
        this.currentValue = 0;
    }

    public int getValue()
```
{  
    return this.currentValue;
}

public void act()
{
    this.increment();
}

Note that the implementation is entirely identical to the implementation of Counter except for the clause implements Animate and Timer's act() method. [Footnote: As we shall see in the next chapter, we could significantly abbreviate this class by writing it as

```java
public class Timer extends Counter implements Animate, Counting
{
    public void act()
    {
        this.increment();
    }
}
```

Now Timer tick = new Timer(); defines a Timer ready to be animated.

**AnimatorThread**

On the other side of this contract is the instruction follower, an AnimatorThread. Like any other kind of Java object, a new AnimatorThread is created using an instance construction (new) expression and passing it the information required by AnimatorThread's constructor. The simplest form of AnimatorThread's constructor takes a single argument, an Animate whose act() method the new AnimatorThread should call repeatedly.

For example, we can animate a Timer by passing it to AnimatorThread's constructor expression:

```java
Timer tick = new Timer();
AnimatorThread mover = new AnimatorThread( tick );
```

There is one more thing that we need to do before tick starts incrementing itself: tell the AnimatorThread to startExecution():

```java
mover.startExecution();
```

An AnimatorThread's startExecution() is a very special method. It returns (almost) immediately. At the same time, the AnimatorThread comes to life and begins following its own instructions. That is, before the evaluation of the method invocation mover.startExecution(), there was only one
Thread running. At the end of the evaluation of the invocation, there are two Threads running, the one that followed the instruction `mover.startExecution()` and the one named `mover`, which begins following the instructions at `tick's act()` method.

Once started, the AnimatorThread's job is to evaluate the expression `tick.act()` over and over again. Each time, this increments `tick's currentValue` field. The AnimatorThread named `mover` calls `tick's act()` method over and over again, repeatedly causing `tick` to act.

We can collapse the two AnimatorThread statements into one by writing

```java
new AnimatorThread( tick ).startExecution();
```

However, this form does not leave us holding onto the AnimatorThread, so we couldn't later tell it to `suspendExecution()`, `resumeExecution()`, or `stopExecution()`. (See below.) If we anticipate needing to do any of these things, we should be sure to hold on to the AnimatorThread (using a label name).

**Creating the AnimatorThread in the Constructor**

If our Timers will always start ticking away as soon as they are created, we can include the Thread creation in the Timer constructor:

```java
public class AnimatedTimer implements Animate
{
    private int currentValue;
    private AnimatorThread mover;

    public AnimatedTimer()
    {
        this.reset();
        this.mover = new AnimatorThread( this );
        this.mover.startExecution();
    }

    public void increment()
    {
        // ... rest of class is same as Timer
    }
}
```

In this case, as soon as we say

```java
Timer tock = new AnimatedTimer();
```

tock will begin counting away. If we invoke `tock.getValue()` at two different times -- even if no one (except its own AnimatorThread) asks tock to do anything at all in the intervening time -- the second value might not match the first. This is because tock (with its AnimatorThread) can act without needing anyone else to ask it.

Here is another class that could be used to monitor a Counting (such as a Counter or a Timer):
public class CountingMonitor implements Animate
{
    private Counting whoToMonitor;
    private AnimatorThread mover;

    public CountingMonitor( Counting whoToMonitor )
    {
        this.whoToMonitor = whoToMonitor;
        this.mover = new AnimatorThread( this );
        this.mover.startExecution();
    }

    public void act()
    {
        Console.println( "The timer says "
                        + this.whoToMonitor.getValue() );
    }
}

Note in the constructor that the first whoToMonitor (this.whoToMonitor) refers to the field, while the second refers to the parameter.

A Generic Animate Object

The way that AnimateTimer and CountingMonitor use an AnimatorThread is pretty useful. There is a cs101 class, AnimateObject, that embodies this behavior. It is probably the most generic kind of animate object that you can have; any other animate object would behave like a special case of this one. We present it here to reinforce the idea of an independent animate object. It generalizes both CountingMonitor and AnimateTimer.

At this point, you should regard this class as a template. Change its name and add a real act() method to get a real self-animating object. In the chapter on Inheritance, we will return to this class and see that there is a way to make this template quite useful directly.

public class AnimateObject implements Animate
{
    private AnimatorThread mover;

    public AnimateObject()
    {
        this.mover = new AnimatorThread( this );
        this.mover.startExecution();
    }

    public void act()
    {
        // what the Animate Object should do repeatedly
    }
}
It is worth noting that an Animate need not be animated by an AnimatorThread. For example, a group of Animates could all be animated by a single SequentialAnimator that asks each Animate to act(), one at a time, in turn. No Animate could act() while any other Animate was mid-act(). Each would have to wait for the previous Animate to finish. This SequentialAnimator would require only a single instruction follower (or Thread) to execute the sequential Animates’ instructions, because it would execute them one act() method at a time. When one animate is acting, no one else can be.

The nature of execution under such a synchronous assumption would be very different from executions in which each Animate had its own Thread and they were all acting simultaneously. Roughly it's the difference between a puppet show with one not-very-skillful puppeteer, who can only operate a single puppet at a time, and a whole crowd of puppeteers each operating a puppet. The potential for chaos is much greater in the second scenario, but so is the potential for exciting interaction. When each object has its own AnimatorThread -- as in the AnimateObject template -- any other Animate (or the methods it calls) can execute at the same time.

**More Details**

This section broadens the picture painted so far.

**AnimatorThread Details**

The AnimatorThread class and the Animate interface reside in the package cs101.lang. This means that any file that uses these classes should have the line

```java
import cs101.lang.*;
```

before any class or interface definition.

The class AnimatorThread specifies behavior for a particular kind of instruction follower. Its constructor requires an object that implements the interface cs101.lang.Animate, the object whose act() method the AnimatorThread will repeatedly execute.

After constructing an AnimatorThread, you need to invoke its startExecution() method. [Footnote: AnimatorThread's instances also have a startExecution() method that is identical to the startExecution() method. This is for historical reasons.] This causes the AnimatorThread to begin following instructions. In particular, the instructions that it follows say to invoke its Animate's act() method, then wait a little while, then invoke the Animate's act() method again (and so on). To temporarily suspend execution, use the AniamtorThread's suspendExecution() method. Execution may be restarted using resumeExecution(). To permanently terminate execution, AnimatorThread has a stopExecution() method. Once stopped, an AnimatorThread's execution cannot be restarted. However, a new AnimatorThread can be created on the same Animate object.

An object -- like an Animate -- is a set of instructions -- or methods -- plus some state used by these instructions. There is nothing to prevent more than one Thread from following the same set of instructions at the same time. For example, it would be possible to start up two AnimatorThreads on the same Timer. If the two AnimatorThreads took turns fairly and evenly, one AnimatorThread would always move from an odd to an even numbered currentValue, while the other would always move from an even to an odd numbered value. Of course, there's nothing requiring that the two AnimatorThreads play fair. Like
children, one might take all of the turns -- incrementing the Timer again and again -- while the other might never (or rarely) get a turn. AnimatorThreads are designed to minimize this case, but it can happen. The problem is more prevalent with other kinds of Threads.

One of the ways in which AnimatorThread tries to "play fair" is in providing intervals between each attempt to follow the act() instructions of its Animate object. The AnimatorThread has two values that it uses to determine the minimum interval between invocations of the Animate's act() method and the maximum interval. Between these two values, the actual interval is selected at random each time the AnimatorThread completes an act(). You can adjust these parameters using setter methods of the AnimatorThread. Values for these intervals may also be supplied in the AnimatorThread's constructor. See the AnimatorThread sidebar for details.

**class AnimatorThread**

AnimatorThread is a cs101 class (specifically, cs101.lang.AnimatorThread) that serves as a special kind of instruction-follower. An AnimatorThread's constructor must be called with an instance of cs101.lang.Animate. The AnimatorThread repeatedly follows the instructions in the Animate's act() method.

An AnimatorThread is an object, so it can be referred to with an appropriate (label) name. It also provides several useful methods:

- **void startExecution()** causes the AnimatorThread to begin following the instructions at its Animate's act() method. Once started, the AnimatorThread will follow these instructions repeatedly at semi-random intervals until it is stopped or suspended.

- **void stopExecution()** causes the AnimatorThread to terminate its execution. Once stopped, an AnimatorThread cannot be restarted. This method may terminate execution abruptly, even in the middle of the Animate's act() method.

- **void suspendExecution()** causes the AnimatorThread to temporarily suspend its execution. If the AnimatorThread is already suspended or stopped, nothing happens. If the AnimatorThread has not yet started and is started before an invocation of resumeExecution(), it will start in a suspended state, i.e., it will not immediately begin execution. This method will not interrupt an execution of the Animate's act() method; suspensions take effect only between act()s.

- **void resumeExecution()** causes the AnimatorThread, if suspended, to continue its repeated execution of its Animate's act() method. If the AnimatorThread is not suspended or already stopped, this method does nothing. If the AnimatorThread is suspended but not yet started, invoking resumeExecution() undoes the effect of any previous suspendExecution() but does not startExecution().

Between calls to the Animate's act() method, the AnimatorThread sleeps, i.e., remains inactive. The duration of each of these sleep intervals is randomly chosen to be at least sleepMinInterval and no more than sleepMinInterval + sleepRange. These values are by default set to a range that allows for variability and slows activity to a rate that is humanly perceptible. If you wish to change these defaults, they may be set either explicitly using setter methods or in the AnimatorThread constructor.
void setSleepRange( long howLong ) sets the desired variance in sleep times above
and beyond sleepMinInterval

void setSleepMinInterval( long howLong ) sets the range of variation in the
randomization

By setting sleepRange to 0, you can make your AnimatorThread's activity somewhat more predictable as
it will sleep for approximately the same amount of time between each execution of the Animate's act() method. Setting sleepMinInterval to a smaller value speeds up the execution rate of the AnimatorThread. Setting it to 0 can be dangerous and should be avoided. If sleepRange is 0, it is possible that this
AnimatorThread will interfere with other Threads' ability to run.

AnimatorThread supplies a number of constructors. The first requires only the Animate whose act method
supplies this AnimatorThread's instructions:

AnimatorThread( Animate who )

The next two constructors incorporate the same functions as setRange and setMinInterval:

AnimatorThread( Animate who, long sleepRange )

AnimatorThread( Animate who, long sleepRange,
long sleepMinInterval )

It is also possible to specify explicitly whether the AnimatorThread should start executing immediately.
By default, it does so. The following constructor allows you to override this explicitly using the boolean
constants AnimatorThread.START_IMMEDIATELY and AnimatorThread.DONT_START_YET.

AnimatorThread( Animate who, boolean startImmediately )

Finally, there are two additional constructors that incorporate both startup and timing information:

AnimatorThread( Animate who, boolean startImmediately,
long sleepRange )

AnimatorThread( Animate who, boolean startImmediately,
long sleepRange, long sleepMinInterval )

Delayed Start and the init() Trick

It is awfully convenient to be able to define an animate object as an Animate that creates and starts its own
AnimatorThread. This hides the Thread creation and manipulation inside the Animate (as in the example of
AnimateTimer), making it appear to be a fully self-animating object from the outside. However, sometimes
we need to separate the construction of the Animate and its AnimatorThread from the initiation of the
AnimatorThread instruction follower. That is, we want the AnimatorThread set up, but not yet actually
running. For example, we might need a part that isn't yet available at Animate/AnimatorThread creation
time. On these occasions, it would be awkward to start the execution of an AnimatorThread in the
constructor of its Animate. For example, if the Animate's act() method relies on other objects and these other objects may not yet be available, you wouldn't want the AnimatorThread to start executing the act() method yet.

An example of this might be in the StringTransformer class in the first interlude, in which you can't read or transform a String until after you've accepted an input connection. Since the input connection might not be available at StringTransformer construction time, one solution to this problem is to delay the starting of the execution of the act() method until after the input connection has been accepted. Once the constructor completes, the newly constructed object's acceptInputConnection method can be invoked. At this point -- and not before -- the AnimatorThread's startExecution() method can be invoked. This means that the call to the AnimatorThread's startExecution() method can't appear in the constructor. But it can't be invoked by any object other than the Animate, because the AnimatorThread is held by a private field of the Animate.

This situation -- that there are things that need to be done that are logically part of the setup of the object, but that cannot be done in the constructor itself -- is a common one. To get around it, there is a convention that says that such objects should have init() methods. Whoever is responsible for setting up the object should invoke its init() method after this setup is complete. The object can rely on the fact that its init() method will be invoked after the object is completely constructed and -- in this case -- connected. We could then put the call to the AnimatorThread's startExecution() method inside this init() method.

Here is a delayed-start version of the AnimateObject template.

```java
public class InitAnimateObject implements Animate
{

    private AnimatorThread mover;

    public InitAnimateObject()
    {
        this.mover = new AnimatorThread( this );
    }

    public void init()
    {
        this.mover.startExecution();
    }

    public void act()
    {
        // what the Animate Object should do repeatedly
    }
}
```

A concrete example of this issue arises if we look at CountingMonitor and don't assume that the Counting will be supplied to the constructor. Here is another version of CountingMonitor without the constructor parameter:

```java
public class InitCountingMonitor implements Animate
{
```

http://www.cs101.org/ypi/animacies.html
private Counting whoToMonitor;
private AnimatorThread mover = new AnimatorThread( this );

public void setCounting( Counting whoToMonitor )
{
    this.whoToMonitor = whoToMonitor;
}

public void init()
{
    this.mover.startExecution();
}

public void act()
{
    Console.println( "The timer says "
    + this.whoToMonitor.getValue() );
}

The use of a method named init() here is completely arbitrary. You are free to define your own method and call it whatever you want. However, you will see that many people follow this convention and provide an init() method for their objects when there is initialization that must take place after the constructor and setup process is complete.

Threads and Runnables

The Animate/AnimatorThread story that we've just seen is not a standard part of Java, though it is only a minor variant on something that is. There are two reasons why we've used AnimatorThreads here. The first is that most of the self-animating object types in this book are objects whose act method is executed over and over again. AnimatorThread is a special kind of Thread designed to do just that. The second is that AnimatorThread contains some special mechanisms to facilitate its use in applications where you might want to suspend and resume its execution or even to stop it entirely. AnimatorThread provides methods supporting this behavior.

There is, however, in Java a more primitive type of Thread, called simply Thread. Like an AnimatorThread, a simple Java Thread can be given an object to animate when the Thread is created. (Its constructor takes an argument representing the object whose instructions the Thread is to follow once it has been started.) However, the Thread does not execute this method repeatedly; it executes it once, then stops. The contract that a Thread requires of the object providing its instructions is not Animate, meaning it can be called on to act repeatedly. Instead, it is Runnable, meaning it can be executed once.

Thread (as of Java 1.1) does not provide suspension, resumption, or cessation methods. In this book, we avoid the use of plain Java Threads.

In addition, it is technically possible in Java to extend a Thread object rather than passing it an independent Runnable. Except in code that creates special kinds of Threads (such as AnimatorThread) capable of animating other objects, the extending of Thread is highly discouraged in this book. Extending Thread to
create an executing object (whose own run() method is the set of instructions to be followed) confounds the notion of an executor with the executed.

**Thread Methods**

### Thread methods

Threads are Java's instruction followers. In this book, we will most often make use of AnimatorThreads. However, it is useful to understand how Java's built-in Thread class works as well.

Like an AnimatorThread, each Thread provides a few methods for its management.

- **void start()** Like AnimatorThread's startExecution(), this method causes the target Thread to begin following instructions. If the Thread's constructor was supplied a Runnable, the Thread begins execution at this Runnable's run() method. When the run() method terminates, the Thread's execution is finished.

- **boolean isAlive()** tells you whether the target Thread is alive, i.e., has been started and has not completed its execution.

- **void interrupt()** sends the target Thread an InterruptedException. Useful if that Thread is sleeping, waiting, or joining.

- **void join()** causes the invoking Thread to suspend its execution until the target Thread completes. Variants allow time limits on this suspension: void join( long millis ) and void join( long millis, long nanos ).

Unlike AnimatorThread, a Thread cannot safely be stopped, suspended, or resumed.

In addition to its role as the type of Java's instruction followers, the Thread class provides useful static (i.e., class-wide) functionality. These methods are static methods of the class Thread:

- **static void sleep( long millis )** causes the currently active Thread to stop executing for millis milliseconds. This method throws InterruptedException, so it cannot be used without some additional machinery (introduced in the chapter on Exceptions). There is a variant method, sleep( long millis, long nanos ) that allows more precision in controlling the duration of the Thread's sleep.

- **static void yield()** is intended to pause the currently executing Thread and to allow another Thread to run. However, not all versions of Java implement Thread.yield in a way that ensures this behavior.

Other Thread features are outside the scope of this course.

**Where do Threads come from?**
We have discussed the idea of AnimatorThreads above, showing how to create self-animating objects by having an AnimatorThread created in an object's constructor. Such an object is born running; it continually acts, over and over, until its Thread is suspended or stopped.

In fact, no execution in Java can take place without a Thread. But something must call the AnimatorThread constructor; this instruction must be executed by a Thread! So where does the first Thread come from? This depends on the particular kind of Java program that you are running. In this book, we look primarily at Java applications. In the appendix, we also answer these questions for Java applets.

### Starting a Program

What does it mean for a Java program to run? It means that there is an instruction follower that executes the instructions that make up this program. In Java, there is no execution without a Thread, or instruction-follower, to execute it. So when a program is run, some Thread must be executing its instructions. Where does this Thread come from, and how does it know what instructions to execute?

Let's answer the first of these questions first. When a Java program is run, a single Thread is created and started. This is not a Thread that your program creates; it is the Thread that Java creates to run your program. Depending on whether your Java program is an application (as we're discussing in this book) or an applet (as you may have encountered on the world-wide web) or some other kind of Java program, there are different conventions as to where this Thread begins its execution. But running a program by definition means creating a Thread -- an instruction follower -- to execute that program.

How does the Thread know where to begin? By convention. What do we mean by a convention? AnimatorThread's use of Animate is a convention. This convention is, in some sense, completely arbitrary. That is, a different interface name or other method might have been used. For example, the raw Thread class uses a different convention, that of Runnable/run(). If you were to design your own type of Thread, you could create a different convention for it to follow. However, once these names and contracts have been selected by the designers of AnimatorThread and Thread, they are absolute rules that cannot be violated.

Similarly, there must be some arbitrary convention as to how a Java program begins. In a standalone application, the convention is that running a Java program means supplying a class to the executable, and by convention a particular method of the class is always the place that execution begins. This default execution does not create an instance of the class, so the method must be a static one. Again by convention, the name of this method is main, it takes as argument an array of Strings, and it returns nothing. That is, the arbitrary but unvarying start point for the execution of a standalone Java application is the

```java
public static void main ( String[] args )
```

method of the class whose name is supplied to the executable.[Footnote: Typically, this means the class you select before choosing run from the IDE menu or the class whose name follows the command java on the command line.]

So if you want to write a program, you simply need to create a class with a method whose signature matches the line above. The body of that main method will be executed by the single Thread that is created at the beginning of a Java execution. When execution of main terminates, the program ends. If you do not want the program to end, you need to do something during the course of executing main that causes things...
to keep going. Typically, this means that you use the body of main to create one or more objects that themselves may execute. For example, if the body of main creates an animate object (with its own AnimatorThread), then that object will continue executing even if the body of main is completed. This is called "spawning a new Thread".

Here is a very simple class that exists solely to create a new instance of the AnimateTimer class:

```java
public class Main {

    public static void main ( String[] args ) {
        Counting theTimer = new AnimateTimer();
    }
}
```

This program simply counts. The instruction follower that begins when this program starts up (e.g., using java Main) executes the main() method, invoking new AnimateTimer() and assigning the result to theTimer. This Thread is now done executing and stops. However, the constructor for AnimateTimer has created a new AnimatorThread and then called that AnimatorThread's startExecution() method. This starts up the new Thread which repeatedly calls AnimateTimer's act() method. The program as a whole will not terminate until the AnimatorThread stops executing, which it will not do by itself. If you run this program, you will need to forcibly terminate it from outside the program!

Since we didn't give this program any way to monitor or indicate what's going on, running it wouldn't be very interesting. But we can use the CountingMonitor above to improve this program:

```java
public class Main {

    public static void main ( String[] args ) {
        Counting theTimer = new AnimateTimer();
        Animate theMonitor = CountingMonitor( theTimer );
    }
}
```

Q. Can you find a more succinct way to express the body of the main method?

Q. What will be printed by this program? On what does it depend? (Hint: fairness.)

The instruction follower executing the Main class's main method exits. However, before it completes it executes the instructions to create and start two separate AnimatorThreads. These AnimatorThreads continue after the execution of the main Thread exits. Again, this program must be forcibly terminated from outside.

Q. Can you cause this program to stop by itself sometime after it has counted to 100? (This is a bit tricky.)
The two versions of the Main class above each contain just the instructions to create an instance or two. In the cs101 libraries, we have provided a Main that does this for you. This allows you to write applications without needing to write public static void main( String[] ) methods yourself.

### class Main

The cs101 libraries include a class, cs101.util.Main, that can be run from the java command line to create an instance of a single class with a no-args constructor. For example, we could implement the unmonitored Timer example using the following command:

```
java  cs101.util.Main  AnimateTimer
```

This causes code much like the first Main class to execute, creating a single instance of AnimateTimer (using its no-args constructor).

The class cs101.util.Main contains nothing but the single static method main (taking a String[] argument). The command above tells Java to start its initial instruction follower at this method -- the static main(String[] ) method of the class cs101.util.Main. The remainder of the information on the command line (in this case, AnimateTester) is supplied to the main method using its parameter.[Footnote: For more detail on arrays ([[]), see the chapter on Dispatch.]

### Style Sidebar

**Using main()**

If you do decide to write your own main() method, you should do so in a class separate from your other classes, generally one called Main and containing only the single public static void main() method requiring a String[] (i.e., an array of Strings). This method may have some complexity, creating several objects and gluing them together, for example.

Alternately, you can create an extremely simple main method in any (or even every) class that you write. In this case, however, the main method should do nothing more than to create a single instance of the class within which it is defined, using that class's no-args constructor. Of course, the signature of each main method is the same: public static void main(String[] args) The main that will actually be executed is the one belonging to the (first) class whose name is supplied to the java execution command. So, for example, in the sidebar on class Main, we said

```
java  cs101.util.Main  AnimateTimer
```

causin cs101.util.Main's main method to be run.

The logic behind these restrictions on the use of main() is as follows. In the second case -- main in many instantiable class's files -- the presence of main allows that object to be tested independently. However, this test is extremely straightforward and predictable. If the main method takes on any additional complexity, it should be separated from the other (instantiable) classes and form its own resource library, one that exists solely to run the program in all its complexity.
Why Constructors Need to Return

In the code above, each Animate's constructor calls the startExecution() method of a new Thread. This in turn repeatedly calls the act() method of the Animate. Why doesn't the constructor just repeatedly call the Animate's act() method itself (e.g., in a while loop)?

This is a fundamental issue. If the Animate's constructor called the act() method itself, the instruction follower -- or Thread -- executing the constructor would be trapped forever in a loop calling act() over and over. The constructor invocation -- the new expression -- would never complete. In the monitored counting example, the invocation of AnimateTimer's constructor would cause the instruction follower to execute the act() method of AnimateTimer over and over again. This instruction follower -- the only instruction follower to be running so far -- would never complete the repeated execution of the act() method. This means that it would never get around to creating the CounterMonitor.

This is why AnimatorThread.startExecution() must be a very special kind of method. The Thread, or instruction follower, that executes startExecution() must return (almost) immediately. It is the new Thread, the one just started, that goes off to execute the act() method. The original Thread returns from this invocation and goes about its business just as if nothing ever happened. In personal terms, this is the difference between doing the job yourself and assigning someone else to do it. True, when someone else does it you have less control over how or when the job gets done; but while someone else is working on it, you can be doing something else.

Chapter Summary

- In Java, activity is performed by instruction followers called Threads.
- An animate object is simply one that has its very own Thread.
- An AnimatorThread is a useful kind of Thread that repeatedly follows the instructions provided by some object's act() method.
  - This object must implement the Animate interface.
  - It must be supplied to the AnimatorThread's constructor.
- An AnimatorThread can also be asked to start, stop, suspend, or resume execution.
- Java programs may involve other Threads.
  - One Thread begins execution at public static void main( String[] args ) when a Java application is begun.
  - GUI objects involve their own Threads.
  - Other Threads may be explicitly created.

Exercises

1. Define a class whose instances each have an internal value that doubles periodically. Each time that the value doubles, the instance should print this new value to the Console.
2. Define a class that periodically reads from the Console and writes the value back to the Console.
3. Define a main class that creates three instances of your doubler.
4. Using the timing parameters of AnimatorThread, demonstrate that not all doublers have to run at the same rate.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>

When Things Go Wrong: Exceptions

Chapter Overview

- What happens when something goes wrong?
- How do I create alternate ways to handle atypical circumstances?

This chapter covers mechanisms for dealing with atypical behavior. Sometimes, exceptional circumstances arise and require different mechanisms to cope with them. In this case, the normal entity-to-entity communication in your system may need to be interrupted. Java provides certain mechanisms for creating alternate paths through your community. These include the throw and catch statements as well as special Exception objects that keep track of these atypical circumstances.

This chapter includes sidebars on the syntactic and semantic details of throw and catch statements, exception objects, and the requirement to declare exceptions thrown. It is supplemented by portions of the reference charts on java methods and statements.

Objectives of this Chapter

1. To be able to read, understand, and write throws clauses as a part of interface and class contracts.
2. To learn how to throw and catch Exceptions and other Throwables.
3. To appreciate the role that anticipating exceptional circumstances plays in the design and testing of programs.

Exceptional Events

So far, the code that we have written has addressed "normal" situations in which nothing goes wrong. But sometimes, unusual things happen in our code, and we have to deal with them. In some cases, these unusual things are unexpected errors; in others, their existence is predictable but we may not know in advance when they are likely to happen. An example of this second kind is a network outage, which happens from time to time and can reasonably be anticipated, but is unexpected when it occurs. Planning for these exceptional circumstances and writing code that can cope with them is an important part of robust coding.

When Things Go Wrong

Consider the following example, drawn from the StringTransformers application of the first interlude. In that scenario, entities called StringTransformers are connected by "tin can telephone" entities called Connectors. Each Connector has an end that you can put something into and an end that produces what you
put into it. A StringTransformer can write to (or read from) a connector if it is holding the appropriate end. In the user interface of that application, there is a way that a user can specify two StringTransformers to be connected. We are going to look in more detail at how the Connector actually gets attached to these two StringTransformers.

Let's say that the two transformers we're going to connect are transformerA and transformerB. In the code that is making the connection, we invoke the specific Connector constructor with these transformers as arguments:

```java
new StringConnector( transformerA, transformerB )
```

The constructor code for StringConnector asks each of the transformers, in turn, to accept a(n input or output) connection. In fact, strictly speaking, A and B need not be StringTransformers at all; they need only implement the OutputAcceptor or InputAcceptor interfaces, since that is the only aspect of their behavior that we rely on here.

```java
public StringConnector( OutputAcceptor a, InputAcceptor b )
{
    a.acceptOutputConnection( this );
    b.acceptInputConnection( this );
}
```

This code is perfectly reasonable assuming that everything goes right. But what happens if transformerA already has an outputConnection in place? It might be that transformerA is a Broadcaster or AlternatingOutputter or some other kind of transformer that can have many outputConnections. It might be that transformerA is willing to throw away its existing outputConnection and replace it with the one currently on offer. But it might also quite reasonably be that transformerA is unwilling and unable to accept an OutputConnection if it already has one in place. In this case, the StringConnector constructor code is in trouble.

This is precisely the sort of situation that we will deal with in this chapter. Something has gone wrong. We can anticipate in our design that this might happen. We want our code to respond appropriately. In other words, we want to design our programs to be able to handle exceptional circumstances.

**Expecting the Unexpected**

When you are designing a program, it is relatively easy to think about what is supposed to happen. You can act out the interactions that you want your program to have. You can draw out storyboards describing what comes next. You design interfaces and protocols to describe the roles each entity plays and the contracts it makes with others. But this is not enough.

In addition to figuring out what *ought* to happen, you also need to anticipate what *might* happen. That is, you need to understand what happens if a component does something unexpected; if the user does something foolish; if a resource that you depend on becomes unavailable or temporarily out of service; or even if a change that you make to your code inadvertently violates an assumption. In all of these cases, unexpected behavior of one portion of the system needs to be dealt with. Good design involves anticipating these possibilities and explicitly deciding what to do and designing for these circumstances.
Exceptional circumstances can be partitioned into three groups. One is the catastrophic failure. In case of a catastrophic failure, there's really nothing that your program can do. This might happen, for example, if someone tripped over the power cord of the computer on which your program was running. In this case, it is reasonable to expect that your computer program will stop executing immediately. There's really nothing that you can do about a catastrophic failure.[Footnote: At least at the time of failure. There are still things that you can do to plan for recovery from catastrophic failure. For example, a banking system may temporarily lose the functioning of an ATM, but it will not lose track of your bank balance entirely. It has been designed to keep this information safe even in the face of computer crashes.]

The second kind of exceptional circumstance is at the other end of the spectrum. This is a situation that is not the intended course of your program, but is so benign that it is dealt with almost as a matter of course. These are the unexpected situations that can be handled with a simple conditional or other testing mechanisms. For example, if we are about to perform a division operator, we might check to make sure that the divisor is not 0. In the extreme, these situations can be difficult to distinguish from "normal" operation.

Most exceptional circumstances fall between these two extremes. That is, they admit some intervention or even solution (unlike catastrophic failure), but handling these circumstances requires cooperation among entities or other additional complexity; it isn't possible or desirable to deal with this situation locally. These are the situations that you must take into account in your design.

When you are planning your program, you will be deciding how to partition the problem among a community of interacting entities and designing how these entities interact. At this point, you should also ask:

- What should happen if one of these entities is unreachable?
- What are all of the ways in which an entity might violate expectations?
- What should happen in each of these cases?
- What should an entity do if it has difficulty fulfilling its contract?

In each of these cases, you should decide whether the circumstance amounts to a catastrophic failure or can be handled by another entity. If it is a catastrophic failure, this circumstance ought to be documented; if not, it provides another set of interactions to build into your system. This exception-handling becomes another part of your system design.

As you break each entity down -- asking what is inside it, decomposing it into further communities of interacting entities -- you should repeat these questions with respect to these entities' mutual commitments. Eventually, you will decompose your problem to the level of individual operations and of interactions with entities outside the system that you are actually building. For these situations, you should ask:

- In what ways might this operation or outside entity fail?
- How else might it violate my expectations?
- Can I test for these circumstances prior to invocation of this operation or resource?
- What should I do if the failure or expectation violation occurs?
If the situation is one that can be ruled out using a simple test -- such as checking for a zero divisor or verifying that the user's input is a legal value and asking for new input if not -- such error checking should be introduced into your design. This strengthens the contracts that entities make with one another. Where violations cannot be handled locally, you will need to decide who should handle the issue and how it should behave.

What's Important to Record

At the time that an exceptional circumstance arises, the currently executing code is in the best position to determine what the problem is. It should take pains to record any information that might help other parts of the program (or a human user or debugger) to figure out what happened. So, for example, in the case of a divide-by-zero error, it would be important to know what the expression was whose value was zero, causing the error. In the case of an invalid value entered by a user, it may be important to know what the invalid value is or what the legal values might be. It is also important to know what kind of thing went wrong: division by zero or illegal argument passed to a method or a label name that's null and shouldn't be or any of a whole host of possible values.

This information -- what kind of thing went wrong and kind-specific additional information that might be useful for figuring out what the problem was or correcting it -- is, in Java, encapsulated in a special kind of object. These are Exception objects. They signal what's gone wrong. There are many different (more specific) types of Exception objects, such as NullPointerException or IllegalArgumentException. You can also define Exception types of your own (using inheritance). In addition to Exceptions, Java also defines a (similar but distinct) class of Errors, meant to designate conditions of catastrophic failure, such as NoClassDefFoundError. You can (but rarely will) define your own Errors as well.

Since Exceptions are objects, you can use them like any other object. If you define your own Exception classes, you can add any fields or methods that you think might be important to allow your program to handle the exceptional circumstance. One thing that is especially useful for an Exception to have is a String (suitable for printing to a user) that explains something about what has gone wrong. In Java's Exception classes, such a String can be supplied to the constructor and retrieved using the instance's getMessage() method.

It can also be very important to know where the problem occurred. Java's Exception classes record the point at which they were thrown (see below), but it can in addition be useful to record (e.g., in the message or in an additional field that you define) some program-specific indication of which code is reporting the exceptional circumstance and what it was trying to do when the exception occurred.

For example, in our OutputAcceptor code, we might recognize that we can't accept an OutputConnection if we already have one. In this case, we might create a new ConnectionRejectedException recording this circumstance:

```java
new ConnectionRejectedException( this.toString()
    + " rejecting redundant OutputConnection" )
```

The ConnectionRejectedException uses the toString() method of the OutputAcceptor within which this code occurs to record who is rejecting the connection. An alternative is just to list the class name and method in a constant String: "OutputAcceptor.acceptOutputConnection()": "The
ConnectionRejectedException might also record the existing OutputConnection and the newly supplied one; in the code fragment above, it does not do this.

Just defining a new exception isn't enough, though. Defining an exception is like composing a letter of complaint. In order for it to have any effect, you have to send out the letter. In the case of an Exception, this is accomplished by **throwing** the Exception.

**Throwing an Exception**

An Exception is an unusual circumstance that requires special handling. In order to understand how an Exception works -- and what it means to throw one -- we first need to look at how method invocation and return normally works.

Let us begin by looking more closely at what happens in our new Connector example, when the user interface calls the StringConnector constructor, which in turn calls the OutputAcceptor's acceptOutputConnection method. We might diagram the normal control flow as follows:

```
User Interface constructor OutputAcceptor
------------------- --------
                      ------
                      (records Connection)
                      -------
                      (more activity)
```

The code from the user interface invokes the StringConnector constructor, then the StringConnector constructor invokes the OutputAcceptor's acceptOutputConnection method. When the acceptOutputConnection method completes, it returns (nothing) to the StringConnector's constructor, which completes its work and provides the newly constructed StringConnector to the User Interface. These arrows are sometimes called the **call path** (and return path) of this execution.

Communication among pieces of code is very simple. Each piece of code can only talk to the other pieces of code about which it knows. In this case, the User Interface knows about the StringConnector's constructor, and the StringConnector's constructor knows about the OutputAcceptor's acceptOutputConnection method. Think of it like an old-fashioned fire-fighting bucket brigade. All of the people line up from the water supply to the fire. A full bucket is passed from hand to hand down the line from the water supply to the fire. The empty bucket must be passed back the same way. In the normal motion of buckets, there is no way for a bucket to skip over a person; it must be passed from hand to hand, returning the way that it came.[Footnote: In this example, the "more activity" line inside the constructor is a shorthand for a more complex picture. This "more activity" actually involves another method call, this one to the InputAcceptor's acceptInputConnection method. So the whole picture is more accurately represented as

```
User Interface constructor OutputAcceptor InputAcceptor
------------------- -------- ----------
                      ------ ------
                      (records Connection)
                      ------- ------
                      (records Connection)
```

This doesn't violate the bucket brigade idea, but it does mean that the bucket brigade has a fork in it. The constructor can pass buckets to (i.e., invoke) both the OutputAcceptor's acceptOutputConnection method and the InputAcceptor's acceptInputConnection method.

Throwing an Exception is different. What happens in this case looks more like the following:

User Interface        Constructor        OutputAcceptor
-------------------->                        -------------------->
                      OH NO!!

When the OutputAcceptor's acceptOutputConnection method realizes that it has a problem it generates an Exception object, as we have seen above. Then, it throws the exception as hard as it can back the way it came. The Exception zooms back along the call path, flying too fast to stop and execute any statements waiting for its return. In fact, the Exception keeps going until it encounters a compatible catch statement. If necessary, it may exit several method bodies. Or, if the catch is in the same block as the throw, it may not exit any method bodies at all. In other words, a throw statement sets an Exception flying, and the flying Exception can only be caught by a matching catch statement; no other intervening statement along the call path matters.

This is, in fact, just what we want. If the OutputAcceptor can't accept an output connection, we don't want the rest of the Constructor to execute. For example, we don't want it to try to convince the InputAcceptor to accept the input end of the connection, because this connection isn't going to work out (since the OutputAcceptor isn't cooperating) and if the InputConnector accepts this one, then it won't later be able to accept a fully operational input connection. So when the OutputAcceptor decides that it has a problem, we want the Exception to propagate all the way back to the user interface code, which should decide that connecting this particular pair of String Transformers may not be such a good idea after all.

The code for the OutputAcceptor might look like this:

```java
... implements OutputAcceptor
{

    private OutputConnection out;

    public void acceptOutputConnection( OutputConnection out )
    {
        if ( this.out == null )
        {
            this.out = out;
        }
        else
        {
            throw new ConnectionRejectedException( this.toString()
                + " rejecting redundant OutputConnection" );
        }
    }
```


This example introduces a new statement type, `throw`, and a new declaration element, `throws`. (Note the s on the declaration element.) The throw statement works just as we have described; it abruptly terminates the execution of this method and causes the Exception to propogate backwards along the return path until a compatible catch statement is encountered. (We will see this below.)

What about the throws clause? Throwing an exception is actually part of the contract that one object makes with another. It is as much a part of a method's contract as its (normal) return type or the parameters it needs. So a method must declare that it may throw an exception (and what type of exception it may throw). This way, anyone calling the method knows to be prepared for it to throw this exception. The throws clause is the final part of a method signature, and throws clauses may appear in interface (abstract) method declarations as well as in method definitions.

Throws clauses are not restricted to methods. Constructors, too, must declare any exceptions that they throw. A constructor can explicitly throw an exception using a throw statement. A constructor (or method) can also throw an exception by calling something that throws an exception and then not catching it. This is what happens with the StringConnector constructor. Here it is, reprinted from above, with the added `throws` clause underlined.

```java
public StringConnector( OutputAcceptor a, InputAcceptor b )
throws ConnectionRejectedException
{
    a.acceptOutputConnection( this );
    b.acceptInputConnection( this );
}
```

The StringConnector constructor invokes OutputAcceptor's `acceptOutputConnection` method. If the OutputAcceptor doesn't accept the output connection, the StringConnector constructor isn't going to be able to fix this. So the StringConnector constructor should itself exit abruptly. In other words, the Exception thrown by `acceptOutputConnection` flies right out of the StringConnector constructor as well, still waiting to find a compatible catch clause.

### Throw Statements and Throws Clauses

A `throw` statement looks a lot like a `return` statement, but it always takes an argument (which can be in parentheses or not), and its argument must be something legal to throw. Anything that extends `Throwable` is legal to throw. In particular, this includes anything that extends `Exception`.

The effect of a `throw` statement is that execution abruptly returns up the call path until a compatible `catch` clause is encountered. Nothing except a compatible `catch` clause can stop the propagation of a thrown object.
If an Exception (except a RuntimeException) is thrown and not caught within a method or constructor body, you must also declare that that method or constructor throws the exception. This is a part of the signature, like saying what a method returns or what arguments a method or constructor expects.

The throws clause appears after the argument list, but before the method/constructor body. The syntax for a throws clause is

\[ \text{throws ExceptionType}_1, \text{ExceptionType}_2, \ldots \text{ExceptionType}_N \]

Every exception thrown and not caught within the body must match (at least) one of the exception types declared thrown by the method or constructor. If the method or constructor throws only a single exception type, the list contains no commas.

Catching an Exception

We have seen how an Exception can be generated and thrown. We have also seen that a thrown exception keeps flying until it encounters a compatible catch statement. Now, we will look at catch statements and how they work. This code introduces new syntax: the try/catch statement type. If throws is syntactically like return, try/catch is a bit like if/else.

A catch statement is properly a try/catch statement (or even more properly a try/catch/finally statement). If you are about to execute a statement that might throw an exception that you'd like to catch, you must first enter a try block. This is just like a regular block, except that it is preceded by the Java keyword try. This notifies Java that exceptions may be thrown and that it should be on the lookout for the ones that you want to catch.

At the end of the possible-exception-throwing code, you end the try block and introduce a catch clause. A catch clause contains a parameter declaration of the type that you wish to catch. The catch clause has a block that describes the instructions to execute if one of these is caught.

For example, the code in the user interface that is trying to connect transformerA (here named by to) and transformerB (here named by from) might say:

```java
try {
    new StringConnector( to, from );
} catch ( ConnectionRejectedException e ) {
    Console.println( "Sorry, can't make that connection. " + "Please try again." );
}
```

- This try/catch statement type has two bodies: one after the keyword try, and one after the catch parameter.
The try body is a statement or set of statements that may throw an exception. In this case, we know that the StringConnector constructor may throw ConnectionRejectedException. We can tell this from its declaration, and so can the Java compiler.

The catch portion of the statement has a single parameter, the exception (type and name) that is to be caught. In this case, the exception type is ConnectionRejectedException, and the name of the exception is e. The name is required, and it may be used inside the catch body, just like a method parameter name can be used inside the method body. It is common to name the exception e, though there's no particular reason for it; it's just like loop variables are often named i.

The catch body contains statements which are executed if and only if the appropriate type of exception is thrown. (The "appropriate type" is the type of the catch's parameter.) Inside the catch body, the parameter name may be used to refer to the exception, though there isn't a whole lot you can do with an exception other than print its message.

In this case, once the exception is caught, a message is printed to the user. This statement might itself appear inside an animate object's act method, so that something is continually listening to the user and trying to make connections on the user's behalf. This message lets the user know that this particular attempt didn't work. If we had supplied additional information along with the exception, we might use it at this point to give the user more information (perhaps flashing the object that refused the connection) or to try to repair the situation (asking whether the user means to delete the existing connection, for example, and then retrying the connection creation).

One try can actually have several catch statements. In this case, once something is thrown inside the try body, it is compared against the catch parameter statements in order until one that matches is found. If a match is found, only the first matching catch body is executed; then control continues at the end of the try/catch statement. If no match is found, the thrown object continues exiting statement blocks until a corresponding catch is found.

For completeness's sake, it is worth mentioning that a try/catch statement can have a finally clause (so that it's really try{}catch{}{}finally{}). In this case, no matter how the statement is exited -- regardless of whether something is thrown, and regardless of whether the thrown object is caught -- the finally statement will be executed. At this point, you shouldn't need to be using finally, but if you ever need to know, the gory details are included in the Java language specification.

## Try Statement Syntax

A try/catch/finally statement has a body after try, a body after each catch clause, and a body after the finally clause if it is present. Each of these bodies is a normal block executed according to the usual block execution rules. If a catch block is executed (i.e., if a matching throwable has been caught), the catch parameter is bound to the caught object during execution of the catch block.

The try body is a statement or set of statements that may throw an exception. Although not every execution of the try statement must throw an exception, the try statement must contain at least one expression that is declared as throwing each of the types of exceptions listed in its catch clauses.
Each `catch` clause has a single parameter (type and name) followed by a block. A catch clause matches the thrown object exactly when the thrown object can be named by a name of the catch clause's parameter type. Only the first matching catch clause is executed.

The `try` statement is executed as follows:

- The `try` block is executed in order until something is thrown or the end of the `try` body is reached.
- If nothing is thrown during the `try` body, execution continues after the final `catch` clause of the `try/catch` statement.)
- If something is thrown during the `try` body, it is compared against the parameter of each catch block, in turn, until a match is found. In this case, that catch block is executed (as a normal block) with the parameter bound to the (matching) caught object. At most one catch block of a try statement is executed.

A try statement may also have a single optional finally clause. This is the keyword finally followed by a block. If the try statement is entered, the finally clause is always executed. This leads to somewhat complicated execution rules, described below and further documented in Sun's Java Language Specification. Finally clauses are largely outside the scope of this book and are included here only for completeness.

The following two points explain the special behavior of try statements with finally blocks

- After execution of at most one matching catch block, execution proceeds at the finally block (if it is present). If a try statement is entered, its finally block is always executed, regardless of the execution within the try statement.
- If no uncaught exceptions remain on exiting the finally block, execution proceeds after the end of the `try/catch/finally` statement. If there is an outstanding thrown object, execution proceeds with the continued flight of that throwable.

**Throw vs. Return**

There are both similarities and differences between `throw` and `return` statement types. Both involve a single Thread following instructions that may take it from one method or constructor to another, often moving across multiple objects. From the perspective of the Thread, the objects (and their methods and constructors) are providing roles that it plays, scripts that it reads, or instructions that it follows.

When a Thread is executing some instructions and reaches a method invocation expression (or an instance creation expression), it carefully records its current place in the script, puts the current script down on the table in front of it, and picks up the invoked method script. If fulfilling that expression in turn involves a further invocation, yet another script will be added to the pile on the table. When an invocation completes, the Thread puts the corresponding script away and returns to the carefully marked pending method invocation (or instance creation) expression on top of the pile.

In other words, to clear off the pile, the Thread must pick up each script in order on its way out and complete any remaining instructions before going on to the next. Every method invocation or instance
creation expression eventually returns control to the body of code from which the call was invoked. The Thread eventually returns to the carefully marked spot and continues from there.

A major difference between return and throw statements is in how this execution proceeds, i.e., whether the Thread continues executing one instruction at a time or simply flies over the instructions looking for a matching `catch` statement. When a Thread returns normally from a method, execution continues one instruction at a time. When a Thread encounters a throw statement, it steps back through its pile of carefully marked scripts rather rapidly, scanning down the instructions until an appropriate `catch` statement is encountered. If the current script doesn't contain a matching `catch` statement, it is summarily discarded and the next script is examined in turn.

This means that a return statement always causes the current method to complete, returning control to whomever called this method. This is true no matter how many statement blocks the return is buried inside. *A return always exits exactly one method invocation.*

In contrast, a throw exits one block at a time until a `catch` of the appropriate type is found. This means that a throw may not exit any methods (if the throw occurs directly inside an appropriate `try/catch`), or the throw may exit many methods (if the exception is not caught in any of these calling methods). *A throw exits blocks until an appropriate catch is encountered.*

### Exceptions, Errors, and RuntimeExceptions

In Java, any instance whose class extends the class Throwable can be thrown and caught. Two special subclasses of Throwable are defined for use under exceptional circumstances.

**Error** is the Java class that denotes a catastrophic failure. This is an event from which your program is not expected to be able to recover. A well-designed robust program that is expected to have an extended lifetime (such as a banking system or an airline reservation system) must have ways of dealing with catastrophic failure, but most programs that you write will not have to worry about such circumstances.

**Exception** is the Java class that indicates a non-catastrophic failure. Exceptions are circumstances from which your program should recover (or at least exit gracefully).

All Java built-in error and exception classes have two constructors, one that takes no arguments and one that takes a single String argument. This String can be accessed using the `getMessage()` method of Error or Exception. If you define your own subclass, it is a good idea to define these two constructors there as well.

**RuntimeException** is a special subclass of Exception. Runtime Exceptions are circumstances from which your program should recover, but -- unlike for other Exceptions -- methods and constructors throwing RuntimeExceptions do not have to declare this fact.

All Exceptions other than_RUNTIMEExceptions are called checked exceptions. A method or constructor that may throw a checked exception must declare this fact, allowing the compiler to check for the presence of exception handlers. This can be very helpful in debugging, so you will generally want to extend Exception rather than RuntimeException.
When overriding a superclass method, a subclass method may only throw those checked exceptions also declared by the (overridden) superclass method. In other words, an overriding method may throw fewer things than promised by its superclass, but it may not throw additional (checked) things.

**Designing good test cases**

One of the most important parts of being a good programmer is knowing how to test your code. To begin this phase, write down all of the assumptions that your code makes. Think of something that violates one of these assumptions; will this break your code? How about something that violates three of these assumptions?

Once you have all of your assumptions written down, think about things that are extreme but within your assumptions. Try to design test cases for these. Think of every feature your code has, and every situation in which this feature could possibly be exercised. Design test cases for these as well. And don't forget the simple cases; it is always worth testing these as well as the pathological ones.

Your goal should be to thoroughly and exhaustively test your code, so you should design your test suite to exercise your program as fully as possible. You should also design test cases to catch bugs you think that other people might make. In particular, you should try to identify any weaknesses or difficult cases and design examples that stress these elements.

Finally, you should keep this test suite around, so that as you modify your code, you can test it again on these same examples, making sure that it still handles all of the old cases.

**Chapter Summary**

- In designing a program, you should anticipate things that can go wrong and design in mechanisms to deal with them.
  - Catastrophic failures cannot be prevented, but certain systems need to design in mechanisms to minimize the damage that they cause.
  - Some failures can be anticipated and avoided through simple checks and guards.
  - Other failures must be handled as they arise, often using Java's exception handling mechanisms.
- Exceptions should record information that is useful for addressing the problem as well as information that is useful for advising the debugger or the human user.
- When an exception is thrown by a method or constructor, it exits each enclosing block in turn until a matching catch statement is encountered.
- Methods and constructors that may throw checked exception types must declare this fact in their signatures.
- A method invocation or constructor that may throw a checked exception may be safely invoked within a try block with a corresponding catch statement. The catch statement is responsible for attempting to recover from the exception.

**Exercises**

http://www.cs101.org/ipij/exceptions.html
1. Describe the process of baking a cake. Include at least three exceptional circumstances that might arise and how these should be handled.

2. Describe the normal conduct of a soccer game. Include at least three exceptional circumstances that might arise and how these should be handled.

3. Define an Exception type called UnbelievableException. Remember to define two constructors.

4. Using your UnbelievableException type, write an animate object that continually asks the user for the user's age, then throws an UnbelievableException if appropriate. Note: the presence of an unbelievable age should not cause the program to terminate.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments: <webmaster@cs101.org>
Inheritance

Chapter Overview

- How do I simplify the program design task by reusing existing code?
- How do I create variants on things I already have?
- When is it not appropriate to reuse code?

This chapter covers class-based inheritance as a way to reuse implementation. Inheritance allows you to define a new class by specifying only the ways in which it differs from an existing class. Those differences can include: additional (or alternative) contracts that it satisfies, behaviors that it provides, internal information that it stores, or startup instructions. Inheritance means that existing code can be adapted and reused, with some modification, in new contexts.

The mechanism by which inheritance works involves extending the parent class definition either by augmenting or overriding behavior defined there. Most of this chapter concentrates on how these mechanisms work. Not every instance of similar behavior is an appropriate context for inheritance. The chapter concludes with a discussion of the limitations of inheritance.

This chapter includes sidebars on the details of method and field lookup. It is supplemented by reference charts on the syntax and semantics of java methods, fields, and class declarations.

Objectives of this Chapter

1. To understand how one class can build on behavior supplied by another.
2. To be able to extend and modify existing definitions
3. To recognize when to use mechanisms other than inheritance to extend behavior.

Derived Factories

We have so far seen several cases in which we wanted to build multiple kinds of things that shared a basic similarity. When this similarity was largely in the contract implemented -- as with Counters and Timers -- we abstracted this similarity into an interface. The interface allowed us to deal with objects without knowing the details of their implementations, i.e., to treat them solely in light of the contracts that they provided.

In this chapter, we are more concerned with situations in which two kinds of objects share not only the same contract but almost the same implementation. For example, the BasicCounter and the Resettable Counter contained almost precisely the same code. In fact, the BasicCounter's code was (except for the
class and constructor name) a proper subset of the Resettable Counter's code. Similarly, the code for AnimateObject was contained in the code for AnimateTimer and the code for CountingMonitor. And almost every StringTransformer simply elaborates on the generic StringTransformer, simply providing a specialized version of the transform() method.

In cases where code really matches at the level of wholesale textual reuse of a class, Java provides a mechanism to allow one type of object to build on the behavior specified by another. This is a relationship between one class and another. Since classes are essentially object factories, we can think of this as a situation in which one factory produces its widgets by buying widgets wholesale from another factory, then adding its own minor tweaks (bells and whistles) to the widgets before claiming to have produced them.

The mechanism by which this is accomplished in Java is called inheritance, and it applies to a relationship between two classes. There is a similar relationship between two interfaces, described below. Inheritance is not ever a relationship between a class and an interface (or between an interface and a class). Inheritance really means an almost literal subsuming of one thing by another.

**Simple Inheritance**

Consider, for example, the AnimateObject class from the previous chapter and its near relative, the CountingMonitor. The AnimateObject class says:

```java
public class AnimateObject implements Animate
{
    private AnimatorThread mover;

    public AnimateObject()
    {
        this.mover = new AnimatorThread( this );
        this.mover.startExecution();
    }

    public void act()
    {
        // what the Animate Object should do repeatedly
    }
}
```

In implementing the CountingMonitor class, we really only want to change the underlined things:

```java
public class CountingMonitor implements Animate
{
    private Counting whoToMonitor;

    private AnimatorThread mover;

    public CountingMonitor( Counting whoToMonitor )
    {
        this.whoToMonitor = whoToMonitor;
    }
}
```
It would be really nice only to have to write the underlined information, not the rest. In fact, we can do almost exactly that. The following definition is *almost* equivalent to the Counter definition above:

```java
public class CountingMonitor extends AnimateObject {

    private Counting whoToMonitor;

    public CountingMonitor ( Counting whoToMonitor )
    {
        this.whoToMonitor = whoToMonitor;
    }

    public void act()
    {
        Console.println( "The timer says "
                        + this.whoToMonitor.getValue() );
    }
}
```

We have preserved the underlining, and you can see that almost the entire new class is underlined. One of the few non-underlined items is the phrase extends AnimateObject. This is the phrase that does almost all of the work. It means, roughly, a CountingMonitor is an AnimateObject, it just provides the additional specified behavior.

1. It has its own private field, whoToMonitor, suitable for labelling a Counting.
2. It has a constructor that takes one argument, a Counting, and holds on to it.
3. Its act() method has a much more interesting body than AnimateObject's.

This code is equivalent to the original definition of CountingMonitor. It is much shorter to write. To use it, simply begin with the instructions for AnimateObject and add the pieces that CountingMonitor provides, *extending* the behavior of the AnimateObject (in the absence of conflicting instructions) to do these additional things.

In essence each CountingMonitor instance has an AnimateObject instance inside of it. Whenever the CountingMonitor can't figure out how to do something, it simply defaults to the behavior of its AnimateObject. That way, the CountingMonitor doesn't have to provide all of the behavior that an AnimateObject already has; it can just rely on the existing implementation.
The remainder of this chapter deals with the details of this proposition.

**java.lang.Object**

There is actually a single built-in type called `Object`, and all other object types (directly or indirectly) extend `Object`. In other words, anything which is not one of the built in types is an `Object` of some sort or another.

```java
class Cat extends Animal
{
   ....
}
```

A class declaration is followed by an optional `extends` clause, then a pair of braces around the body of the class definition. If the `extends` clause is missing (e.g., `class Widget { ... }`), the default clause `extends Object` is assumed. Thus, *every class* (implicitly or explicitly, directly or indirectly) extends `Object`.

The class `Object` provides some basic functionality that every other class necessarily inherits. This means that you can guarantee that every Java object has, e.g., a `toString()` method. See the sidebar on The class `Object` for details.

---

**The class Object**

The class `java.lang.Object` is the root of the inheritance hierarchy, i.e., the class of which all other classes are subclasses. Every Java object is guaranteed to implement each of the methods provided by `Object` (though their implementations may vary).

```java
boolean equals( Object ) returns true exactly when the argument Object is the same Object as the one whose method is invoked. This is exactly the same thing that == would do on two Objects. You may override equals to do something somewhat more interesting.

String toString() returns a String ostensibly suitable for printing. It contains a lot of useful information in a generally illegible format, so if you are interested in being able to read your objects, you may wish to override this method to print something more easily human-readable.

class getClass() returns the class object (i.e., factory) from which this instance was created.

Object clone() is a peculiar method of Object because although every object implements it, it can only be used with instances of classes that also implement the Cloneable interface. If a class implements the Cloneable interface, the inherited version of clone() simply creates a new object of the same type as the original and whose fields have the same values as the fields of the original. You may override clone() to do whatever you wish.[Footnote: If you call the clone() method of an object that doesn't implement Cloneable, it will throw CloneNotSupportedException. See the next chapter for more on Exceptions.]
Object also provides other methods (finalize, hashcode, wait, notify, and notifyAll) that are beyond the scope of the material covered here.

**Superclass Membership**

When one class extends another -- as in the CountingMonitor/AnimateObject example above, we say that the extending class (CountingMonitor) is a **subclass** of the extended class (AnimateObject), and that the extended class is a **superclass** of the extending class. Neither subclass nor superclass is an absolute description; instead, both describe relationships between two classes.

When we say that one class is a subclass of another, what we mean is that we can treat instances of the subclass in all respects as though they were members of the superclass. For example, we can use a CountingMonitor anywhere we can use an AnimateObject. We can assign a CountingMonitor to a name whose type makes it appropriate for labelling AnimateObjects. (After all, a CountingMonitor *is* an AnimateObject.) We can return a CountingMonitor from a method that expects to return an AnimateMonitor, or pass one as an argument to a method expecting an AnimateObject parameter. A CountingMonitor is simply a special kind of AnimateObject.

In fact, subclasses have all of the type-relational properties of classes and the interfaces that they implement. A subclass instance can be assigned to a name of the superclass type. It answers true to the `instanceof` predicate on the superclass. It can even be automatically coerced **up-cast** to its superclass type. This is the same kind of automatic coercion that happens from int to long, and it is similarly guaranteed always to succeed and never to lose information.

Treating a CountingMonitor as an AnimateObject doesn't actually change the CountingMonitor, though. The CountingMonitor is still a CountingMonitor, with its extended `act()` method and its Counting to keep track of. This is the same situation as when an object is treated according to its interface type: this narrows the view of the object, but it doesn't change the underlying object.

If you are currently holding what looks like a superclass instance (e.g., an AnimateObject), and you suspect that it is actually an instance of a subclass, you can attempt to do a **down-cast** coercion on it. As with primitive types, a narrowing conversion is one that may not work or may lose information.

For example, if AnimateObject `ao` has some value that you think might be a CountingMonitor, you can try the expression

```
(CountingMonitor) ao
```

(e.g., in an assignment statement or in a method invocation). However, if you're wrong and this AnimateObject is not a CountingMonitor, this will cause your program serious problems. (See the next chapter for information about how these problems arise and what you can do about them.) So you may want to test whether this is an OK thing to do first, using a **guard** expression:

```java
CountingMonitor cm;
if ( ao instanceof CountingMonitor )
{
```

This first checks to see whether it's OK to treat the AnimateObject as a CountingMonitor.

So far, we have seen that instances have several types: the type of the class from which the instance was created, the types of any interfaces that class implemented, and the types of any superclass that this class extends. This may mean many interface types (since a class can implement many interfaces). A class can only extend a single superclass, but this does not limit the number of legal class types because the superclass may itself extend another class, and so on. Where does this end?

We can use the idea of superclass membership to create very powerful abstractions, but not without the help of casting. For example, Java provides a class, Vector, that allows us to hold on to a collection of Objects; it behaves sort-of like a whole bunch of names, but indexed by number. Vector provides an addElement() method that takes any Object as an argument. This means that any Object can be inserted into a Vector. For example, you can insert a String into a Vector, and an AnimateObject as well:

```java
Vector v = new Vector();
v.addElement("Silly string");
v.addElement(new Timer());
```

However, when we retrieve the elements we've inserted, we discover that Vector's elementAt() method doesn't know the type of the Object we've inserted. Instead, elementAt() returns an Object; it is up to us to figure out what kind of thing we've gotten back. For example, the first thing in the Vector (at element 0) is the String "Silly string". So we can say

```java
Object o = v.elementAt(0);
```

or

```java
String s = (String) v.elementAt(0);
```

but not

```java
String s = v.elementAt(0);
```

because this is an illegal attempt to assign a value of type Object (v.elementAt(0)) to a name of type String. The explicit cast expression of the previous line is needed to make this statement legal.

**Overriding**

The examples of inheritance in the previous section demonstrated that a subclass can extend the functionality of its superclass. The subclass can also modify superclass functionality by overrriding, or redefining, methods provided by the superclass. In fact, CountingMonitor overrode the act() method provided by AnimateObject. This just wasn't a very interesting example because AnimateObject's act() method didn't do anything.

Consider the following classes:
public class Super 
{ 
    public void doit() 
    { 
        Console.println( "super method" ); 
    } 

    public void doitAgain() 
    { 
        this.doit(); 
    } 
}

public class OverridingSub extends Super 
{ 
    public void doit() 
    { 
        Console.println( "overridingSub method" ); 
    } 
}

Now suppose that we create an instance of OverridingSub and ask it to doit():

    OverridingSub over = new OverridingSub();
    over.doit();

As expected, this prints overridingSub method. What if we labelled the OverridingSub with a Super name?

    Super supe = new OverridingSub();
    supe.doit();

The same thing: overridingSub method Recall that using a different type of name doesn't change the underlying object.

super.

What if we still want to be able to access Super's doit() method from the subclass? To do this, we need a special expression much like this. The expression this refers to the instance whose code is being executed. The expression super refers to the superclass of the object containing the actual executing code.

public class ExpandingSub extends Super 
{ 
    public void doit() 
    { 
       super.doit();
       Console.println( "expandingSub method" );
    } 
}

In this case, we'll get the effect of executing the superclass method followed by the local println:
If we reverse the lines of the method body, we will reverse the order of the printed lines.

**Outside-in rule**

There is one more trick lurking in this example. This is the doitAgain() method in Super. We know what happens when we ask an instance of Super to doitAgain(): it does the same thing as if we'd asked it to doit(). But what if we ask a subclass instance?

```java
over.doitAgain()
```

The first thing that happens is that we have to find the doitAgain() method for OverridingSub. To do this, we start looking at the outermost (sub) class. This is OverridingSub. But it doesn't contain an appropriate method. So we move up the hierarchy, inside the object, to the superclass. Super does define doitAgain(), so now we know what code to execute. But the body of Super's doitAgain() method says this.doit(). Who is this?

The expression this always refers to the object on behalf of whom you are executing. At the moment, we're executing some code in the class Super. But we are doing it for an instance of OverridingSub; we just happen to be looking at over as though it were a Super, just as we did when we labelled it with a Super-type name. Looking at over as a Super doesn't make it one, though. So when we call this.doit(), we go right back to the outside (OverridingSub) and start working our way in again, looking for a doit() method. So the effect of invoking over.doitAgain() is the same as invoking over's doit(), not the Super method.

[Outside in pic]

**Problems with Private**

It isn't always completely straightforward to extend a class. Consider the BasicCounter and ResettableCounter classes from the chapter on Designing with Objects. Because the BasicCounter wasn't designed with inheritance in mind, there is a problem in extending it. In fact, we have to go back and modify the BasicCounter before we can describe the Resettable version directly in terms of it.

```java
class BasicCounter implements Counting {
    int currentValue = 0;

    void increment() {
        this.currentValue = this.currentValue + 1;
    }

    int getValue() {
        return this.currentValue;
    }
```
To implement the Resettable Counter class, we would like to be able to write the following:

```java
public class Counter extends BasicCounter implements Resettable {

    public Counter() {
        this.reset();
    }

    public void reset() {
        this.currentValue = 0;
    }

}
```

We have preserved the underlining, and you can see that almost the entire new class is underlined. This says that a Counter is just like a BasicCounter except:

1. It implements the Resettable interface (in addition to Counting, already implemented by -- and hence inherited from -- BasicCounter).
2. It has a no-args constructor that calls its own reset method.
3. It has a reset method that sets its currentValue field to 0.

But this code is not entirely adequate. In fact, it does not compile as is. The problem is that the currentValue field is not a part of the Counter class any more. The field currentValue is defined in BasicCounter. But BasicCounter's currentValue field is private, meaning that only BasicCounters (and the BasicCounter class, or factory) can access that field. The solution is to change the visibility of the field from private to protected. This allows the Counter subclass to access BasicCounter's currentValue field. Now, the Counter code in this chapter does the same thing as the Counter code in the Chapter on Designing with Objects.

The moral here is that if you want your class to be extensible -- to be able to be inherited from -- you will need to make sure that subclasses can get access to anything that they need to be able to manipulate. This in turn opens those aspects of your class up to manipulation by other classes, since that information is no longer private. The visibility level protected is an intermediate point between private and public, but it does not always provide adequate protection. For details, see the chapter on Abstraction.

**Constructors are Recipes**

We already know that constructors give the special instructions for how to create a particular kind of object. How does this interact with inheritance?

```java
this()
```
When a class has more than one constructor, we can express one constructor in terms of another using the special syntax `this()`. For example, we might define a `Point` class that either could be instantiated using specified values for the x and y coordinates or could take on the default value (0,0). We might define the constructors this way:

```java
public class Point {
    private int x, y;

    public Point() {
        this(0, 0);
        // constructor would continue here....
    }

    public Point(int x, int y) {
        ....
    }
}
```

The line `this(0, 0);` in the first (no-args) constructor means "create me using my other constructor and the arguments 0, 0". In other words, when we say `new Point()`, invoking the no-args constructor, this line transfers the responsibility of providing the instructions for the construction of the `Point` to the two-int constructor, supplying the ints 0 and 0 as values. Now, the second constructor would execute, creating a `Point`. This new `Point`'s construction process would continue in the first constructor at the comment

```
// constructor would continue here....
```

The point being constructed would be the point resulting from the second constructor's invocation on 0, 0. Since there are in fact no more instructions in the first constructor after the comment, execution of this constructor would terminate and the new point returned would be the point corresponding to (0, 0).

The special buck-passing constructor `this()` can only be used as the first line of a constructor.

**super()**

 Constructors and inheritance work similarly. Making an inherited object (the "inner object" that belongs to the superclass) is just like passing the buck to a same-class constructor. The first line of any constructor may be an explicit invocation of the superclass constructor, supplying whatever arguments are necessary between the parentheses.

For example, if we wanted to extend the `CountingMonitor` class, above, to determine whether the reading of its `Counting` had changed since the previous reading, we could add a field (to keep track of the previous reading) and a conditional in the `act()` method. But how would we deal with the constructor? The beginning of this class might read:

```java
public class ChangeDetectingCountingMonitor extends CountingMonitor {
```
private int previousReading;

public ChangeDetectingCountingMonitor( Counting who )
{
    super( who );
    // ....

The first line of this constructor says "create my inner CountingMonitor instance using who as its constructor parameter." When the superclass constructor completes its execution, the remainder of the ChangeDetectingCountingMonitor constructor body is executed, extending the CountingMonitor instance and wrapping it in whatever it needs to be a full-fledged ChangeDetectingCountingMonitor.

**implicit super()**

We have seen that, when no explicit constructor is supplied, Java blithely inserts a no-args constructor. Java actually has two dirty little secrets about constructors:

1. **If no constructor is provided for a class, Java automatically adds a no-arguments constructor.**
2. **Unless a constructor explicitly invokes its superclass constructor or another (this()) constructor of the same class, Java automatically inserts super(); as the first line of the constructor.**

This means that a class that doesn't seem to have a constructor actually has the following one:

```
public ClassName () {
    super();
}
```

What does this do? It means that you can create an instance of the class with `new ClassName()` -- because the constructor has no parameters, so you don't have to give it any arguments -- and it also means that each instance of `ClassName` has an instance of the superclass hiding inside it. That is, `super();` is a special incantation that means "Make me an instance of my superclass." (Be careful: there are two readings of this request: "Give me an instance..." and "Turn me into an instance...". The second reading is correct.)

The BasicCounter class has such an implicit, automatically inserted constructor, but the Counter class doesn't. Counter does automatically get the implicit call to `super();` though:

```
public BasicCounter () {
    super();
}
```

and

```
public Counter()
{
    super();
    this.reset();
}
```
You can, of course, insert this no-args make-me-an-instance-of-my-superclass constructor into every class definition, and some people like to do so explicitly.

Details:

1. `super();` may only appear as the first line of a constructor.
2. The form `super(args)` may be used if the superclass constructor takes arguments.
3. If a constructor is defined, this constructor is not automatically added. So, for example, `Echo does not` have a no-args constructor.
4. If a superclass does not have a no-args constructor, an explicit call to `super(args)` must be used as Java's automatic insertion of `super()` will cause a compile-time error.

What if a class doesn't have a superclass? *Every class is a subclass except* `Object`. *If a class doesn't have an extends in its declaration, Java automatically inserts* `extends Object`. That means that the automatically-inserted constructor will in general make sense.

**Beware:** Since Java will automatically invoke the no-args version of `super()` unless you explicitly invoke a superclass constructor, either (1) the superclass must *have* a no-args constructor or (2) you must explicitly invoke the superclass constructor yourself, supplying the requisite arguments. If you create a class without a no-args constructor, you can get into trouble extending it.

**Style Sidebar**

**Explicit use of this. and super()**

Although it is not strictly speaking necessary, it is good style to Use `this.` wherever it is appropriate, i.e., to denote calls to an object's own fields or methods. While it makes your code somewhat more verbose, it also makes it easier to read and to understand what's going on. No method call should ever be made without reference to its target (i.e., whose method is being called). Field accessor expressions should always include a reference to the field's owner, distinguishing them from other name accesses (including parameter and local variable references).

A class declaration that does not contain an explicit `extends` clause still `extends Object`. Stating this explicitly may make it easier to read your code.

A constructor that does not call another (`this()`) constructor explicitly calls the superclass constructor. If the superclass constructor is not invoked explicitly, Java will insert a(n implicit) call to `super()`, the superclass's no-args constructor. You can make this implicit call explicit by including `super();` as the first line of any constructor that doesn't explicitly invoke another self- or superclass constructor. This helps to remind you that it is being called anyway.

**Interface Inheritance**
A class cannot inherit from an interface; it implements the interface, providing behavior to match the interface's specification. But one interface can extend another. Interface inheritance is much simpler than class inheritance. In interface inheritance, the methods and fields of the inherited (super) interface are simply combined into the methods and fields of the inheriting (sub) interface. The syntax for interface inheritance is identical to the syntax for class inheritance, but since there can be no overriding of method specifications, and since all fields are public and static therefore cannot be overridden, there is really no complexity to interface inheritance.

As with class inheritance, if one interface extends another, all instances implementing the subinterface are instances belonging to both types.

**Relationships Between Types**

There are three different type-to-type relationships that will be important in creating systems. These three relationships correspond to three distinct mechanisms: implementation, extension, and coupling.

**Implementation** is a relationship in which one type provides a specification and a second type provides a specific way of implementing that specification. In this case, the first type is called an interface and the second type is called a class. For example, an Alarm is one way of implementing the Resetable specification; an Animation is another.

**Extension** is a relationship in which one type adds functionality to another. There are actually two variants of extension. In one, both types are specifications (i.e., interfaces) and the extending specification adds commitments to the extended specification. StartableAndResetable is an extension of Startable. In the other, both types are implementations (i.e., classes) and the extending implementation adds functionality to the extended implementation. A CheckingAccount adds check-writing functionality to a BankAccount. Extension is implemented using inheritance, the primary subject of this chapter.

**Coupling** is a way of giving one object the ability to ask another to help it. For example, a MicrowaveOven may have a Clock, but a MicrowaveOven isn't a Clock. MicrowaveOven doesn't implement Clock behavior or extend it. Each MicrowaveOven has a corresponding Clock, and when the MicrowaveOven needs to know what time it is, it checks with its own Clock. In this case, the relationship is one-to-one (one MicrowaveOven per Clock, one Clock per MicrowaveOven). There are other cases in which the relationship may be many-to-one (many Chickens, one Coop) or one-to-many. [IM: Unlike extension and implementation, coupling is really a relationship between instances; however, like implementation and extension, it is generally defined within the class.]

It is important to know which of these three relationships ought to hold as you design your code.

It is always advisable to factor out common commitments and to separate the users of these contracts from their implementors. Wherever possible, an object should be known by an interface type rather than a class type to make it possible for alternate implementations to be used. This is true for both name declarations and method return types. The only time when an interface cannot be used routinely is in a construction expression. [Footnote: But see, e.g., the Factory pattern [GHJV] for an approach to this problem.]

Interface implementation, the result of introducing these interfaces, is generally easy to recognize. An interface, after all, provides the contract without the actual implementation.
It is generally more difficult, especially for the novice programmer, to determine whether it is appropriate to use inheritance or merely containment. Inheritance is actually relatively rare (among classes) and should be used only when the new class really reuses the complete behavior of the existing class. This is because inheritance makes the implementation of the new class tremendously dependent on the details of the implementation of the existing class. Coupling is a much more general mechanism. In this case, the new kind of object simply relies on a previously existing kind of object to provide behavior, forwarding messages on to the instance of the pre-existing class. If the coupling relies on an interface type rather than on a class type, a different implementation can easily be substituted.

If you are constructing a class and want to make use of behavior implemented by another class, you must determine whether you are better off using inheritance (i.e., extension) or coupling. Here are some questions that you should ask:

- Does this new class present to its users the full range of behavior provided by the existing class (inheritance) or just some of that behavior (coupling)?
- Does this new class add behavior to the existing class (inheritance) or override it (coupling or a common subclass)?
- Can instances of this new class legitimately be treated as instances of the existing class (inheritance) or would this be inappropriate (coupling or common interface)?
- Does an instance of this new class have a different lifetimes from the associated instance of the existing class (coupling)?

It is only when the superclass will be wholly reused, and when the subclass really is an extension of the implementation provided by the superclass, that inheritance should be used. Occasionally, this justifies the use of an abstract class to encapsulate common behavior that is extended differently by different classes.

### Abstract Classes

A class can have a method that is just a signature -- an abstract method. In a class, however, the abstract method must be explicitly declared abstract. (Recall that methods in an interface are assumed to be abstract, even if they are not explicitly so declared.)

If a class has one or more abstract methods, it isn't a complete implementation. (It doesn't specify how to do the un-implemented method!) In this case you cannot directly make an instance of this class. (This is like a partial recipe -- you can't cook anything edible with it, but it may be useful in building more complete recipes. We will see how to use one recipe to build another in the chapter on Inheritance.)

A class with one or more abstract methods is called an abstract class. You cannot construct an instance of an abstract class. [Footnote: Technically, a class can be abstract even if it has no abstract methods. However, every class with at least one abstract method must be declared abstract.]

Abstract classes can be useful when you want to specify a partial implementation. You should not use an abstract class when you only want to specify a contract; that is the function of an interface.

We will see examples of abstract classes in later chapters.
Chapter Summary

- Inheritance is a mechanism that allows one class to reuse the implementation provided by another.
- Inheritance should be used only when instances of the subclass can also reasonably be considered instances of the superclass.
- A class always extends exactly one superclass. If a class does not explicitly extend another, it implicitly extends the class `Object`.
- Method lookup always begins with an object's actual (most specific sub)class, even when the method is invoked by a `this` expression in superclass code.
- A superclass method or (non-private) field can be accessed using a `super` expression.
- If a constructor does not explicitly invoke another (`this()` or `super()`) constructor, it implicitly invokes the superclass's no-args constructor.

Exercises

1. In the first interlude, we wrote "UpperCaser extends StringTransformer". Explain.

2. Extend the Counter to count by 2.

3. Complete the definition of `ChangeDetectingCountingMonitor` from above.

4. In this exercise, you will re-implement AnimateTimer in two different ways and then compare them.
   a. Re-implement Timer by extending Counter.
   b. Extend the class in the previous exercise by making it Animate.
   c. Now re-implement AnimateTimer by extending AnimateObject directly.
   d. What if any type relations would exist between an instance of the class produced in (b) and the class produced in (c)?

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Dealing With Difference: Dispatch

Chapter Overview

- How can I do different things at different times or under different circumstances?
- How can one method respond appropriately to many different inputs?

In previous chapters, we have looked at entities that respond to each input in roughly the same way. In this chapter, we will look at how an entity can respond differently depending on its input. In particular, we will look at how to build the central control loop of an entity whose job is to dispatch control to one of a set of internal "helper" procedures.

This chapter introduces several mechanisms for an entity to generate different behavior under different circumstances. Conditionals allow you to specify that a certain piece of code should only be executed under certain circumstances. This allows you to prevent potentially dangerous operations -- such as dividing by zero -- as well as to provide variant behavior.

The decision of how to respond often depends on the value of a particular expression. If there are a fixed finite number of possible values, and if the type of this expression is integral, we can use a special construct called a switch statement to efficiently handle the various options. A switch statement is often used together with symbolic constants, names whose most important property is that each one can be distinguished from the others.

Arrays are specialized collections of things. They allow you to treat a whole group of things uniformly. Arrays can be used to create conditional behavior under certain circumstances.

Procedural abstraction (covered in the next chapter) also plays a crucial role in designing good dispatch structures.

This chapter includes sidebars on the syntactic and semantic details of if, switch, and for statements, arrays, and constants. It is supplemented by portions of the reference chart on Java Statements.

Conditional Behavior

The animate objects that we have seen so far generally execute the same instructions over and over. A clock ticks off the time. A stringTransformer reads a string, transforms it, and writes it out. A web browser receives a url request, fetches, and displays the web page. And so on. These entities repeatedly execute what we might call a central control loop, an infinitely repeated sequence of action.

In this chapter, we look instead at entities whose responses vary from one input to the next, based on properties of that input. The actual responses are not the subject of this chapter; instead, we will largely
assume that the object in question has methods to provide those behaviors. The topic of this chapter is how the central control loop selects among these methods. This function -- deciding how to respond by considering the value that you have been asked to respond to -- is called dispatch.

Imagine that we are building a calculator. One part of the calculator -- its graphical user interface, or GUI -- might keep a list of the buttons pressed, in order. The central controller might loop, each time asking the GUI for the next button pressed. The primary job of this central control loop would be to select the appropriate action to take depending on what kind of button was pressed, and then to dispatch control to this action-taker. For example, when a digit button is pressed, the calculator should display this digit, perhaps along with previously pressed numbers.[Footnote: Pressing 6 right after you turn on a calculator is different from pressing 6 after pressing 1 right after you turn on a calculator. In the first case, the calculator displays 6; in the second, it displays 16.] Pressing an arithmetic function key -- such as + or * -- means that subsequent digits should treated as a new number -- the second operand of the arithmetic operator -- rather than as additional digits on the first. Pressing = causes the calculator to do arithmetic. And so on.

In this example, the calculator's central control loop is behaving like a middle manager. It's not the boss, who gets to set direction. It's not the worker, who actually does what needs to be done. The dispatcher is there to see that the boss's directions (the button pressed) get translated into the appropriate action (the helper procedure). The dispatcher is simply directing traffic. This kind of behavior, in which different things happen under different circumstances, requires conditional behavior. We have already seen a simple kind of conditional behavior using Java's if statement. In this chapter, we explore several different means of achieving conditional behavior in greater detail.

Throughout this chapter, we will assume that we have methods that actually provide this behavior. For example, the calculator might have a processDigitButton method which would behave like exercise # in Chapter 7. Another method, processOperatorButton, would apply the appropriate operation to combine the value currently showing on the calculator's display with the number about to be entered. We will also use methods such as isDigitButton to test whether a particular buttonID corresponds to a number key. Separating the logic surrounding the use of these operations from their implementation is an important part of good design and the topic of much of the chapter on Encapsulation.

In this chapter, we are going to concern ourselves with what comes after the first line of the calculator's act method:

```java
public void act()
{
    SomeType buttonID = this.gui.getButton();
    ....
}
```

The remainder of this method should contain code that calls, e.g., processDigitButton if buttonID corresponds to one of the buttons for digits 0 through 9, or processOperatorButton if buttonID corresponds to the button for addition. This chapter is about deciding which of these is the correct thing to do.

**If and else**
We have already seen the if statement, Java's most general conditional. Almost every programming language has a similar statement type. An if statement is a compound statement involving a test expression and a body that can include arbitrary statements. Any conditional behavior that can be obtained in Java can be accomplished using (one or more) if statements. An if statement corresponds closely to normal use of conditional sentences in every-day language. For example, "If it is raining out, take an umbrella with you" is a sentence that tells you what to do when there's rain. Note that this sentence says nothing about what to do if there is no rain.

**Basic Form**

Every if statement involves two parts: the test expression and the consequent statement. The test expression represents the condition under which the consequent should be done. The test expression is some expression whose type must be boolean. In our example sentence, this boolean expression is "it is raining out". This expression is either true or false at any given time. [Footnote: Excluding that sort of grey dreary drippy weather that haunts London and certain times of the year in Maine, of course.] making it a natural language analog to a true-or-false boolean. In Java, this expression must be wrapped in parentheses.

When an if statement is executed, this conditional expression is evaluated, i.e., its value is computed. This value is either true or false. The evaluation of the boolean test expression is always the first step in executing an if statement. The rest of the execution of the if statement depends on whether this test condition is true or false.

In the English example above, if "it is raining out" is true -- i.e., if it is raining out at the time that the sentence is spoken -- then you should take an umbrella with you. That is, if the condition is true, you should do the next part of the statement. This part of the if statement -- the part that you do if the test expression's value is true -- is called the consequent.

In Java, execution of an if statement works the same way. First, evaluate the boolean test. If the value of the test expression is true, then execute the consequent. If the value of the test expression is false, the consequent is not executed. In this case, evaluating the test expression is the only thing that happens during the execution of the if statement. Note that the value of the expression that matters is its value at the time of its evaluation. If the test is executed at two different times, it may well have two different values at those times.

In Java, the consequent may be any arbitrary statement (including a block). In this book, we will always assume that the consequent is a block, i.e., a set of one or more statements enclosed in braces.
Figure #@@. The execution path of an if statement.

We could write pseudo-code for our English conditional as follows:

```java
if ( currentWeather.isRaining() )
{
    take(umbrella);
}
```

This isn't runnable code, of course, but it does illustrate the syntax of a basic if statement: the keyword if, followed by a boolean expression wrapped in parentheses, followed by a block containing one or more statements. To execute it, we would first evaluate the (presumably boolean) expression `currentWeather.isRaining()` (perhaps by looking out the window) and then, depending on whether it is raining, either take an umbrella (i.e., execute `take( umbrella )`) or skip it.

A somewhat more realistic example is the following code to replace a previously defined number, $x$, with its absolute value:

```java
if ( x < 0 ) {
    x = - x;
}
```

This code does nothing just in case $x$ is greater than or equal to 0. [Footnote: It evaluates the expression $x < 0$, of course, but it "does nothing" that has any lasting effect.] If $x$ happens to be less than 0, the value of $x$ is changed so that $x$ now refers to its additive inverse, i.e., its absolute value.

Note that the same if statement may be executed repeatedly, and the value of the boolean test expression may differ from one execution of the if statement to the next. (For example, it may be raining today but not tomorrow, so you should take your umbrella today but not tomorrow.) The value of the boolean test expression is checked exactly once each time the if statement is executed, as the first step of the statement's execution.

**Else**

The if statement as described above either executes its consequent or doesn't, depending on the state of the boolean test expression at the time that the if statement is executed. Often, we don't want to decide whether (or not) to do something; instead, we want to decide which of two things to do. For example, if it's raining, we should take an umbrella; otherwise, we should take sunglasses. We could express this using two if statements:

```java
if ( currentWeather.isRaining() )
{
    take(umbrella);
}
if ( ! ( currentWeather.isRaining() ) )
```
Recall that `!` is the Java operator whose value is the boolean opposite of its single argument. So if `currentWeather.isRaining()` is true, then `!(currentWeather.isRaining())` is false; if `currentWeather.isRaining()` is false, then `!(currentWeather.isRaining())` is true.

These two conditional statements, one after the other, are intended to express alternatives. But they don't, really. For example, the two statements each check the boolean condition `currentWeather.isRaining()`. This is like looking out the window twice. In fact, the answer in each of these cases might be different. If we don't get around to executing the second if statement (i.e., looking out the window the second time) for a little while, the weather might well have changed and we'd find ourselves without either umbrella or sunglasses (or with both). The weather doesn't usually change that often (except in New England), but there are plenty of things that your program could be checking that do change that quickly. And, since your program is a community, it is always possible that some other member of the community changed something while your back was turned. [Footnote: But see chapter 20, where we discuss mechanisms to prevent the wrong things from changing behind your back.]

Instead of two separate if statements, we have a way to say that these two actions are actually mutually exclusive alternatives. We use a second form of the if statement, the if/else statement, that allows us to express this kind of situation. An if/else statement has a single boolean test condition but two statements, the consequent and the alternative. Like the consequent, the alternative can be almost any statement but will in this book be restricted to be a block.
Executing an if/else statement works mostly like executing a simple if statement: First the boolean test expression is evaluated. If its value is true, the consequent statement is executed and the if/else statement is done. The difference occurs when the boolean test expression's value is false. In this case, the consequent is skipped (as it would be in the simple if) but the alternative statement is executed in its place. So in an if/else statement, exactly one of the consequent statement or the alternative statement is always executed. Which one depends on the value of the boolean test expression.

The following code might appear in the calculator's act() method, as described above. It is looking at which button is pressed, just like a good manager, and deciding which helper procedure should handle it.

```java
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else
{
    this.processOperatorButton( buttonID );
}
```
This code presumes some helper functions. The method `isDigitButton` verifies that the `buttonID` corresponds to the keys 0 through 9. The `process...` methods actually implement the appropriate responses to these button types.

Because there is only one test expression in this statement, it is always the case that at the single time of its evaluation (per if statement execution), it will be either true or false. If the test expression is true, the consequent statement will be executed (and the alternative skipped). If it is false, the alternative statement will be executed (and the consequent skipped). Exactly one of the consequent or the alternative will necessarily be executed each time that the if statement is executed.

**Cascaded Ifs**

The if/else statement is a special case of a more general situation. Sometimes, it is sufficient to consider one test and decide whether to perform the consequent or the alternative. But the example we gave of determining whether the `buttonID` was a digit or not probably isn't one. After all, a non-digit might be an operator, but it also might, for example, be an `=`. We probably need to check more than one condition, although we know if any one of these conditions is true, none of the others is. This is a perfect situation for a cascaded if statement.[Footnote: The test for `isDigitButton`, etc., may seem mysterious right now, and indeed we will simply assume the existence of these boolean-returning predicates for now. An implementation is provided in the section on Symbolic Constants, below, and discussed further in the chapter on Encapsulation.]

```java
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
} else
{
    if ( this.isOperatorButton( buttonID ) )
    {
        this.processOperatorButton( buttonID );
    } else
    {
        this.processEqualsButton( buttonID );
    }
}
```

In fact, the situation is really even more complex:

```java
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
} else
{
    if ( this.isOperatorButton( buttonID ) )
    {
```
this.processOperatorButton( buttonID );
}
else
{

    if ( this.isEqualsButton( buttonID ) )
    {
        this.processEqualsButton( buttonID );
    }
    else
    {
        // and so on until...
        throw new NoSuchButtonException( buttonID );
    }
}

These if's inside else's can get to be quite difficult to read, not to mention the pressure that they put on the right margin of your code as each subsequent if is further indented. [Footnote: The final lines of such a sequence also contain an awful lot of closing braces.] In order to avoid making your code too complex -- and too right-handed -- there is an alternate but entirely equivalent syntax, called the cascaded if statement. In this statement, an else clause may take an if statement directly, rather than inside a block. Further, the consequent block of this embedded if statement is lined up with the consequent block of the original if statement. So the example above would now read

if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else if ( this.isOperatorButton( buttonID ) )
{
    this.processOperatorButton( buttonID );
}
else if ( this.isEqualsButton( buttonID ) )
{
    this.processEqualsButton( buttonID );
} // and so on until...
else
{
    throw new NoSuchButtonException( buttonID );
}

Note that instead of ending with many close braces in sequence, a cascaded if statement ends with a single else clause (generally without an if and test expression) followed by a single closing brace.
Like a simple if/else statement, exactly one block of a cascaded if statement is executed. Once that block executes, the entire statement is finished. The difference is that if the first expression's value is false, the next condition is evaluated, and then the next, and so on, until either

- one test expression evaluates to true, in which case the corresponding body is executed and execution of the statement is then terminated, or
- an else without an if and test is reached, in which case the corresponding body is executed, or
- the end of the statement is reached, in which case its execution is complete.

Since an else with no if and test is always executed, such an else must be the last clause of the cascaded if.

**Many Alternatives**

A conditional is a very general statement. With it, it is possible to write extremely convoluted programs. In order to make your program as easy to understand as possible, it is a good idea to keep your conditionals clean. A reasonable rule of thumb is that you should be able to explain the logic of your if statement easily to a friend. If you have to resort to pen and paper, your conditional expression may be too complex. If you have to write down more than two or three things, your conditional logic is most likely out of control.

For example, you should not test too many things simultaneously in one test expression. If you have a complex condition to test, use a boolean-returning method (a **predicate**) to keep the test expression simple. By naming the predicate appropriately, you can actually make your code much easier to read, as we did with isDigitButton and isOperatorButton, above. We will return to this point in the section on Procedural Abstraction in the chapter on Encapsulation.

As we have seen, you can embed if statements. In the example that we gave above, the embedded statements were actually mutually exclusive alternatives in the same set of tests: the button is either a digit
or an operator or the equals button or.... In this case, you should use the cascaded if syntax with which we replaced our embedded ifs.

But sometimes it is appropriate to embed conditionals. For example, in the calculator's act() method, inside the isOperatorButton block, we might further test whether the operation was addition or subtraction or multiplication or division.

```java
if ( this.isDigitButton( buttonID ) )
{
    this.processDigitButton( buttonID );
}
else if ( this.isOperatorButton( buttonID ) )
{
    if ( this.isPlusButton( buttonID ) )
    {
        this.handlePlus();
    }
    else if ( this.isMinusButton( buttonID ) )
    {
        this.handleMinus();
    }
    else if ( this.isTimesButton( buttonID ) )
    {
        this.handleTimes();
    }
    else if ( this.isDivideButton( buttonID ) )
    {
        this.handleDivide();
    }
    else
    {
        throw new NoSuchOperatorException( buttonID );
    }
}
else if ( this.isEqualsButton( buttonID ) )
{
    // etc.
}
```

In this case, these further tests are a part of deciding how to respond to an operator button, including an operator-specific exception-generating clause. Note that the additional tests appear inside an if body, not inside an unconditional else. Using an embedded conditional to further refine a tested condition is a reasonable design strategy.

**Beware of multiply evaluating an expression whose value might change.** Instead, evaluate the expression once, assigning this value to a temporary variable whose value, once assigned, will not change between repeated evaluations.

The example above of looking out the window to check the weather may work well in southern California, but it is ill-advised in New England, where the weather has been known to change at the drop of a hat. Similarly, repeated invocation of a method returning the current time can be expected to produce different
values. So can repeated invocations of a Counting's getValue method. If we execute the following conditional

```java
if ( theCounter.getValue() > 1 )
{
    Console.println( "My, there sure are a lot of them!" );
}
else if ( theCounter.getValue() == 1 )
{
    Console.println( "A partridge in a pear tree!" );
}
else if ( theCounter.getValue() == 0 )
{
    Console.println( "Not much, is it?" );
}
else if ( theCounter.getValue() < 0 )
{
    Console.println( "I'm feeling pretty negative" );
}
else
{
    Console.println( "Not too likely, is it?" );
}
```

it is possible that the counter will be incremented in just such a way that "Not too likely" might be printed.

Q. Describe how the process of executing this conditional might be intertwined with the incrementing of the counter to result in each of the five different values being printed. How might no value be printed?

**If Statement Syntax**

An if statement consists of the following parts:

- The keyword if, followed by
  - an expression of type boolean, enclosed in parentheses, followed by
  - a (block) statement.

This may optionally be followed by an else clause. An else clause consists of the following parts:

- The keyword else, followed by either
  - a (block) statement
  
  or

- the keyword if, followed by
  - an expression of type boolean, enclosed in parentheses, followed by
  - a (block) statement, optionally followed by
  - another else clause.
Execution of the if statement proceeds as follows:

First, the test expression of the if is executed. If its value is true, the (block) statement immediately following this test is executed. When this completes, execution continues after the end of the entire if statement, i.e., after the final else clause body (if any).

If the value of the first if test is false, execution continues at the first else clause. If this else clause does not have an if and condition, its body (block) is executed and then the if statement terminates. If the else clause does have an if test, execution proceeds as though this if were the first test of the statement, i.e., at the beginning of the preceding paragraph.

**Limited Options: Switch**

An if statement is a very general conditional. Often, the decision of what action to take depends largely or entirely on the value of a particular expression. For example, in the calculator, the decision as to what action to take when a user presses a button can be made based on the particular button pressed. What we really want to do is to see which of a set of known values (all of the calculator's buttons) matches the particular value (the actual button pressed). This situation is sometimes called a dispatch on case.

There is a special statement designed to handle just such a circumstance. In Java, this is a switch statement. A switch statement matches a particular expression against a list of known values.

Before we look at the switch statement itself, we need to look briefly at the list of known values. In a Java switch statement, these values must be **constant expressions**.

**Constant Values**

When we are choosing from among a fixed set of options, we can represent those options using symbolic constants. A symbolic constant is a name associated with a fixed value. For example, it would be lovely to write code that referred to the calculator's PLUS_BUTTON, TIMES_BUTTON, etc. But what values would we give these names? For that matter, what is the type of the calculator's buttonID?

The answer is that it doesn't matter. At least, it doesn't matter as long as PLUS BUTTON is distinct from TIMES BUTTON and every other buttonID on the calculator. We don't want to add PLUS BUTTON to TIMES BUTTON and find out whether the value is greater or less than EQUALS BUTTON, or to concatenate PLUS BUTTON and EQUALS BUTTON. But we do want to check whether buttonID == PLUS BUTTON, and the value of this expression ought to be (guaranteed to be) different from the value of buttonID == TIMES BUTTON (unless the value of buttonID has changed). Contrast this with a constant such as Math.PI, whose value is at least as important as its name.

These symbolic constants, then, must obey a simple contract. A particular symbolic constant must have the same value at all times (so that EQUALS BUTTON == EQUALS BUTTON, always), and its value must be distinct from that of other symbolic constants in the same group (PLUS BUTTON != EQUALS BUTTON). These are the ONLY guaranteed properties, other than the declared type of these names.

**Symbolic Constants**
It is common, though not strictly speaking necessary, to declare symbolic constants in a class or interface rather than on a per instance basis. It makes sense for them to appear in an interface when they form part of the contract that two objects use to interact. For example, you might communicate with me by passing me one of a fixed set of messages -- MESSAGE_HELLO, MESSAGE_GOODBYE, etc. -- and the interface might declare these constants as a part of defining the messages that we both are expected to understand and use. This means that these symbolic constants are declared static.

It makes sense that a name such as this, which is part of a contract, might be declared public. This allows it to be used by any objects that need to interact with the symbolic constant's declaring object. Symbolic constants like this need not be public, but they often are. (Private symbolic constants would be used only for internal purposes. Package-level or protected symbolic constants might be used in a restricted way.)

In Java, a name is declared final to indicate that its value cannot change. This is one of the properties that we want our symbolic constants to have: unchanging value. A value declared final cannot be modified, so you need not worry that extra visibility will allow another object to modify a constant inappropriately.

It is common, though somewhat arbitrary, to use ints for these constants. There are some advantages to this practice, and it does simplify accounting. For example, by defining a set of these constants in sequence one place in your code, it is relatively easy to keep track of which values have been used or to add new values.

```java
public static final int...
    PLUS_BUTTON = 10,
    MINUS_BUTTON = 11,
    TIMES_BUTTON = 12,
    ...
```

Of course, you should never depend on the particular value represented by a symbolic constant (such as EQUALS_BUTTON), since adding a new symbolic name to the list might cause renumbering. The particular value associated with such a name is not important.

So symbolic constants are often public static final ints.

**final**

In Java, a name may be declared with the modifier final. This means that the value of that name, once assigned, cannot be changed. Such a name is, in effect, constant.

The most common use of this feature is in declaring final fields. These are object properties that represent constant values. Often, these fields are static as well as final, i.e., they belong to the class or interface object rather than to its instances. Static final fields are the only fields allowed in interfaces.

In addition to final fields, Java parameters and even local variables can be declared final. A final parameter is one whose value may not be changed during execution of the method, though its value may vary from one invocation of the method to the next. A final variable is one whose value is unchanged during its scope, i.e., until the end of the enclosing block. [Footnote: final fields and parameters are not strictly
speaking necessary unless you plan to use inner classes. They may, however allow additional efficiencies for the compiler or clarity for the reader of your code.]

Java methods may also be declared final. In this case, the method cannot be overridden in a subclass. Such methods can be inlined (i.e., made to execute with especially little overhead) by a sufficiently intelligent compiler.

Java classes declared final cannot be extended (or subclassed).

Using Constants

Properties such as the button identifiers are common to all instances of Calculators. In fact, they are reasonably understood as properties of the Calculator type rather than of any particular Calculator instance. They can (and should) be used in interactions between Calculator's implementors and its users. In general, symbolic names (and other constants) can be a part of the contract between users and implementors.

This means that it is often useful to declare these static final fields in an interface, i.e., in the specification of the type and its interactions. In fact, static final fields are allowed in interfaces for precisely this reason. Thus, the definition of interfaces in chapter 4 is incomplete: interfaces can contain (only) abstract methods and static final data members.

For example, the Calculator's interface might declare the button identifiers described above:

```java
public interface Calculator
{
    public static final int PLUS_BUTTON = 10,
    MINUS_BUTTON = 11,
    TIMES_BUTTON = 12,
    ...
   _EQUALS_BUTTON = 27;
}
```

Now any user of the Calculator interface can rely on these symbolic constants as a part of the Calculator contract. For example, the isOperatorButton predicate might be implemented as

```java
public boolean isOperatorButton( int buttonID )
{
    return ( buttonID == PLUS_BUTTON )
    || ( buttonID == MINUS_BUTTON )
    || ( buttonID == TIMES_BUTTON )
    || ( buttonID == DIVIDE_BUTTON );
}
```

[Footnote: Note the absence of any explicit conditional statement here. Using an if to decide which boolean to return would be redundant when we already have boolean values provided by == and by ||. See the Sidebar on Using Booleans in the chapter on Statements.]

If we choose our numbering scheme carefully, the predicate isDigitButton could be implemented as
public boolean isDigitButton( int buttonID )
{
    return ( 0 <= buttonID ) && ( buttonID < 10 ) ;
}

Of course, this is taking advantage of the idea that the digit buttons would be represented by the corresponding ints. This is a legitimate thing to do, but ought to be carefully documented, both in the method's documentation and in the declaration of the symbolic constants:

/*
 * Symbolic constants representing calculator button IDs.
 * The values 0..9 are reserved for the digit buttons,
 * which do not have symbolic name equivalents.
 */
public static final int PLUS_BUTTON = 10,
    MINUS_BUTTON = 11,
    TIMES_BUTTON = 12,
    ... 
   _EQUALS_BUTTON = 27;

and

/*
 * Assumes that the digit buttons 0..9 will be represented by
 * the corresponding ints. These values should not be used for
 * other buttonID constants.
 */
public boolean isDigitButton( int buttonID )
{
    return ( 0 <= buttonID ) && ( buttonID < 10 ) ;
}

Style Sidebar

Use Named Constants

A constant is a name associated with a fixed value. Constants come in two flavors: constants that are used for their value, and symbolic constants, used solely for their names and uniqueness. Calculator.PLUS_BUTTON (whose value is meaningless) is a symbolic constant, while Math.PI (whose value is essential to its utility) is not. But constants -- named values -- are a good idea whether the value matters or not.

Introducing a numeric literal into your code is generally a bad idea. One exception is 0, which is often used to test for the absence of something or to start off a counting loop. Another exception is 1 when it is used to increment a counter. But almost all other numeric literals are hard to understand. In these cases, it is good style to introduce a name that explains what purpose the number serves.

Numbers that appear from nowhere, with no explanation and without an associated name, are sometimes called magic numbers (because they appear by magic). Like magic, it is difficult to know what kind of
stability magic numbers afford. It is certainly harder to read and understand code that uses magic numbers.

In contrast, when you use a static final name, you give the reader of your code insight into what the value means. Contrast, for example, EQUALS_BUTTON vs. 27. You also decouple the actual value from its intended purpose. Code containing the name EQUALS_BUTTON would still work if EQUALS_BUTTON were initially assigned 28 instead of 27; it relies only on the facts that its value is unchanging and it is distinct from any other buttonID.

Syntax

We turn now to a switch statement. A switch statement begins by evaluating the expression whose value is to be compared against the fixed set of possibilities. This expression is evaluated exactly once, at the beginning of the execution of the switch statement. Then, each possibility is compared until a match is found. If a match is found, "body" statements are executed. A switch statement may also contain a default case that always matches. In these ways, a switch statement is similar to, but not the same as, a traditional conditional.

Basic Form

A simple switch statement looks like this:

```java
switch ( integralExpression )
{
    case integralConstant:
        actionStatement;
        break;
    case anotherIntegralConstant:
        anotherActionStatement;
        break;
}
```

To execute it, first the integralExpression is evaluated. Then, it is compared to the first integralConstant. If it matches, the first actionStatement is executed. If integralExpression doesn't match the first integralConstant, it is compared to anotherIntegralConstant instead. The result is to execute the first actionStatement whose integralConstant matches, then jumps to the end of the switch statement.

For example, we might implement the calculator's act method like this:

```java
switch ( buttonID )
{
    case Calculator.PLUS_BUTTON:
        this.handlePlus();
        break;
    // ...
    case Calculator.EQUALS_BUTTON :
        this.handleEquals();
        break;
}
```
The presence of the break statements as the last statement of each set of actions is extremely important. They are not required in a switch statement, but without them the behavior of the switch statement is quite different. See the Switch Statement Sidebar for details.

### Break and Continue Statements

The `break` statement used here is actually more general that just its role in a `switch` statement.

A `break` statement is a general purpose statement that exits the innermost enclosing `switch`, `while`, `do`, or `for` block.

A variant form, the labelled `break` statement, exits all enclosing blocks until a matching label is found. A labelled `break` does not exit a method, however. The labelled form of the `break` statement looks like this:

```
label:

blockStatementText
{

  // body text
  break label;
  // more body text

} endBlockStatementText
```

One or both of `blockStatementText` or `endBlockStatementText` may be present; for example, this block may be a while loop, in which case `blockStatementText` would be the code fragment `while ( expr )` and there would be no `endBlockStatementText`. [Footnote: The labelled block may be any statement containing a block, including a simple sequence statement. The body text may contain any statements, including -- in the case of a labelled `break` -- other blocks, so that a labelled `break` may exit multiple embedded blocks.]

This code is equivalent to [Footnote: Here, `LabelBreakException` is a unique exception type referring to this particular labelled break statement.]

```
try
{

  blockStatementText
  {

    // body text
    throw new LabelBreakException();
    // more body text

  } endBlockStatementText

} catch ( LabelBreakException e )
```
That is, the labelled break statement causes execution to continue immediately after the end of the corresponding labelled block.

A similar statement, `continue`, also exists in unlabelled and labelled forms.

An unlabelled `continue` statement terminates the particular body execution of the (while, do, or for) loop it is executing and returns to the (increment and) test expression.

The labelled `continue` statement works similarly, except that it continues at the test expression of an enclosing labelled while, do, or for loop. The labelled `continue` statement

```java
label:

    blockStatementText
    {
        // body text
        continue label;
        // more body text
    } endBlockStatementText
```

is equivalent to

Figure #@@. Control flow diagrams for `break` and `continue` statements.
The Default Case

In an if statement, if none of the test expressions evaluates to true, a final else clause without an if and test expression may be used as the default behavior of the statement. Such an else clause is always executed whenever it is reached.

In a switch statement, a similar effect can be achieved with a special case (without a comparison value) labelled default:

```java
switch ( buttonID ) {
    case Calculator.PLUS_BUTTON:
        this.handlePlus();
        break;
    // ...
    case Calculator.EQUALS_BUTTON :
        this.handleEquals();
        break;
    default :
        throw new NoSuchButtonException( buttonID );
}
```

If no preceding case matches the value of the test expression, the default will always match. It is therefore usual to make the default the final case test of the switch statement. (No case after the default will be tested.) When the default clause is the last statement of your switch, it is not strictly speaking necessary to end it with a break statement, though it is not a bad idea to leave it in anyway. The final break; statement is omitted in this example because it would never be reached after the throw. (Any instruction follower executing the throw would exit the switch statement at that point.)

It is often a good idea to include a default case, even if you believe that it is unreachable. You would be amazed at how often "impossible" circumstances arise in programs, usually because an implicit assumption is poorly documented or because a modification made to one part of the code has an unexpected effect on another.
Variations

It is possible to write a switch statement without using breaks. In this case, when a case matches, not only its following statements but all statements within the switch and up to a break or the end of the switch statement will be executed. This can be useful when the action for one case is a subset of the action for a second case.

Beware of accidentally omitted break statements in a switch. Because omitting the break is sometimes what you want, it is legal Java and the compiler will not complain. Omitting a break statement will cause the statements of the following case(s) to be executed as well.

If two (or more) cases have the same behavior, you can write their cases consecutively and the same statements will be executed for both. This is, in effect, giving the first case no statements (and no break) and letting execution "drop through" to the statements for the second case. For example:

```
switch (buttonID)
{
    case Calculator.PLUS_BUTTON:
    case Calculator.MINUS_BUTTON:
    case Calculator.TIMES_BUTTON:
    case Calculator.DIVIDED_BY_BUTTON:
    case Calculator.ZERO_BUTTON:
        // perform actions
    default:
        // perform default actions
}
```
In this case statement, the same action would be taken for each of the four operator types. The buttonID pressed is passed along to the operator handler to allow it to figure out which operator is needed.

**Switch Statement Pros and Cons**

A switch statement is very useful when dispatch is based on the value of an expression and the value is drawn from a known set of choices. The switch expression must be of an integral type and the comparison case values must be constants (i.e., literals or final names) rather than other variable names. When a switch statement is used, the switch expression is evaluated only once.

A switch statement cannot be used when the dispatch expression is of an object type or when it is a floating point number. It also cannot be used with a boolean, but since the boolean expression has only two possible values, an if statement with a single alternative makes at least as much sense in that case.

The requirement that a switch expression must be of integral type is one reason why `static final` ints are often used as symbolic constants. `int` is a convenient integral type and symbolic constants are naturally compatible with switch statements.

A switch statement cannot be used when the comparison values are variable or drawn from a non-fixed set. That is, if the dispatch expression must be compared against other things whose values may change, the switch statement is not appropriate. For example, you wouldn't want to use a switch statement to compare a number against the current ages of the employees of your company, because these are changing values.

The switch statement is also not appropriate for expressions that may take on any of a large range of values. ("Large" is subjective, but if you wouldn't want to write out all of the cases, that's a good indication that you don't want a switch statement.) For example, you wouldn't want to do a dispatch on the title of a returned library book, testing it against every book name in the card catalog, even if you represented names as symbolic constants rather than as Strings.[Footnote: Of course, if you represented the names as Strings, you couldn't use a switch statement because String is an object type.]

**Switch Statement Syntax**

A switch statement contains a test expression and at least one case clause. After that, the switch statement may contain any number of case clauses or statements in any order:

```
switch ( integralExpression )
{
    caseClause caseClauses or statements
}
```

The `integralExpression` is any expression whose type is an integral type: `byte, short, int, long, or char`. 
A caseClause may be either

    case constantExpression :  

or

    default : 

If the caseClause contains a constantExpression, this must be an expression of an integral type whose value is known at compile time. Such an expression is typically either a literal or a name declared final, although it may also be an expression combining other constant expressions (e.g., the product of a literal and a name declared final).

Note that each caseClause must end with a colon.

The embedded statements may be any statement type [!!?!].

Typically, the actual syntax of a switch statement is

    switch ( integralExpression )
    { 
      caseClauses  
        statements ending with break;
        caseClauses  
        statements ending with break;
      ...
      default :  
        statements optionally ending with break;
    } 

where caseClauses is one or more case clauses.

Arrays

Sometimes, what we really want to do when dispatching is to translate from one representation to another. For example, in constructing a Calculator, we might want to move from the symbolic constants used to identify buttons above to the actual labels appearing on those buttons. We might even want to move between the labels on buttons and the buttons themselves. If our collection of objects is indexed using an integral type -- either because it is naturally indexed or because we have used ints as symbolic constants -- we can often accomplish this conveniently using arrays.
What is an Array?

An array is an integrally indexed grouping of shoeboxes or labels. You can think of it sort-of like a wall full of numbered mailboxes. In identifying a mailbox, you need to use both a name corresponding to the whole group ("the mailboxes in the lobby") and an index specifying which one ("mailbox 37"). Similarly, an array itself is a thing that can be named -- like the group of mailboxes -- and it has members -- individual mailboxes -- named using both the array name and the index, in combination. For example, my own particular individual mailbox might be named by `lobbyMailboxes[37]`.

An array has an associated type that specifies what kind of thing the individual names within the array can be used to refer to. This type is sometimes called the base type of the array. For example, you can have an array of chars or an array of Strings or an array of Buttons. The individual names within the array are all of the same type, say `char` or `String` or `Button`.

That is, an array is a collection of nearly-identical names, distinguished only by an int index. An array of shoebox-type -- for example, an array of chars -- really is almost like a set of mailboxes, each of which is an individual shoebox-name. To identify a particular shoebox, you give its mailbox number. For example, you can look and see what (char) is in mailbox 32 or put an appropriately typed thing (char) in mailbox 17. Label-type arrays work similarly, though it's hard to find an analogously appropriate analogy. (A set of dog-tags or post-it notes is along the right lines, but it is harder to visualize these as neatly lined up and numbered.) A label-type array -- such as an array of Buttons - is an indexed collection of labels suitable for affixing on things of the appropriate type -- such as Buttons. The names affixed on individual Buttons are names like `myButtons[8]`, the ninth button in my array. [Footnote: Yes, that's right, `myButtons[8]`, the ninth button. Array elements, like the characters in Strings, are numbered starting from 0.]

Figure #@@. An array is like a wall of numbered mailboxes.

Figure #@@. This array, named `labels`, has eight elements, named `labels[0]` through `labels[7]`. Note the difference between what's attached to `labels` and what's attached to `labels[0]`. 
Arrays of primitive (dial) types are similar, but not exactly the same as label-type arrays. This array, named `dials`, has eight elements, named `dials[0]` through `dials[7]`. Note the difference between the value of `dials` (which is actually a label!) and the value of `dials[0]`.

**Array Declaration**

An array type is written just like the type it is intended to hold, followed by square braces. For example, the type of an array of chars is `char[]` and the type of an array of Buttons is `Button[]`. Note that, like `char` and `Button`, `char[]` and `Button[]` denote types, not actual Things. So, for example,

```java
char[] initials;
```

makes the name `initials` suitable for sticking on things of type `char[]`; it doesn't create anything of type `char[]` or otherwise affix `initials` to some Thing. Similarly,

```java
Button[] pushButtons;
```

creates a label, `pushButtons`, suitable for attaching to a `Button[]`, and nothing more. Note that both `initials` and `pushButtons` are *label* names, not *shoebox* names. The names of array types are always label types, although a particular array may itself be suitable either for holding shoebox (e.g., `char`) or label (e.g., `Button`) types.
Array Construction

To actually create a `char[]` or `Button[]`, you need an array construction expression. This looks a bit like a class instantiation expression, but it is actually not quite the same. An array construction expression consists of the keyword `new` followed by the array type with an array size inside the square braces. For example,

```
new char[26]
```

is an expression that creates 26 char-sized mailboxes, numbered 0 through 25. Similarly,

```
new Button[ 518 ]
```

is an expression whose value is a brand new array of 518 Button-sized labels. Note that arrays are indexed starting at 0, so the last index of a member of this array will be 517, one less than the number supplied to the array construction expression. [Footnote: An array construction expression can be passed any expression with integral type (byte, short, int, long, or char) and its size and indexing will be set accordingly.]

The expression

```
pushButtons = new Button[ numButtons ]
```

makes the name `pushButtons` refer to a new array of Button-sized labels. How many? That depends on the value of `numButtons` at the time that this statement is executed.

The statement

```
String[] buttonLabels = new String[16];
```

combines all of these forms, creating a name (buttonLabels) suitable for labeling an array of Strings (String[]), constructing a 16-String array, and then attaching the name buttonLabels to that array. Note that the text String[] appears twice in this definition, once as the type and once (with an integral argument between the brackets) in the array construction expression.

Array Elements

To access a particular member of the array, you need an expression that refers to the array (such as its name), followed by the index of the particular member inside square braces. For example,

```
buttonLabels[2]
```

is an expression of type String that refers to the element at index 3 of the String array named by buttonLabels. Recall that, since the indices of buttonLabels run from 0 to 15, buttonLabels[2] is the third element of the array.

This expression behaves very much as though it were a name expression. Like a name, an array element expression of label type may be stuck on something, or may be null. An array element of shoebox type (e.g., initials[6]) behaves like a shoebox name.
You can use these array member expressions in any place you could use a name of the same type. So, for example, you can say any of the following things:

```java
buttonLabels[2] = "Hi there";
String firstString = buttonLabels[0];
Console.println( buttonLabels[ Calculator.PLUS_BUTTON ] );
if ( buttonLabels[ currentIndex ] == null ) ...
```

(assuming of course that `Calculator.PLUS_BUTTON` and `currentIndex` are both int names).

**Array Syntax**

**Array Type**

An array is a label name whose type is any Java type followed by `[]`. The array is an array of *that type*. Admissible types include shoebox (primitive) types, label (object) types, and other array types. An array is declared like any other Java name, but using an array type. For example, if `baseType` is any Java type, then the following declaration creates a label, `arrayName`, suitable for affixing on an array of `baseType`:

```java
baseType[] arrayName;
```

**Array Initialization**

By default, an array name's value is null. An array name may be defined at declaration time using an array literal. This consists of a sequence of comma-separated constant expressions enclosed in braces:

```java
baseType[] arrayName = { const0, const1, ... constN };
```

[Check this: literals only, or also symbolic constants? Can this be done w/non-String object types?]

**Array Construction**

Unless an array initialization expression is used in the declaration, an array must be constructed explicitly using the array construction expression

```java
new baseType[ size ]
```

Here, `baseType` is the base type of the array (i.e., this expression constructs an array of `baseType`) and `size` is any non-negative integral expression.

**Array Access**
The expression `arrayName[index]` behaves as a "regular" Java name. Its type is the array's base type.

Arrays are numbered from 0 to `arrayName.length - 1`. Attempting to access an array with an index outside this range throws an `ArrayOutOfBoundsException`.

**Manipulating Arrays**

The particular names associated with individual members of an array behave like ordinary (shoebox or label) names. What is unusual about them is how you write the name -- `arrayName[index]` -- and not any of how they actually behave.

You can find out how many elements are in a particular array with the expression `arrayName.length`. Note that there are no parentheses after the word length in this expression. Technically, this is not either a field access or a method invocation expression, although it looks like one and behaves like the other.

Note also that the value of the expression `arrayName.length` is *not* the index of the last element of the array. It is in fact one more than the final index of the array, because the array's indices start at 0. Attempting to access an array element with a name smaller than 0 or greater than or equal to its length is an error. In this case, Java will throw an `ArrayOutOfBoundsException`. 
Once you construct an array, the number of elements in that array does not change. However, this immutable value is the number of elements in the array itself, not the number of elements associated with the name. If the name is used to refer to a different array later, it may have a different set of legal indices. For example:

```java
char[] firstInitials = new char[10];
firstInitials[5] = 'f';
firstInitials[5] = 'g';
   // changes the value associated with a particular mailbox
firstInitials = new char[2]
   // changes the whole set of mailboxes
   // now pushButtons[3] isn't legal either!
```

Stepping through An Array Using a for Statement

One common use of arrays is as a way to step through a collection of objects. If you are going to work your way through the collection, one by one, it is common to do so using a counter and a loop.

We can write this with a while loop:
int index = 0;

while (index < array.length) {
   // do something
   index = index + 1;
}

Note that index can't be initialized inside the while statement or it wouldn't be bound in the test expression. Local (variable) names have scope only from their declarations until the end of their enclosing blocks.

This is so common, there's a special statement for it. The while statement above can be replaced by

for (int index = 0; index < array.length; index = index + 1) {
   // do something
}

Note that the for loop also includes the declaration of index, but that index only has scope inside the for loop. It is as though index's definition plus the while loop were enclosed in a block.

For additional detail on for statements, refer to the sidebar.

For Statement Syntax

The syntax

    for ( initStatement; testExpression; incrementStatement )
    {
        body
    }

is the same as

    {
        initStatement;

        while ( testExpression )
        {
            body
            incrementStatement;
        }
    }

The expression testExpression is any single boolean expression. It falls within the scope of any declarations made in initStatement.
Both `initStatement` and `incrementStatement` are actually allowed to be multiple statements separated by commas:

e.g.

\[
i = i + 1, \ j = j + i
\]

Note that `initStatement`, `testExpression` and `incrementStatement` are separated by semicolons, but that individual statements within `initStatement` and `incrementStatement` are separated by commas. There is no semicolon at the end of `incrementStatement`.

### Using Arrays for Dispatch

In addition to their use as collection objects, arrays can be used as a mechanism for dispatch. This is because the same variable can be used to index into multiple arrays or be passed to appropriate methods. We are not going to use an array to do the calculator's central dispatch job right now. Instead, we will consider the problem of constructing actual GUI Button objects that will appear on the screen. There should be one Button corresponding to each of the symbolic constants described above. Each of these Buttons will need an appropriate label, to be passed into the Button constructor. We might create a method, `String getLabel( int buttonID )` for this purpose.

We could use our `getLabel` to say

\[
\text{new Button( this.getLabel( buttonID ) )}
\]

or even

\[
\text{gui.add( new Button( this.getLabel( buttonID ) ) )}
\]

Such a `getLabel` method, which could translate from `buttonIDs` to labels, would also be useful for generating Strings suitable for printing to the Console, e.g., for debugging purposes.

One way to implement this method would be with an if statement. In this case, the body of the method might say:

```java
if ( buttonID == Calculator.PLUS_BUTTON )
{
    return "+";
}
else if ( buttonID == Calculator.MINUS_BUTTON )
{
    return "-";
}
else if ( buttonID == Calculator.TIMES_BUTTON )
{
    // and so on....
```
Of course, this would get rather verbose rather quickly.

Because we are really doing a dispatch on the value of buttonID, and because we've cleverly chosen to implement these symbolic constants as ints, we could opt instead to use a switch statement:

```java
switch (buttonID)
{
    case Calculator.PLUS_BUTTON :
        return "+";
    case Calculator.MINUS_BUTTON :
        return "-";
    // and so on....
}
```

This may be somewhat shorter, but not much. It does have the advantage of making the dispatch on buttonID more explicit. But we can do still better.

Q. In the immediately preceding switch statement, why are there no break statements?

If we create an array containing the button labels, in order, corresponding to the buttonID symbolic constants, then we can use the buttonID to select the label:

```java
String[] buttonLabels = { "0", "1", "2", "3", "4",
    "5", "6", "7", "8", "9",
    "+", "-", "+", "/",
    // and so on...up to
    "="};
```

In this case, the entire body of our getLabel method might say simply

```java
return this.buttonLabels[buttonID];
```

This example is relatively simple, but in general arrays can be used whenever there is an association from an index set (such as the buttonIDs) to other values. The idea is that the index pulls out the correct information for that particular value. This is a very simple form of a very powerful idea, which we shall revisit in the chapter on Object Dispatch.

**When to Use Which Construct**

Arrays are in many ways the most limited of the dispatch mechanisms. They work well when the action is uniform up to some integrally indexed decisions, e.g., some integrally indexed variables need to be supplied. Setting up the array appropriately allows for very concise code. This is not always possible, though, either because there isn't an obvious index set, because the index set is not integral, because it is not possible to set up the necessary association, or because the needed responses are nonuniform.

Switch statements also rely on integrally indexed decisions on a single expression, but they are otherwise quite general in the action(s) that can take place. They are useful any time the decision is made by testing the expression against a pre-known fixed set of constants. In other words, a switch statement can be used whenever an array is appropriate, though it may be more verbose. A switch statement can also be used in cases of nonuniform response, where an array would not be appropriate.
Ifs are very general. You can do anything with them. You should use them when none of the other mechanisms are appropriate.

In a subsequent chapter, we will see an additional dispatch mechanism, object dispatch, that resembles the implicit nature of array-based dispatch, but without many of its restrictions.

Chapter Summary

- Dispatch is the process of deciding what action needs to be taken based on one's input. It is essentially a middle management function.
- Conditional statements are used when a piece of code should be executed under some but not all circumstances.
  - An if statement may consist only of a single boolean test expression and a body. This body is executed only if the test expression's value is true.
  - An if statement may optionally have an else clause with a body that is executed only when the if's test expression has the value false.
  - The else clause of an if statement may itself be an if statement. In this case, it is preferable to use cascaded rather than embedded ifs.
  - Each test expression is evaluated independently as it is reached.
- Numbers generally should not appear in code. Instead, use symbolic constants with descriptive names.
- A switch statement is used when different actions must be taken depending on the value of a single expression.
  - This expression is evaluated only once. Its type must be integral.
  - In a switch expression, the value is compared against different cases, which must be constants. Once a case matches, the statements of the switch body are executed until either a break or the end of the switch body is reached.
  - Switch has a specialized case, default, which always matches.
- An array is a uniformly typed collection of names.
  - The type of the array member names is the array's base type. The array member names may be either shoebox names or label names, depending on the base type.
  - The type of the array is "array of base type". The array name is a label name.
  - The names of array members are written using the array name followed by an integral index enclosed in square brackets.
  - The indices of an array run from 0 to \( arrayName.length - 1 \).
  - Like an object, an array must be explicitly created using new.

Exercises

1. In the section entitled "Many Alternatives", there is an example of a counter whose getValue() method is invoked repeatedly.
a. Describe an execution sequence in which the value printed would be "My, there sure are a lot of them!"

2. Describe an execution sequence in which the value printed would be "A partridge in a pear tree!"

3. Describe how the process of executing this conditional might be intertwined with the incrementing of the counter to result in the printing of none of the messages.

2. Convert the following to a for loop:

```java
int sum = 0;
int i = 1;
while ( i < MAXIMUM )
{
    sum = sum + i;
    i = i + 2;
}
```

3. Write a method that takes an array of ints and returns the sum of these ints.

4. Suppose that you have access to an array of StringTransformers, each of which has a method satisfying String transform( String ). Write a method, produceAllTransformations, that takes in a String and returns an array of Strings. The first element of the returned array should correspond to the transformation of the argument String by the first transformer, the second to the transformation of the argument String by the second transformer, and so on. You may assume that the name of the array of StringTransformers is transformerFunctions.

5. Consider the following code, excerpted from the definition of class EmotionalSpeaker.

```java
public String transformEmotionally( Type emotion, String what )
{
    switch ( emotion )
    {
    case HAPPY:  return sayHappily( what );
    case SAD:    return saySadly( what );
    case ANGRY:  return sayAngrily( what );
    }
}
```

Where, e.g.,

```java
private String sayHappily( String what )
{
    return "I'm so happy that ";
}
```

(You may assume similar definitions for the other emotions, with appropriate modifications.)
Define the symbolic constants HAPPY, SAD, and ANGRY, and provide a type for emotion.

6. In the previous exercise, the switch statement contains no breaks. What happens when we invoke `transformEmotionally( SAD, "I am here." )`?

7. Using an array, modify the code for `transformEmotionally` so that it fits in a single line. The array definition need not fit on that line.
Encapsulation

Chapter Overview

- How do I package up implementation details so that a user doesn't have to worry about them?
- How do I make my code easier to read, understand, modify, and maintain?

Good design separates use from implementation. Java provides many mechanisms for accomplishing this. In this chapter, we review a variety of mechanisms that allow this sort of separation.

Procedural abstraction is the idea that each method should have a coherent conceptual description that separates its implementation from its users. You can encapsulate behavior in methods that are internal to an object or methods that are widely usable. Methods should not be too complex or too long. Procedural abstraction makes your code easier to read, understand, modify, and reuse.

Packages allow a large program to be subdivided into groups of related classes and instances. Packages separate the names of classes, so that more than one class in a program may have a given name as long as they occur in different packages. In addition to their role in naming, packages have a role as visibility protectors. Packages provide visibility levels intermediate between public and private. Packages can also be combined with inheritance or with interfaces to provide additional encapsulation and separation of use from implementation.

Inner classes are a mechanism that allows one class to be encapsulated inside another. Perversely, you can also use an inner class to protect its containing class or instance. Inner classes have privileged access to the state of their containers, so an inner class can provide access without exposing the object as a whole.

Objectives of this Chapter

1. To understand how information-hiding benefits both implementor and user.
2. To learn how to use procedural abstraction to break your methods into manageable pieces.
3. To be able to hide information from other classes using visibility modifiers, packages, and types.
4. To recognize inner classes.

Design, Abstraction, and Encapsulation

This chapter is about how information can be hidden inside an entity. There are many different ways that this can be done. Each of these is about keeping some details hidden, so that a user can rely on a
commitment, or contract, without having to know how that contract is implemented. There are numerous benefits from such information hiding.

First, it makes it possible to use something without having to know in detail how it works. We do this all the time with everyday objects. Imagine if you had to understand how a transistor works to use your computer, or how a spark plug works to use your car, or how atoms work to use a lever.

Second, information-hiding gives some flexibility to the implementor. If the user is not relying on the details of your implementation, you can modify your implementation without disturbing the user. For example, you can upgrade your implementation if you find a better way to accomplish your task. You can also substitute in different implementations on different occasions, as they may become appropriate.

Finally, hiding information is liberating for the user, who does not expect nor make great commitment to particulars of the implementation. The name for this idea -- of using more general properties to stand in for detailed implementation -- is abstraction. To facilitate abstraction, it is often convenient to package up the implementation details into a single unit. This packaging-up is called encapsulation.

**Procedural Abstraction**

Procedural abstraction is a particular mechanism for separating use from implementation. It is tied to the idea that each particular method performs a well-specified function. In some cases, a method may calculate the answer to a particular question. In others, it may ensure the maintenance of a certain condition or perform a certain service. In all cases, each method should be accompanied by a succinct and intuitive description of what it does.[Footnote: It is not, however, essential that a method have a succinct description of how it does what it does. How it accomplishes its task is an implementation detail.] A method whose function is not succinctly describable is probably not a good method. Conversely, almost every succinctly describable function should be a separate method, albeit perhaps a private or final one.

This idea, that each conceptual unit of behavior should be wrapped up in a procedure, is called procedural abstraction. In thinking about how to design your object behaviors, you should consider which chunks of behavior -- whether externally visible or for internal use only -- make sense as separate pieces of behavior. You may choose to encapsulate a piece of behavior for any or all of the following reasons:

- It's a big, ugly function and you want to hide the "how it works" details from code that might use it. Giving it a name allows the user to ignore how it's done.
- It's a common thing to do, and you don't want to have to replicate the code in several places. Giving it a name allows multiple users to rely on the same (common) implementation.
- It's conceptually a separate "task", and you want to be able to give it a name.

Note also that the behavior of a method may vary slightly from invocation to invocation, since the parameters can influence what the code actually does.

**The Description Rule of Thumb**

Each method in your program should have a well-defined purpose, and each well-defined purpose in your program should have its own method. You should be able to succinctly state what each method in your
program does. If you cannot, your methods are either too large (i.e., should be broken into separable conceptual units) or too small (i.e., should be combined so that each performs a "complete" task.

Note that having a succinct description of what a method does is quite different from being to state succinctly how it accomplishes this. It is unfortunately all too common that a method's implementation is obscure. It is important that the user understand when, why, and under what circumstances your method should be used, i.e., what it does. You provide a method precisely so that the user will not have to understand how your method works.

For example, it is common to test complex conditions using a single predicate. One such instance might be the Calculator's `isDigitButton()` method, which determines whether a particular Calculator button represents the digits 0 through 9 (or instead is, e.g., an arithmetic operator). The logic behind `isDigitButton()` might be somewhat obscure. However, it is easy to succinctly state what the method determines and, therefore, when and why you might use it. This use of predicates as abstractions make code for easier to read, decompose, and understand.

The importance of succinct summarizability does not mean that there is exactly one method per description. For example, one succinctly summarizable method may in turn rely on many other succinctly summarizable methods. This is the "packaging up substeps" idea from Chapter 1: making a sandwich may be described in terms of spreading the peanut butter, spreading the jelly, closing and cutting the sandwich. Each substep may itself be a method. When the substeps are not likely to be useful for anything except the larger method of which they are a part, these methods should be private to their defining class.

It may also be the case that multiple methods each implement the same well-defined purpose. For example, multiple similar methods may operate on different kinds of arguments. A method that draws a rectangle may be able to take a `java.awt.Rectangle`, two `java.awt.Points`, or four ints as arguments. Each of these methods will have a different signature. They may, however, rely on a common (shared) method to actually perform much of the work, sharing as much code as possible. (See the repetition rule of thumb, below.)

Or it may be the case that multiple distinct object types each have similar methods performing similarly summarized functions. In this case, it may make sense to have a common interface implemented by each of these classes, documenting their common purpose. Occasionally it even makes sense to split off the method into its own class, turning instances of the new class into components of the old. (See the discussion of using contained objects in the chapter on Object Oriented Design.)

When a single method does too many things, it can be difficult to decide whether you want to invoke it. It can be awkward to figure out what it is really doing. And the interdependencies among subtasks can make your code hard to maintain, especially if the assumptions that caused you to bundle these pieces together no longer hold.

Succinct summarizability makes your code immensely easier to read. By choosing descriptive names, you can often make your code read like the English description of what it does. This makes it easier to read, understand, modify, and maintain your code.

**The Length Rule of Thumb**
A single method should ideally fit on a single page (or screen). Often a method will only be a few lines long. If you find yourself writing longer methods, you should work on figuring out how to break them up into separable substeps. The description rule of thumb is handy here.

When a method's implementation takes up too much space, it is difficult to read, understand, or modify. It can be hard to hold the whole method in your head. It can be overwhelming to try to figure out what it is actually doing.

Appropriate method length is a matter of some individual judgment. Some people don't like to write methods longer than a half-page. Others regularly write much longer methods. As you become a more skilled programmer, you will become accustomed to keeping track of larger and more complex programs. But more complex programs do not mean longer methods. It will always be the case that brevity of individual units -- such as methods -- makes the overall flow easier to understand. Mnemonic names (describing what the method accomplishes) and programs that read like English descriptions of their behavior (through the use of well-chosen names) make your code more comprehensible to subsequent readers.

How do you know when to break code into pieces? If you discover that you have written a method that does not fit on a single page, you should write an outline for how the code works. Each of the major steps of this outline should be turned into a method. The original code should be rewritten in terms of these methods. The major steps should now be shorter methods. If these are still too long, repeat this process until each piece of code has a succinct description and occupies no more than two pages of code.

Note: Do not worry about inefficiency created by having too many small methods. First, intelligible code is so much easier to read and maintain, and code carefully optimized for efficiency so much more difficult to work with, that it rarely pays to do this sort of optimization until you are a skilled programmer. Further, a good compiler should be able to optimize. For example, if you make a method private or final, the compiler can in-line it.

The Repetition Rule of Thumb

Any time that the same code appears in two different places, you should consider capturing this common patterns of usage in a single method. When this happens, it is often because there is an idea expressed by this code. It is useful to give this idea a name, and to encapsulate or abstract it for reuse. Even if there are minor differences in the code as it appears, you may be able to abstract to a common method by supplying the distinguished information as arguments to the method. Each of the original pieces of code should be rewritten to use the common method.

Methods created by abstracting two or more pieces of code within the same class are often declared private. This is appropriate whenever the common behavior is local to the particular object and not something you want to make generally available. At other times, though, the common code is a useful and nameable function on its own. Though you may discover the commonality by replicating code, the existence of a separate method to replace this redundancy can be turned into an opportunity to export this functionality if it should make sense to do so.

Combining redundant code is also important in the case of constructors. Constructors can share code by having one invoke another -- using the special this() construct -- or by using a call to one or more (private) helper methods. A common programming mistake is to modify only one constructor when in
reality the same change must be made to every constructor. Having the bulk of the work of the constructor
done by a common method (or shared by using this()-constructors) eliminates this error.

Sharing redundant code shortens your program, making it easier to read, understand, modify, and maintain. It also helps to isolate a single point where each piece of behavior is performed. This single point can be understood, modified, and debugged once rather than each time it (redundantly) appears.

Example

In the example immediately below, we will modify code based on redundancy, i.e., the repetition rule of thumb. The result will also make our code more succinct and easier to read. The newly created method will be succinctly summarizable and a legitimately separable subtask.

Consider a bank account, which might have a method that allows the account's owner to obtain balance information:

```java
int getBalance( Signatory who ) throws InvalidAccessException
{
    if ( ! who == this.owner )
    {
        throw new InvalidAccessException( who, this )
    }
    // else
    return this.balance;
}
```

It might also have a withdraw method that allows the owner to remove amount from the account, returning that amount as cash:

```java
public Instrument withdraw( int amount, Signatory who ) throws InvalidAccessExcep
{
    if ( ! who == this.owner )
    {
        throw new InvalidAccessException( who, this )
    }
    // else
    this.balance = this.balance - amount;
    return new Cash( amount );
}
```

We could abstract the common pattern here, which is the verification of a signatory's right to access this account:

```java
private void verifyAccess( Signatory who ) throws InvalidAccessException
{
    if ( ! who == this.owner )
    {
        throw new InvalidAccessException( who, this )
    }
}
```

Now, we can rewrite getBalance and withdraw:
int getBalance( Signatory who ) throws InvalidAccessException
{
    this.verifyAccess( who );
    return this.balance;
}

public Instrument withdraw( int amount, Signatory who ) throws InvalidAccessException
{
    this.verifyAccess( who );
    this.balance = this.balance - amount;
    return new Cash( amount );
}

Much simpler, much more succinct, and in addition if we later need to modify the access verification routine, there is only a single place -- verifyAccess() -- where changes will need to be made.

Style Sidebar

Procedural Abstraction

- Use procedural abstraction when a method call would make your code (at least one of)
  - shorter, or
  - easier to understand.
- Your method should be concisely describable as "single function", though the function may itself have many pieces.
- Use parameters to account for variation from one invocation to the next.
- Return a value when the target of an assignment varies; leave the actual assignment out of the method body.
- Share code where possible. This is especially true among constructors, where one constructor can call another using this().
- Make internal helper procedures private. Make generally useful common functionality public (or protected).

Benefits of Abstraction

Abstracting procedures -- creating short, succinctly describable, non-redundant methods -- has many benefits. Even in the simple example of the preceding section, we can see many of these.

Procedural abstraction makes it easier to read your code, especially if methods have names corresponding to their succinct descriptions and the flow of code reads like the logic of the English description. Compare the before-and-after withdrawal methods of the bank account in the previous section.

Greater readability makes it easier to understand and figure out how to modify and maintain code. Separating functionality into bite-sized pieces also creates many opportunities to modify individual methods. Sharing these methods also centralizes the locations needing modification. For example, we
could add a digital signature check to the verification procedure of the bank account by modifying only 
verifyAccess, not the bodies of getBalance or withdraw.

In contrast, long methods with complicated logic can be particularly hard to modify, either because their 
interconnected logic can be so difficult to understand or because it can be hard to find the right place to 
make the change.

As the needs of your code change, you will also find it easier to rearrange and reconfigure what your code 
does if the logical pieces of the code are separated. For example, we might add a wireTransfer method to 
the bank account. In doing so, we can reuse the verifyAccess method.

Of course, smaller methods make for bite-sized debugging tasks. It is much easier to see how to debug 
access verification in the newer bank account than in the version where each account interaction has its 
own verification code and where verification is intimately intertwined with each transaction. And if we 
need to modify the verification procedure -- to give diagnostic information, to step through the method, or 
to fix it -- there is a central place to make these changes.

Procedural abstraction also makes it easier to change behavior by substituting a new version of a single 
method. If a method is not private, it can be overridden by a subclass, specializing or modifying the way in 
which it is carried out without changing its succinct specification. We could, for example, have a more 
secure kind of bank account using the digital signature verification method alluded to above.

Many of the advantages of procedural abstraction are also provided by good object design. A method 
signature is a reasonable abstraction of the behavior of an individual method. An interface plays a similar 
role for an entire object, packaging up (encapsulating) the behavioral contract of an object so that its 
particular implementation may vary. Interfaces also make it easier to see how a single abstraction can have 
many coexisting implementations.

Protecting Internal Structure

Procedural abstraction is an important way to separate use from implementation and a significant part of 
good program design. Procedural abstraction is not the only kind of abstraction that you need in a program, 
though. Often, other techniques are used, either alone or with procedural abstraction, to hide 
implementation details. For example, if you use procedural abstraction to create local helper methods, you 
generally will not want these helper methods to be available for other objects to use.

In this section, we will look at several ways to protect internal structure -- such as helper methods -- from 
use by others. These techniques protect implementation by making parts of the inner structure of an object 
inaccessible from outside that object or that group of interrelated objects. This packaging of internal 
structure is another kind of encapsulation. This section discusses some Java-specific ways to encapsulate 
functionality. Many programming languages offer similar mechanisms.

private

One of the most straightforward ways to protect internal structure -- such as fields or helper methods -- is 
to declare them private. We have seen in the section above how private methods can be used for procedural 
abstraction -- to break up a long procedure, to capture common patterns, etc. -- without exposing these
functions to other objects. A method (or other member) declared private can only be called from within the class.

**Beware:** This is not the same thing as saying that only an object can call its own private methods. An object can call the private methods of any other instance of the same class.

Private is extremely effective at protecting methods and other members from being used by other objects. However, a member declared private cannot be accessed from code within a subclass. This means that if you modify code in a subclass that relies on a private helper method in the superclass, you will have to recreate that private helper method.

**Packages**

An alternative to the absolute protection of private is the use of packages. A package is a collection of associated classes and interfaces. You can define your own packages. Libraries -- such as the Java source code or the cs101 distribution -- generally define packages of their own. The association among classes and interfaces in a package can be as loose or as tight as you wish to make it.

Sometimes the association among objects is merely by convenience: many kinds of objects deal with the same kind of thing. Most of the cs101 packages are of this sort. Often, it makes sense to define a set of interrelated classes and interfaces in a single package and to provide only a few entry points into the package, i.e., a few things that are usable from outside the package. These packages represent associations by shared interconnectedness. Most of the interlude code is of this sort. Java defines a large number of packages, some of each kind.

In the bank account, we might well choose to define the interface `Instrument` (representing cash and checks, among other things) and classes `BankAccount`, `CheckingAccount`, `Cash`, etc. in a single package, say `finance`.

Packages play two roles in Java. The first concerns names and nicknames. Packages determine the proper names of Java classes and interfaces. The second role of packages is as a visibility modifier somewhere between private and public.

**Packages and Names**

A class or interface is declared to be in particular package `packageName` if the first non-blank non-comment line in the file says

```
package packageName;
```

`packageName` may be any series of Java identifiers separated by periods, such as `java.awt.event` and `cs101.util`. By convention, package names are written entirely in lower case. A file that is not declared to be in a specific package is said to be in the default package, which has no name.

Every Java class or interface actually has a long name that includes its package name before its type name. So, for example, `String` is actually `java.lang.String`, because the first line of the file `String.java` says

```
package java.lang;
```
and Console is cs101.util.Console, because it is declared in a file that begins

    package cs101.util;

Any (visible) class or interface can always be accessed by prefacing its name by its package name, as in java.awt.Graphics or cs101.util.Console. If we declare the package finance as described above, the interface finance.Instrument would actually have a distinct name from the interface music.Instrument.

In some cases, you can also access the class more succinctly. If you include the statement

    import packageName.ClassName;

after the (optional) package statement in a file, you may refer to ClassName using just that name, not the long (package-prefaced) name. So, for example, after

    import cs101.util.Console;

the shorter name Console may be used to refer to the cs101.util.Console class. Similarly,

    import packageName.*;

means that any class or interface name in packageName may be referred to using only its short name, unprefaced by packageName.

Note, however, that this naming role for packages is only one of convenience and does not provide any sort of actual encapsulation. The use of a shorter name does not give you access to anything additional. In particular, it does not change the visibility of anything. Anything that can be referred to using a short name after an import statement could have been referred to using the longer version of its name in the absence of an import statement.

There are three exceptions to the need to use an import statement, i.e., three cases in which the shorter name is acceptable even without an explicit import.

1. Names in the default package can always be referred to using their short names.
2. Names in the current package (i.e., the package of which the file is a part) can always be referred to using their short names.
3. Names in the special package java.lang can always be referred to using their short names.

You are not allowed to have an import statement that would allow conflicts. So, for example, you could not have both statements

    import finance.*;
    import music.*;

if both packages contain a type named Instrument. You could, however,
import finance.BankAccount;
import music.*;

since the first of these import statements doesn't shorten the name of the interface

Package Naming Summary

A class or interface with name TypeName that is declared in package packageName may always be accessed using the name packageName.TypeName, provided that it is visible. (See the visibility summary sidebar.)

The class or interface may be also accessed by its abbreviated name, TypeName, without the package name, if one of the following holds:

- The class or interface is declared in the default (unnamed) package.
- The class or interface is declared in the current package, i.e., packageName is also the package where the accessing code appears.
- The class or interface is declared in the special package java.lang, i.e., packageName is java.lang.
- The file containing the accessing code also contains one of the following import statements:
  - import packageName.TypeName;
  - import packageName.*;

Packages and Visibility

The second use of packages is for visibility and protection. This use does accomplish a certain kind of encapsulation. We have already seen private and public, visibility modifiers that prevent the marked member from being seen or used or make it accessible everywhere. These two modifiers are absolute. Packages allow intermediate levels of visibility.

Between private and public are two other visibility levels. One uses the keyword package. The other is the level of visibility that happens if you do not specify any of the other visibility levels. This is sometimes called "package" visibility, although it differs from friendly visibility in other languages and, additionally, there is no corresponding keyword for it.

A member marked protected visible may be used by any class in the same package. In addition, it may be referenced by any subclass. It is illegal -- and causes a compiler error -- if something outside the package, not a subclass, tries to reference a member marked protected visible.

A member, class, or interface not marked with a visibility modifier is visible only within the package. It may not be accessed even by code within subclasses of the defining class or interface, unless they are within the package.
This means that classes and interfaces may be declared without the modifier `public`, in which case they can only be used as types within the package. Members may be declared without a modifier, in which case they can be used only within the package, or they may be declared `protected`, in which case they can be used only within the package or within a subclass. A non-public class or interface need not be declared in its own separate Java file.

Note, however that although a subclass may increase the visibility of a member, it may not further restrict visibility. So a subclass overriding a `protected` method may declare that method `public`, but not unmodified (package) or `private`.

There is no hierarchy in package names. This means that the package `java.awt.event` is completely unrelated to the package `java.awt`; their names just look similar.

**Visibility Summary**

A member, class, or interface marked `public` may be accessed anywhere.

A member marked `protected` may be accessed anywhere within the containing package or anywhere within a subclass (or implementing class).

A member, class, or interface not marked has "package" visibility and may be accessed anywhere and only within the containing package.

A member marked `private` may only be accessed within the containing class or interface.

We can use this approach to encapsulate certain aspects of our `BankAccount` example without making all of the relevant members private. After all, we want to protect these members from misuse by things outside the financial system (and therefore presumably outside the package `finance`), not from legitimate use by other things within the banking system.

So we might declare:

```java
public class BankAccount
{
    ...
}
```

and

```java
public interface Instrument
{
    public abstract int getAmount();
    public abstract void nullify();
}
```

but

```java
class Cash implements Instrument
{
```
private int amount;
private boolean valid;

protected Cash( int amount )
{
    this.amount = amount;
    this.valid = true;
}

public int getAmount()
{
    return this.amount;
}

protected void nullify()
{
    this.valid = false;
}

This absence of the keyword public on the class definition means that the class Cash is accessible only to things inside the finance package. The Cash constructor is declared protected, so Cash may be created only from within this package. But the two methods that Cash implements for its interface, Instrument, must be public because you cannot reduce the visibility level declared for a method and the interface's methods are declared public.[Footnote: The methods of a public interface must be public, but an interface not declared public may have methods without a visibility modifier.]

Unfortunately, the guarantees of packaging are not absolute. There is nothing to prevent someone else from defining a class to reside in an arbitrary package. For example, I could declare a class Thief in package financial, allowing Thief instances full access to the Cash constructor.

Inheritance

Inheritance can be used as a way of hiding behavior. Specifically, you can create hidden behavior by extending a class and implementing the additional behavior in the subclass. Conversely, labelling an object with a name of a superclass type has the property that it makes certain members of that object invisible.

You cannot invoke a subclass method on an object labelled with a superclass type that does not define that method, even though the object manifestly has the method. You can take advantage of this in combination with the visibility modifiers, for example creating a package-only subclass of a public class. Outside the package, instances of this subclass will be regarded as instances of the superclass, but because the subclass type is not available (since it is not visible outside the package), its additional features cannot be used.

For example, a specialized package-internal type of BankAccount might allow checks to be written:

    class CheckingAccount extends BankAccount
    {
      ...

    protected Instrument writeCheck( String payee,
              int amount,
              Signatory who );
Now, if I have a CheckingAccount but choose to label it with a name of type BankAccount, I cannot write a check from that account:

```java
BankAccount rainyDayFund = new CheckingAccount(...);
rainyDayFund.withdraw( 10000 );
```
works fine, but not

```java
rainyDayFund.writeCheck( "Tiffany's", 10000, diamondJim );
```

That is, the only methods available on an expression whose type is BankAccount are the BankAccount methods. The fact that this is really a CheckingAccount is not relevant.

The idea of using superclass types as ways of abstracting the distinctions between a CheckingAccount and a MoneyMarketFund is an important one. Sometimes subclasses provide extra (or different versions of) functionality. These distinctions are not necessarily relevant to the user of the class, who should be able to treat all BankAccounts uniformly.

Note, however, that the true type of an object is evident at the time of its construction; it must be constructed using the class name in a new expression. Also, if the type is visible, an explicit cast expression can be used to access subclass properties.[Footnote: For example, (CheckingAccount) rainyDayFund;]

Finally, recall the discussion in chapter 10 on the inappropriateness of inheritance unless you are legitimately extending behavior. Inheritance should not be used, for example, when you need to "cancel" superclass properties.

**Clever Use of Interfaces**

The discussion above of inheritance and encapsulation applies doubly for interfaces. Interfaces are a good way of achieving the subtype properties of inheritance without the requirements of strict extension. Further, an interface type cannot contain implementation, only static final fields and non-static method signatures. This means that an interface cannot divulge any properties of the implementation that might vary from one class to another or that a subclass might override. If it's in the interface, it's in every instance of every class that implements that interface.
The example in the preceding section of a `CheckingAccount` protected by subclassing are even cleaner in the case of the `Cash` and `Check` classes, which are package-local but implement the public interface `Instrument`. This means that things outside the package may hold `Cash` or `Check` objects, but will not know any more than that they hold an `Instrument`. Any methods defined by `Cash` or `Check` but not by `Instrument` are inaccessible except inside the package `finance`.

Like a superclass, the protections of an interface can be circumvented if the implementing class type is visible to the invoking code. And, as always, the true type of an object is known when you invoke its constructor.

These issues are covered further in chapters 4 and 8, on Interfaces and Designing With Objects.

**Inner Classes**

The final topic in this chapter is inner classes. Inner classes allow a variety of different kinds of encapsulation. At base, an inner class is a remarkably simple idea: An inner class is a class defined inside another. There are several varieties of inner classes, and some of their behavior may seem odd.

Because an inner class is defined inside another class, it may be protected by making it invisible from the outside, for example by making it private. This makes inner classes particularly good places to hide implementation. The actual types of private inner classes are invisible outside of their containing objects, making the inheritance and interface tricks of the previous section more powerful.

Conversely, inner classes can also be used to protect their containing objects. An inner class lives inside another object and has privileged access to the state of this "outer" object. For this reason, inner classes can be used to provide access to their containing objects without revealing these outer objects in their entirety. That is, an inner class's instance(s) can (perversely) be used to limit access to its containing class.

**Beware:** Although an inner class is defined inside the text of another class, there is no particular subtype relationship established between the inner and outer classes. For example, an inner class normally does not extend its containing (outer) class.

**Static Classes**

A static inner class is declared at top level inside another class. It is also declared with the keyword `static`. Static inner classes are largely a convenience for defining multiple classes in one place. A static class declaration is a static member of the class in which it is declared, i.e., it is similar to a static field or static method declaration.

Understanding static inner classes is quite straightforward. There are only a few real differences between a static inner class and a regular class. First, the static inner class does not need to be declared in its own text file, even if it is public. In contrast, an ordinary public class must be declared in a file whose name matches the name of the class. Second, the static inner class has access to the static members of its containing class. This includes any private static methods or private static fields that the class may have.

The proper name of a static inner class is `OuterClassName.InnerClassName`. 
Beware: This naming convention looks like package syntax (or field access syntax), but it is not.

The constructor for a static class is accessed using the class name, i.e.,

```java
new OuterClassName.InnerClassName()
```

perhaps with arguments as with any constructor.

**Member Classes**

A member class is defined at top level inside another class, but without the keyword static. A member class declaration is a non-static member of the class in which it is declared, i.e., it is similar to a non-static field or method declaration. This means that there is exactly one inner class (type) corresponding to each instance of the outer class. If there are no instances of the outer class, there are in effect no inner class types. When an outer instance is created, a corresponding inner class (i.e., factory) is created and may be instantiated. Note that this does not necessarily make any inner class instances; it just creates the factory object. The inner class and all of its instances have privileged access to the state of the corresponding outer class instance. That is, they can access members, including private members.

An example may make this clearer. Suppose that we want to have a Check class corresponding to each CheckingAccount. The Check class that corresponds to my CheckingAccount is similar to the Check class that corresponds to your CheckingAccount, but with a few differences. Specifically, my Check class (and any Check instances I create) should have privileged access to my CheckingAccount, while your Check class should have privileged access to your CheckingAccount. So, in effect, the Check class corresponding to my CheckingAccount is different from the Check class corresponding to your CheckingAccount. It differs precisely in the details of the particular CheckingAccount to which it has privileged access. Creating a third CheckingAccount -- say, Bill Gates's CheckingAccount -- should cause a new kind of Check, Bill Gates's Checks, to come into existence. These Checks differ from yours and mine. Note that creating Bill Gates's CheckingAccount also creates Bill Gates's Check type, but doesn't necessarily create any of Bill Gates's Check instances. Bill still has to write those....

```java
class CheckingAccount extends BankAccount {
    ...

    protected class Check implements Instrument {
        private BankAccount originator = CheckingAccount.this;

        private String payee;
        private int amount;

        private boolean valid;

        ....

        protected Check( String payee, int amount, Signatory who ) {
            ...
            if ( ! who.equals( CheckingAccount.this.owner ) ) {
            ....
```
throw new BadCheckException( this );
}
this.validate( Signatory );

this.payee = payee;
this.amount = amount;
this.valid = true;
}

Instrument cash() throws BadCheckException
{
    if ( ! this.valid )
    {
        throw new BadCheckException( this );
    }
    Instrument out = this.originator.withdraw( this.amount );
    this.nullify();
    return out;
}

In this case, there is in effect one Check class for each CheckingAccount. This is precisely what you'd want: each CheckingAccount has a slightly different kind of Check, varying by who is allowed to sign it, etc.

The proper name of a member class is instanceName.InnerClassName, where instanceName is any expression referring to the containing instance. So a way to name Bill's check type is gatesAccount.Check (assuming gatesAccount is Bill's CheckingAccount), and he can write a new Check using

    new gatesAccount.Check( worthyCharity, 1000000, billSignature )

Note that he can't just say new Check(...), because that leaves ambiguous whether he's writing a check from his account or from mine.

There is a special syntax that may be used inside the inner class to refer to the containing (outer class) instance: OuterClassName.this. For example, in the Check constructor code above, a particular Check's Signatory is compared against the owner of the containing CheckingAccount by comparing it with the owner of the containing CheckingAccount instance. This ensures that I can't sign a Bill Gates Check, nor he one of mine. It is accomplished by looking at CheckingAccount.this's owner field. Note the use of the CheckingAccount.this syntax to get at the particular CheckingAccount whose Check class is being defined.

The Check serves as a safely limited access point into the CheckingAccount. For example, each Check knows its CheckingAccount's owner. When a new Check is being created, the Check's Signatory is compared against the account owner (CheckingAccount.this.owner, a field access expression) to make sure that this person is an authorized signer. The identity of the allowable Signatory of the check is hidden, but it is fully encapsulated inside the Check itself. Anyone can get hold of the Check without being able to get hold of the Signatory (or BankAccount balance) inside.
Local Classes and Anonymous Classes

There are two additional kinds of inner classes, local classes and anonymous classes. They are briefly explained here but their intricacies are beyond the scope of this chapter.

A local class declaration is a statement, not a member. A local class may be defined inside any block, e.g., in a method or constructor. There is in effect exactly one local class for each execution of the block. For example, if a local class is defined at the beginning of a method body, there is one local class type corresponding to each invocation of the method, i.e., the class depends on the invocation state of the method itself.

The syntax of a local class method is much like member class declaration, but the name of a local class may only be used within its containing block. A local class's name has the same visibility rules as any local name, i.e., its scope persists from its declaration until the end of the enclosing block. You may only invoke a local class's constructor with a `new` expression within this scope. You may return these instances from the method or otherwise use these instances elsewhere, but their correct type will not be visible elsewhere. Instead, you must refer to them using a superclass or interface type.

A local class has privileged access to the state of its containing block as well as to the state of its containing object (class or instance). The local class may access the parameters of its containing method, as well as any local variables in whose scope it appears, provided that they are declared final. If a local class is defined in a nonstatic member (method or constructor), the local class's code may access its containing instance using the `OuterClassName.this` syntax. If a local class is defined in a static member (e.g., in a static method), the local class has only a containing class, not a containing instance.

An anonymous class declaration is always a part of an anonymous class instantiation expression. Anonymous classes may be defined and instantiated anywhere where an instantiation expression might occur. They have a special, very strange syntax. An anonymous class is only good for making a single instance as an anonymous class declaration cannot be separated from its instantiation. Anonymous classes are a nice match for the event handling approaches of the Event Delegation chapter.

The syntax for an anonymous class declaration-and-instantiation expression is

```
new TypeName () { memberDeclarations }
```

where `TypeName` is any visible class or interface name and `memberDeclarations` are non-static field and method declarations (but not constructors).[Footnote: If there is necessary instance-specific initialization of an anonymous class, this may be accomplished with an instance initializer expression. Such an expression is a block that appears at top level within the class and is executed at instance construction time.] If `TypeName` is a class, the anonymous class extends it; if `TypeName` is an interface, the anonymous class implements it. In either case, `memberDeclarations` must include any method declarations required to make an instantiable (sub-)class. The evaluation rules for this expression create a single instance of this new -- and strictly nameless -- class type. Like a local class, the anonymous class's code may access any final parameters or local variables within whose scope it appears, and may use `OuterClassName.this` to refer to its containing instance if its declaration/construction expression appears within a non-static member.

Inner Classes
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Accessibility</th>
<th>Class is contained within</th>
<th>Access to static members of containing class?</th>
<th>Access to containing instance (including its fields and methods)?</th>
<th>Access to parameters and local variables of containing block?</th>
</tr>
</thead>
<tbody>
<tr>
<td>like static member (public, protected, private, etc.)</td>
<td>like member (public, protected, private, etc.)</td>
<td>like local variable name, i.e., only within block</td>
<td>invisible</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declaration syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility static class ClassName {</td>
</tr>
<tr>
<td>members }</td>
</tr>
<tr>
<td>as statement in anonymous class instantiation expression</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Where declared?</th>
<th>at top level in OuterClass</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Instantiation syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>new OuterClass . InnerClass ( ... )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access to static members of containing class?</th>
<th>yes</th>
<th>yes</th>
<th>yes</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to containing instance (including its fields and methods)?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes, using OuterClass . this</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes, using OuterClass . this</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access to parameters and local variables of containing block?</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declaration syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility class ClassName {</td>
</tr>
<tr>
<td>members }</td>
</tr>
<tr>
<td>only possible in instantiation (see below).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Where declared?</th>
<th>at top level in OuterClass</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Instantiation syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>new SuperTypeName ()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>members</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Chapter Summary

- An abstraction relies only on general properties, leaving implementation details to vary.
- Encapsulation packages up and hides those details.
- Procedural abstraction uses methods to accomplish abstraction and encapsulation.
- A method should be short, have a succinctly summarizable function, and not contain code that is redundant with other methods.
- Abstraction and encapsulation enhance the readability, comprehensibility, modifiability, and maintainability of code.
- Packages provide grouping among interrelated classes.
- The full name of a class or interface is prefaced by its package name.
  - Import statements allow you to circumvent this longer name.
  - Some other short names are automatically available, even without an import statement.
- Visibility modifiers limit access to class members, including inner classes. Together with the use of superclass or interface type names, they provide a way to limit access to an object.
- Inner classes are a mechanism for defining one class inside another.
  - This can be used to hide the inner class.
  - This can also be used to limit access to the outer class by distributing the inner class instead.

Exercises

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Intelligent Objects and Implicit Dispatch

Chapter Overview

- How can I exploit method "ownership" to make objects do what I want?
- How do I pass behavior around?
- How do I know which method will be invoked?

Methods belong to objects. In some cases, as when getter and setter methods allows access to an object's internal state, the reason for housing methods in objects is clear. But in many cases, it may be less obvious why a method ought to be affiliated with a particular object. In this chapter, we look at several cases in which methods are used in concert with their owning objects to accomplish tasks that might not be obvious.

Methods can be used as a way to create implicit dispatch. Many objects, belonging to many different classes, can each be given a method of the same name (and footprint). In this case, dispatching to the correct code is as simple as asking the object to perform this method for you.

Fixing the name of the method but leaving the owning object to vary allows you to do a wide range of things. You can, in effect, pass a method as an argument (by passing its containing object), return a method from a procedure (by returning its containing object), or store it in a name or other structure (by storing its containing object). You can remember who called you and arrange to call that object back; you can build complex homogeneous structures by exploiting the fact that one object is associated with other, equally intelligent, objects that can cooperatively solve problems that none could solve individually.

Each of these mechanisms works because every method is associated with an object. If the method name is fixed at the time that the program is written, its target object can be allowed to vary, allowing a runtime decision as to which piece of code -- which instructions, which method body -- should actually be executed.

Objectives of this Chapter

1. To understand Java's method dispatch mechanism.
2. To be able to use the same-named method in different classes of objects to create an implicit dispatch.
3. To appreciate how Runnables can be used to encapsulate procedures.
4. To learn how to set up and use callbacks so that a method can convey information to its calling object without returning.
5. To recognize various forms of recursion and to be able to use structural recursion as a problem-solving technique.
6. To understand, recognize, learn how, increase familiarity, master details, appreciate, discover, be able to ....

**Procedural Encapsulation and Object Encapsulation**

In the previous chapters, we saw how a central control loop can be used as a dispatcher, invoking different methods at different times depending on circumstances such as the value of a particular piece of state. We also saw how the different responses can be packaged up inside methods and how these methods in turn can be encapsulated inside objects. In this chapter, we will take these ideas one step further and use Java's method dispatch mechanism (plus some clever design) to determine what response is appropriate under various circumstances.

Before we turn to the use of objects as a dispatch mechanism, let's briefly review some of the properties of methods and of objects.

A method is a set of instructions to be followed. The method instructions are executed when an instruction-follower evaluates a corresponding method invocation expression, i.e., a call to the method. The method instructions may require some information to be able to execute; these are the arguments to the method. The method instructions may also produce some information; this is the method's return value.

Every method belongs to a particular object; there are no methods "just floating around" in Java. Each method body is textually contained in a class definition. Regular methods belong to individual instances of the class in which they are textually contained. (Static methods belong to the class object itself.)

For example, if `yesBox` is a Checkbox and you want to find out whether `yesBox` is currently selected, you can ask `yesBox` to supply you with that information by using the method invocation expression `yesBox.isSelected()`. There's no way to just ask `isSelected()`, though: you have to know whose `isSelected()` method it is.

Methods encapsulate behavior, but they do not by themselves encapsulate state. This is the role of objects. An object typically contains both methods -- sets of instructions -- and persistent information. For example, the Checkbox named by `yesBox` has a method called `isSelected()`, which provides instructions for how to determine whether `yesBox` is currently checked. When the expression `yesBox.isSelected()` is evaluated, those instructions are executed and the desired information is produced. But when the method is not being invoked, the method itself doesn't have any information or action. In contrast, even in the absence of any method invocation, the Checkbox `yesBox` contains state indicating whether it is currently selected, perhaps in the form of a private boolean field.

Objects, then, package both behavior (in the form of methods that can be invoked) and persistent state that provides a background context for that behavior. (Presumably `yesBox.isSelected()` behaves differently depending on whether the hypothetical private boolean field is true.) An object exists even when none of its methods is being invoked, and its fields persist between method invocations. An object is thus a powerful mechanism for modeling parts of the world. By making that state internal to the object, hiding it from external access, and providing a set of methods that give selective access to that state, objects can be used to encapsulate the coherent behavioral aspects of real-world things. The method `isSelected()` by itself would have little meaning. The object `yesBox` provides a context for the `isSelected()` method, so that it legitimately models coherent persistent behavior.
The name of a method to be invoked must be chosen at the time that you are writing your program. In contrast, the particular object whose method will be invoked need not be known until the time that you actually run the program. For example, when the expression `yesBox.isSelected()` is written, the method name -- `isSelected` -- and even its footprint -- no arguments -- is already known. No other method can be invoked with this expression. But at the time that the expression is written, it may not be possible to tell to which object the name `yesBox` will refer. It may, in fact, not even be possible to tell the exact type of the object to which `yesBox` refers, although we know that it will be some type of Checkbox. (It could be any subtype of Checkbox.)

In the remainder of this chapter, we will see how fixing the method name and allowing its target object to vary gives the programmer a great deal of additional power. In the first -- central -- example of this technique, we will shift specialized behavior from their previous location in the handler methods within a single object to a new role within separate objects, objects encapsulating both those handler methods and associated state. We will see how this migration of behavior from procedural encapsulation to object encapsulation provides a different model for dispatch, and how it can be used to make object-oriented programming a remarkably powerful technique.

**From Dispatch to Objects**

Consider the following problem: You are writing code that will retrieve objects, one at a time, and print them out to the user. Some of these objects will be Strings. Some of the objects will be Points, items representing two-dimensional coordinates. A Point object has methods to retrieve its individuate coordinates, called `getX()` and `getY()`, each returning an int. And some of the objects will be Dimensions, items representing two-dimensional extents, with int-returning `getWidth()` and `getHeight()` methods. Your job is to write the `printObject` method.

**A Straightforward Dispatch**

You might implement this by using a simple dispatch mechanism. Since this dispatch is done on the basis of the object's class, you cannot use a switch statement. So we'll try an if:

```java
public void printObject( Object o )
{
    if ( o instanceof String )
    {
        Console.println( o );
    }
    else if ( o instanceof Point )
    {
        Point p = (Point) o;
        Console.println( "Point: (" + p.getX() + "," + p.getY() + ")" );
    }
    else if ( o instanceof Dimension )
    {
        Dimension d = (Dimension) o;
        Console.println( "Dimension: (" + d.getWidth() + "," + d.getHeight() + ")" );
    }
}
```
[Footnote: This code suffers from a few problems, not the least of which is that it doesn't do anything about the possibility that o is none of the above. While we'd never write such code in a real application, we'll skip the else error condition clause here for pedagogic succinctness.]

**Procedural Encapsulation**

Of course, knowing what we do about procedural encapsulation, this looks like a superb opportunity to break out the concisely describable code. There are two relatively obvious routines lurking here:

```java
public String pointToString( Point p )
{
    return "Point: (" + p.getX() + "," + p.getY() + ")";
}

and

public String dimensionToString( Dimension d )
{
    return "Dimension: (" + d.getWidth() + "," + d.getHeight() + ")";
}
```

We might also, for symmetry, add

```java
public String stringToString( String s )
{
    return s;
}
```

although it doesn't seem particularly well-motivated at the moment.

Given these routines, we might rewrite printObject as

```java
public void printObject( Object o )
{
    if ( o instanceof String )
    {
        Console.println( this.stringToString( (String) o ) );
    }
    else if ( o instanceof Point )
    {
        Console.println( this.pointToString( (Point) o ) );
    }
    else if ( o instanceof Dimension )
    {
        Console.println( this.dimensionToString( (Dimension) o ) );
    }
}
```

**Variations**
The new printObject still has a certain amount of redundant code. We can pushing the Console.println out of the individual ifs, but then we'll need to remember the String returned by each toString method. We could write

```java
public void printObject( Object o )
{
    String s = "";
    if ( o instanceof String )
    {
        s = this.stringToString( (String) o );
    }
    else if ( o instanceof Point )
    {
        s = this.pointToString( (Point) o );
    }
    else if ( o instanceof Dimension )
    {
        s = this.dimensionToString( (Dimension) o );
    }
    Console.println( s );
}
```

In yet another optimization, we could actually transfer the coercion into the individual toString methods, calling them on Objects rather than on specialized types. This makes the methods somewhat less general -- what if they're called on the wrong type of objects? -- but if we can be sure that they'll always be called appropriately, it cleans up our dispatch code further.

```java
public String pointToString( Object o )
{
    Point p = (Point) o;
    return "Point: (" + p.getX() + "," + p.getY() + ")";
}

public String dimensionToString( Object o )
{
    Dimension d = (Dimension) o
    return "Dimension: (" + d.getWidth() + "," + d.getHeight() + ")";
}

public String stringToString( Object o )
{
    return (String) s;
}
```

Now the dispatch routine reads

```java
public void printObject( Object o )
{
    String s = "";
    if ( o instanceof String )
    {
        s = this.stringToString( o );
    }
    else if ( o instanceof Point )
```
{  
s = this.pointToString( o );
}
else if ( o instanceof Dimension )
{
  s = this.dimensionToString( o );
}  
Console.println( s );
}

Pushing Methods Into Objects

We can take this whole approach one step further, and in doing so dramatically simplify our dispatcher code. Instead of trying to give this dispatcher object a toString method for each individual type that it might need to know about, we can put the toString methods into the individual types directly. For example, Point might have a method that says:

```
public class Point
{

    //...

    public String toString()
    {
        return "Point: (" + this.getX() + "," + this.getY() + ")";
    }
}
```

This is just the old pointToString, with a few modifications. First, note that we've eliminated the argument that pointToString needed. This is because the Point we're converting is this, i.e. the particular object whose toString() method is being executed. Second, we don't need a coercion. That's because if this set of instructions is being executed, it is because this (Point) object's toString() method has been called, i.e., we must be dealing with a Point. You simply can't call Point's toString() method on a Dimension (or a String).

A similar modification gives us Dimension's toString() method:

```
public class Dimension
{

    //...

    public String toString()
    {
        return "Dimension: (" + this.getWidth()
                + "," + this.getHeight() + ")";
    }
}
```

And finally String's toString method is quite simple:
public class String {
    //...
    public String toString() {
        return this;
    }
}

Now, if `origin` names the Point with coordinates (0,0) and `square` names the Dimension with height 25 and width 25, `origin.toString()` returns the String "Point: (0,0)", while `extentless.toString()` returns the String "Dimension: (25,25)". Each object knows how to turn itself into a String using the `toString()` method provided by its class.

In point of fact, the Java class `java.lang.Object` has a `toString()` method, and so any Java object necessarily has a `toString()` method. In many cases, the `toString()` method is inherited from `Object` and so prints a rather ugly representation of the object. You may wish to override the `toString()` method of any class you expect to be printing out a lot. For example, there is a real class called `java.awt.Point`, but its `toString()` method isn't quite as succinct as the one we've given here.

What Happens to the Central Loop?

We have seen that writing the methods inside their respective classes makes them considerably more succinct. After all, the `toString()` method of Point just has to give instructions for how to print `this`, i.e., the particular Point whose `toString()` method is being invoked. At the time that the method is invoked, all of the relevant information is present in the `target` -- the object whose method is invoked, i.e., `this`. But we haven't come to the best part yet.

Suppose that our types each implement their own `toString()` method. What, then, does the dispatcher look like?

The new dispatch code is

```java
    public void printObject( Object o )
    {
        Console.println( o.toString() );
    }
```

Where did the conditional go? The answer is that it is hidden inside Java's method dispatch mechanism. Java decides which `toString()` method to invoke by looking at the target's type.

Whenever an instruction-follower evaluates a method invocation expression, Java does a quick calculation to determine what kind of object the target -- the method's owner object -- is. Depending on the class of that object, Java looks up the appropriate method to invoke. (The argument types also play a role in selecting the method invoked, specifically by selecting a method whose footprint is appropriate.)
dispatch based upon the type of the target object is a simple form of **polymorphism**. In general, polymorphism means doing different things with different types of objects.

If we move the dispatchee methods out to their respective classes, we give each kind of object its own type-specific way to respond to the request. Here, a particular -- known, fixed -- method name and footprint is polymorphic with respect to the target object to which it belongs. (Instances of many classes support the same method footprint. Each class provides a different implementation.) By allowing the target object to vary, we cause the same expression to invoke different pieces of code.

This approach has several benefits. First, the dispatcher becomes significantly more succinct. Second, the code that actually does the work is associated with a specific type, meaning that it doesn't have to worry about verifying type or coercion. Java does both dispatch and coercion automatically. The method is necessarily invoked on a target of the appropriate type, because the target helps to determine which method is invoked. Finally, if a new object type is to be added (e.g., to the printObject method), the particular instructions for converting it to a String can be added in the definition of the object's class; printObject no longer needs to worry about which types it is suited to handle. In fact, since toString is a method defined in the class java.lang.Object, printObject can handle any kind of Object at all.

**The Use of Interfaces**

In the example above, we gained great power from pushing the conversion to a String into each specific object type. Of course, any object type not supplied with its own toString() method simply inherits one from its superclass. Since java.lang.Object is the root of the class inheritance hierarchy, each class is guaranteed to have a toString() method, if only the one defined for Object. But sometimes you will want to use polymorphism to dispatch to a method that isn't defined on java.lang.Object. What do you do then?

Consider the Calculator buttons of an earlier chapter. In that example, number buttons are supposed to display themselves on the Calculator screen, while arithmetic operator buttons are supposed to perform calculations and the clear button is supposed to erase whatever happens to be displayed. The central dispatcher of that program checked which button had been pressed and called the appropriate helper method, contained within the dispatcher object.

Precisely the same sort of logic that we applied to the object printer would work here. First, we need to define a series of object types. For example, we might have a NumberButton class whose ten instances represent the number keys, from 0 to 9. We might have an OperatorButton class, one of whose instances would represent the addition function of the calculator. And we might have a ClearButton class with a single instance corresponding to the calculator's clear key.

Each of these classes might be endowed with a buttonPressed method, to be invoked by the dispatcher when the corresponding calculator button is pressed. For example, ClearButton's buttonPressed method might say resetCalculator, while a NumberButton's buttonPressed method would invoke displayDigit. Whose resetCalculator and displayDigit methods are these? They belong to the calculator. In order to do its job, the buttonPressed method will need to be given access to the CalculatorState -- an object representing what's going on inside the Calculator -- as an argument.

```java
public class ClearButton
{

```
public void buttonPressed( CalculatorState calc )
{
    calc.resetCalculator();
}

When the individual clear button's buttonPressed method is invoked, it will in turn ask the calculator to reset itself.

public class NumberButton
{
    private final int whichDigit;

    public NumberButton( int which )
    {
        this.whichDigit = which;
    }

    public void buttonPressed( CalculatorState calc )
    {
        calc.displayDigit( this.whichDigit );
    }
}

Note that there are ten different NumberButton instances, and each instance will need to remember which digit it represents.[Footnote: Once assigned, this digit doesn't change; hence, the field is declared final.] When, for example, the 0 button's buttonPressed method is invoked, it asks its calculator to display its digit, i.e., 0. The code for other button types is similar.

When we are done writing these button types, we will need to add code to the calculator dispatcher (or to some other part of the system) that creates all of the necessary instances of these classes. We might, for example, stick these instances into an array indexed by the buttonID ints described in chapter 12. This would be a field of our animate calculator object:

    private Object[] buttonObjects = new Object[ Calculator.LAST_BUTTON_ID ];

And then, inside the constructor for that object, we need initialization code:

    for ( int buttonID = 0; buttonID < 10; buttonID = buttonID + 1 )
    {
        this.buttonObjects[ buttonID ] = new NumberButton( buttonID );
    }

    // and so on for operators, clear....

Once we have instantiated these button types, what does the dispatcher look like? Its job will simply be to invoke the appropriate button object's buttonPressed method.
public void act()
{
    int buttonID = this.gui.getButton();
    this.buttonObjects[ buttonID ].buttonPressed( this.calcState );
}

There is just one problem: this code won't compile. The array buttonObjects is an array of Objects. But most Objects don't have a buttonPressed( CalculatorState ) method.

Why wasn't this a problem for the toString method of the object printer? Because each Object has a toString() method, we didn't have to do anything special to make the corresponding line of code -- the invocation of the object's toString() method -- work. However, if we try this trick with a method that isn't possessed by every object, we will find that our code won't compile. We can resolve this by using an interface that specifies this contract.

    public interface CalculatorButton
    {
        public void buttonPressed( CalculatorState calc );
    }

This interface gives just the information we need -- the presence of a buttonPressed method that requires a CalculatorState -- without saying anything about how a particular CalculatorState should respond to a button's being pressed. It leaves those aspects of the method to each class that provides an implementation for CalculatorButton's buttonPressed method.

We will also need to go back and add this interface to each of the individual calculator button classes. For example:

    public class ClearButton implements CalculatorButton
    {
        public void buttonPressed( CalculatorState calc )
        {
            calc.resetCalculator();
        }
    }

Now, we can rewrite our declaration of the buttonObjects array.

    private CalculatorButton[] buttonObjects
        = new CalculatorButton[ Calculator.LAST_BUTTON_ID ];

Finally, our code will compile!

The calculator button is a more general example than the object printer, but both illustrate the same set of ideas. By pushing methods out of the central dispatcher object and into the classes representing distinct types of objects, we can package up the methods with the information that they need to do their jobs. We can also largely eliminate the explicit dispatcher of the chapter 12, using Java's method dispatch mechanism in its place. This approach is very much in keeping with the philosophy of object-oriented design: keep behavior together with state encapsulated in objects.
Runnables as First Class Procedures

We have actually seen a special case of this kind of target-polymorphism-as-dispatch in our use of Animates as the instructions for AnimatorThreads. In that case, an AnimatorThread does very different things depending on the class of the particular object whose act() method it executes. In other words, AnimatorThread uses its constructor argument -- the object whose act() method it is supposed to execute -- to determine what it is supposed to do. The method footprint -- act() -- is fixed by the Animate contract. Naming this method there allows the programmer to write it explicitly into code. Remember, method names cannot be deduced and runtime, though their target objects can.

There is a similar situation in Java involving the interface Runnable (with a single method, run() ) and the class Thread. A Thread is started on a particular object, and the Thread follows the instructions supplied by that object's run() method. By starting them on instances of different classes of Runnable objects, Threads can be induced to behave in very different ways. Like act(), run() exploits Java's target-based dispatch mechanism to create different kinds of behavior.

But Runnables and run() can be used even without starting a new Thread, simply because they are fixed names for executable behavior that takes no arguments.[Footnote: Everything said here for run() could be done with another method with a different name, but that name, too, would have to be fixed when the program is written. For no-arguments executable code, run() and Runnable make a convenient convention. If you wish to pass arguments to this procedure, you will need to define your own interface and your own method signature, as Java offers no standard conventions.] Suppose that you want to pass a procedure around from one object to another. For example, suppose that you want to create a secret message and later, you will give that message to a decoder that will print out your secret message. One way to do this is to make the secret message a Runnable object and to use the secret message's run() method as a way for the decoder to get the message out.

```java
public class SecretMessage implements Runnable {

    private String message;

    public SecretMessage( String message )
    {
        this.message = message;
    }

    public void run()
    {
        Console.println( this.message );
    }

}

public class SecretDecoder
{

    public void decode( Runnable secret )
    {
        secret.run();
    }

}
```

[Footnote: Everything said here for run() could be done with another method with a different name, but that name, too, would have to be fixed when the program is written. For no-arguments executable code, run() and Runnable make a convenient convention. If you wish to pass arguments to this procedure, you will need to define your own interface and your own method signature, as Java offers no standard conventions.]
Now, if we have

```java
SecretMessage message = new SecretMessage( "Meet me at midnight." );
```

and

```java
SecretDecoder decoder = new SecretDecoder();
```

then we can try

```java
decoder.decode( message );
```

which will print

```
Meet me at Midnight.
```

to the Java console. The message stays safe inside the SecretMessage as the SecretMessage is passed from method to method, stored in fields, returned from methods, and otherwise passed around the system. Because it has a run() method, that method can eventually be invoked to get the desired behavior from of the object.

In fact, by the time that this object makes it to the decoder, we might have lost track of the fact that it is a SecretMessage. Suppose that we have an object `toBeRun`, and all that we know about it is that it is a Runnable. We can still ask

```java
decoder.decode( toBeRun );
```

And now we might find out, for example, that someone has replaced our message with some Fireworks:

```java
public class Fireworks implements Runnable
{
    private Color color;

    public Fireworks( Color color )
    {
        this.color = color;
    }

    public void run()
    {
        Console.println( "Crash! Bang! You see "+ this.color.toString() );
    }
}
```

Polymorphic dispatch ensures that `toBeRun` will print its message if it is a SecretMessage, and will explode colorfully if it is Fireworks. You do not need to know what kind of thing it is to arrange to send it
to the right method; instead, Java's dispatch mechanism ensures that even when you don't know exactly what type of thing you have, the right method will be invoked.

**Callbacks**

A particular circumstance in which this "do the right thing" aspect of Java's method dispatch is important is called **callbacks**. A callback is a situation in which one object has invoked a method of another, and the second object needs to get some information back to the first without returning from the method invocation. There are a few prerequisites for callbacks:

1. The invoking object must pass a reference to itself into the original invocation, or must otherwise indicate whose method is to be "called back."
2. The invoking method and the invoked method must agree upon the name of the callback method.
3. The invoked method must record the reference to the invoking object -- the callback target -- e.g., as a parameter to the original invocation or as a field.
4. At the appropriate occasion, the invoked method must invoke the callback method on the callback target. The fixed method name is used in this expression; the reference to the callback target is a variable.

Suppose, for example, that we have an object whose purpose is to create many separate "web spiders", simple programs that traverse the Internet looking for interesting information.[Footnote: Such programs can be very useful, but you must be extremely careful in writing them. Serious disasters have been caused by web spiders that got out of control, for example creating so many spiders that the network filled up with spiders and couldn't sustain its regular traffic.] Your original object will want to know when the spider finds interesting information. But the spider won't want to stop executing when it finds the first interesting piece of information. Instead, the spider should take the address of its sponsor with it when it goes crawling through the web, and any time it finds an interesting piece of information it should "call back" the sponsor object, giving it that information without stopping its execution.

The actual situation for a web spider is a little bit more complicated than this description because web spiders often don't run on the same computer as their sponsor and so can't make direct method calls. But we can use this idea as the framework for some code that illustrates callbacks.

```java
public class SpiderStarter {

    private String interestingStuff = "";

    public void startSpider() {
        new Spider( this ); // give invoked method a reference to the invoker, i.e., the callback target
    }

    /* informationFound is the callback method.
    * It simply records the information...
    */
    public void informationFound( String interestingItem ) {
    }
}
```

if ( this.interestingStuff == null )
{
    this.interestingStuff = interestingItem;
}
else
{
    this.interestingStuff = this.interestingStuff
    + " and also "
    + interestingItem;
}
}

/* This is a simple utility method. */
public void printInfoSoFar()
{
    Console.println( "I heard " + this.interestingStuff );
}

This class provides three methods. The first starts up a Spider, telling the Spider who its sponsor is. The
second provides a way for the Spider to call it back (when it finds information). The third provides a way
for other objects to ask the SpiderStarter to let it know what information it has collected.[Footnote: Strictly
speaking, this code might be subject to problems if we start up more than one Spider. We really need to
protect the interestingStuff using synchronization, as described in part 5 of this book. These issues don't
affect the main point of this chapter, but you should be aware of them if you want to run a code example
like this one.]

The definition for Spider might read

public class Spider extends AnimateObject
{

    // where to record the callback target
    private SpiderStarter sponsor;

    public Spider( SpiderStarter who )
    {
        this.sponsor = who; // record the callback target
    }

    public void act()
    {
        // Some code that looks for interesting stuff.
        // if you find it, call back

        this.sponsor.informationFound( interestingInfo );
    }
}
Now, we might say

```java
SpiderStarter mamaTarantula = new SpiderStarter();
mamaTarantula.startSpider();
```

This starts a spider going. The "looking for interesting stuff" part of the Spider is missing, but we can still see how a Spider might take advantage of the callback mechanism. Since a Spider is an AnimateObject, its act() method will be executed over and over again. Each time, if it finds some interesting information, it will invoke its sponsor's informationFound method with the interesting information. But SpiderStarter's informationFound method just adds the new information to its information store and returns, so the AnimatorThread that runs the Spider AnimateObject is free to call its act() object again.

Consider trying to write Spider without the callback. SpiderStarter doesn't call a method of Spider's directly, so Spider can't return a String that way. Even if SpiderStarter did call Spider directly, mamaTarantula presumably wants the Spiders to keep going even after they find their first piece of interesting information. So it is very important that the individual Spiders have a way to get information back without stopping their own execution. This is precisely the kind of situation in which a callback is useful.

Callbacks are a very general mechanism that can be used any time one object needs to get information to its invoker without returning the information directly. They require agreement on the name of a method -- perhaps specified by an interface contract -- that will be used to produce the callback. Callbacks take advantage of the idea that Java's dispatch mechanism will call the appropriate piece of code. Good object encapsulation ensures that the information supplied in a callback gets to the appropriate place.

**Recursion**

One final example of how Java's method dispatch mechanisms work is the idea of **recursion**. Recursion is the name for a technique in which the same named method is called over and over again, doing something slightly different each time. There are two kinds of recursion: structural recursion, which is quite common in Java and other object-oriented programming languages, and functional recursion, which is much more prevalent in functional programming languages.

**Structural Recursion**

Structural recursion is a natural extension of method dispatch to a uniform collection of objects. It is really just the idea that an object can act on its own behalf -- i.e. provides methods specifying its own behavior -- coupled with the idea that one object can contain -- or have fields that are -- other objects. For example, the calculator had (access to) many CalculatorButton objects, and it relied on them to each provide the appropriate behavior. Structural recursion is just like this, except that the object doing the relying and the component object on which it relies are instances of the same class.
A Recursive Class Definition

Suppose, for example, that we have a class called LinkedList:

```
public class LinkedList {
    private Object contents;
    private LinkedList next;

    public LinkedList(Object c) {
        contents = c;
    }

    public Object getContent() {
        return contents;
    }

    public void setContent(Object c) {
        contents = c;
    }
}
```
public class LinkedList 
{
    private LinkedList next;
    private Object contents;

    public LinkedList( Object what, LinkedList next ) 
    { 
        this.contents = what;
        this.next = next;
    }

    // maybe some methods....
}

To begin with, this definition is recursive. That is, the LinkedList type is defined in terms of itself. Note that this isn't at all the same thing as saying that a particular LinkedList is defined in terms of itself; it just means that a LinkedList consists of its contents (some arbitrary object) and its next element, which is either nothing (i.e., this is the last element) or also a LinkedList.

The idea of an object that has associates -- or contains components -- of the same type really isn't all that strange. For example, if we have a representation for a person, we might use the same representation for that person's parents. The same "method" for figuring out who your father is should apply equally well to figure out who his father is.

To create a LinkedList, you need to give it a LinkedList. To make this work, there needs to be a simple case that is not explicitly recursive. This is called a base case. In the case of the LinkedList definition, the base case is null: null is a (non)value that can be associated with a name of type LinkedList that is not defined in terms of a LinkedList. A LinkedList with a null next field is the last element in the list.

So, for example, we can say

    LinkedList shorty = new LinkedList( "Not least", null );

We can also say

    LinkedList list = new LinkedList( "Pen Ultimate", shorty );

or even

    list = new LinkedList( "First and foremost",
         new LinkedList( "Sandwich filling", list ) );

Each of these LinkedList objects either has a next field that refers to another LinkedList object, or has a next field that is unassigned, i.e., has the value null.

Methods and Recursive Structure
Structural recursion is simply a way in which methods can take advantage of the recursive definition of LinkedList. It relies on the idea that each of the recursively contained objects is itself a full-fledged intelligent entity. For example, suppose that you are providing a LinkedList with a method to convert itself to a String. This method might, e.g., be suitable for printing out all of the elements contained in a LinkedList. Since one LinkedList contains another (through its next field), we can make use of the fact that that next element is also an intelligent LinkedList and will be able to convert itself to a String as well.

In writing the code to convert a particular LinkedList instance to a String, there are two possibilities.

1. Perhaps this is the last element in the list, i.e., this LinkedList object's next field is null. Then we can solve this problem simply: just convert the contents of this object to a String.
2. Otherwise, this is not the last element; this object contains a non-null next field. In this case, converting this LinkedList to a String requires converting the contents of this object, then adding a comma, then converting this object's next (LinkedList) to a String. But that next LinkedList is an intelligent object, too. We can just ask it to convert itself!

It may seem like there's a bit of sleight of hand going on here. This argument may look suspiciously like a circular definition. But it is not. Let's examine the logic here carefully.

The first of these is the simple case in which there is no further recursion. As in the definition, this is called the base case. This condition would apply if we asked the LinkedList labeled shorty to print itself -- i.e., if we invoked shorty.toString() -- which would return the String "Not Least". There is only one element in this list, so printing its contents suffices.

The second case is called the recursive case, the case that relies on recursion to work. It says, roughly, I know how to convert myself to a String, and my next knows how to convert itself to a String, so I will simply combine those two answers. Of course, the way that the next LinkedList element converts itself to a String relies on this same code....so here it is. Imagine this definition inside the class LinkedList, where the comment says maybe some methods....

```java
public String toString() {
    if ( this.next == null ) {
        return this.contents.toString();
    } else {
        return this.contents.toString()
            + ', ' + this.next.toString();
    }
}
```

Suppose that we invoke list.toString(). In this case, the object referred to by the name list has contents "First and foremost", so it would begin its answer with that String. But that's not enough. Because list's next field isn't null, it also needs to do something about that next field. It can't complete its answer until it knows how to print the LinkedList that is its next field. Luckily, list.next is also a LinkedList, so it knows how to convert itself to a String. So after "First and foremost", list adds in a comma. Then list invokes its next field's toString() method to find out how to end its String.
When `list.next's toString()` method is invoked, it checks to see whether its `next` field is null. Since it isn't, it can't use the base case. So it first converts its own contents into a String -- "Sandwich filling" -- and then adds a comma, and then asks its `next` field to convert itself to a String.

Once again, the LinkedList has a non-null `next` field, so once again the recursive case is invoked, creating "Pen Ultimate" + ", " + the value of its `next` field's `toString()` method.

The `next` field of this LinkedList is the same object referred to by the name shorty. We've already seen how shorty converts itself to a String using the base case -- returning "Not least" -- so now we can finish off "Pen Ultimate" + ", " + "Not least". This is returned to `list.next`, completing "Sandwich filling, Pen Ultimate, Not least". Finally, this String is returned to the LinkedList labelled `list`, and that LinkedList can return its value as a String: "First and foremost, Sandwich filling, Pen Ultimate, Not least".

The Power of Recursive Structure

The power of recursion here comes from the fact that each of the individual LinkedList elements knows how to combine its `next` field's `toString()` with its own contents. "If only my `next` field could supply its `toString()`," the LinkedList seems to say, "I could produce my answer. But of course the answer for the `next` field can be constructed out of its contents and its `next` field, and so on, until we come to the base case: a LinkedList in which the `next` field is null, so there's no need to get its `toString()`.

[Important] Note that it is crucially important that the recursive case invoke the same-named method on a simpler object. That is, each recursive step must get a little bit closer to the base case. Imagine instead a situation in which you were printing a circular LinkedList. In this case, there would always be a `next` LinkedList to print, and the process would never end.[Footnote: Actually, to prevent just such situations, the computer may have the ability to detect this circumstance -- an infinite loop -- and to object to it by raising an exception.]

A similar kind of structural recursion could be used to find out whether a particular object is contained in a LinkedList. In this case, there are actually two base cases.

1. If `this.contents` is the desired object, then the LinkedList contains that object, i.e., return true.
2. If `this.contents` is not the desired object, but `this.next` is null, then this LinkedList doesn't contain the desired object, i.e., return false.
3. Otherwise, since `this.contents` is not the desired object, this LinkedList contains the desired object exactly when the desired object is contained by the LinkedList `this.next`.

There's a fairly straightforward translation of this into Java code:

```java
public boolean contains( Object what )
{
    if ( this.contents == what )
    {
        return true;
    }
    else if ( this.next == null )
    {
        return false;
    }
```
else
{
    return this.next.contains( what );
}
}

[Footnote: Actually, Java's && and || operators are guaranteed to evaluate their operands from left to write, proceeding only until the value of the expression is known. In the case of &&, as soon as one operand is false, no further operands need be evaluated. In the case of ||, evaluation stops as soon as an operand is true. This means that we could rewrite contains as:

    public boolean contains( Object what )
    {
        return ( ( this.contents == what )
        || ( ( this.next != null )
            && this.next.contains( what ) ) );
    }
]

Structural recursion is an extension of "the object can handle it" to the case in which the method invocation expression is contained within the same method that it invokes. Because the target of the invoked method is a "simpler" object -- one that is somehow closer to the base case -- this approach ultimately produces a satisfactory answer.

Functional Recursion

Functional recursion is a further extension of the idea of recursion. In this case, there is no structure whose inherently recursive nature is exploited by the recursion. Instead, the necessary subsequent simplifications -- steps to get closer to the base case -- happen in one of the method's arguments.

For example, many kinds of numerical calculations can be performed using purely functional recursion. In this case, it is common to define one or more base cases -- e.g., how the function should behave on a simple number such as 1 -- and then to recursively build a solution for one number out of the solution for a smaller number. Factorial is one such function:

1. The factorial of 1 is 1.
2. The factorial of an arbitrary number, n, is n times the factorial of n-1.

The first of these is the base case. It simply produces an answer, with no recursion necessary. The second of these is the recursive case. It wishfully assumes that you know how to calculate the factorial of n-1, then uses that to construct the factorial of n. By "peeling off" one number at a time, it is possible to calculate the factorial of any number. This is really just like structural recursion, but there's no change of the method's target here.

    public int factorial( int n )
    {
        if ( n == 1 )
        {
            return 1;
        }
    }
else
{
    return n * this.factorial( n - 1 );
}
}

Factorial of 5 is 5*factorial of 4, which is 4*factorial of 3, and so on until factorial of 1, which is 1. So factorial of 2 is 2*1, and factorial of 3 is 3*(2*1), of 4 is 4*(3*2*1), and of 5 is 5*4*3*2*1. This is just like LinkedList's toString() method, except that the accumulation isn't coming from changing the target of the method invocation.

Chapter Summary

- Objects encapsulate information necessary to make methods effective.
- When multiple classes have methods with the same name, Java chooses the method that matches the target's (most specific) type.
- Dispatch can be replaced by empowering objects directly. Depending on the type of the target object, the same textual method invocation will actually call different code. This is called method polymorphism.
- A common superclass or interface, providing the method signature for the polymorphic method, is required for this kind of implicit dispatch.
- Method dispatch based on the target object can be used for other purposes as well:
  - Behavior can be passed to methods, returned from methods, and stored in objects by making it the run method of a Runnable object.
  - An executing method can give information to the object that called it, without returning, by using an explicitly agreed upon callback method.
- Recursion is a situation in which one method name is invoked repeatedly.
  - In structural recursion, the target of the method varies.
  - In functional recursion, at least one of the method's arguments varies.
  - In all recursions, there must be a base case that does not involve recursion.
  - In the recursive case, the recursive call must be to a method/target/argument that is somehow closer to the base case.

Exercises

1. Write toString() methods for an Address object and for a Date object. How would printObject have to change if it might be asked to print an Address or a Date as well as a String, Point, or Dimension?

2. Write clone() methods for Point and Dimension. (A clone() method should create a new copy of its target object.) Write a dispatcher called cloneObject( Object o ).

3. Write an animate AlarmedTimer class that counts by itself, as the Timer class of chapter 9 does. In addition, it should have a setAlarm( int interval, Alarmable who ) method. When this method is invoked,
the AlarmedTimer should callback the Alarmable's alarmReached() method every int ticks. Here is Alarmable:

```java
public interface Alarmable
{
    public void alarmReached();
}
```

4. Using the LinkedList code above, add a method that returns the Object that is the contents of the last element in a LinkedList. For example, `list.getLast()` would return "Not least", as would `shorty.getLast()`.

5. Define a recursive structure for a family tree. Each person in the tree should have a father and a mother, which should be either another person or -- e.g., if the information were not available -- null. Give this a method that prints all ancestors of a given individual.

Bonus: Give this structure the ability to print only all female ancestors (using Console.println).

Extra Bonus: Would your female-ancestor-printer print my father's mother?

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Event-Driven Programming

Chapter Overview

- How do we design an entity to emphasize its responses to various events?

In previous chapters, we have seen how an animate object can use its explicit control loop as a dispatcher, calling appropriate methods depending on what input it receives. In this chapter, we discuss a style of programming that shifts the emphasis from the dispatcher to the various handler methods called by that control loop. Entities designed in this way highlight their responses to a variety of situations, now called events. An implicit -- behind-the-scenes -- control loop dispatches to these event handler methods.

This event-driven style of programming is very commonly used in graphical user interfaces (GUIs). In Java, AWT's paint methods are an example of this kind of event-driven programming. This chapter closes with an exploration of a portion of the `java.awt` package, including `java.awt.Component` and its subclasses, to illustrate the structure of programs written in an event-driven style.

Objectives of this Chapter

1. To recognize event-driven control
2. To understand that event handlers describe responses to events, not their causes
3. To be able to write event handlers for simple event-driven systems

Control Loops and Handler Methods

In chapter 11, we looked at mechanisms for explicit dispatch. In that chapter, the job of the central control loop was to decide what needs to be done and then to call a helper procedure to do it. In this way, a single control loop can handle a variety of different inputs or circumstances. We saw, for example, how a calculator might respond differently to a digit, an operation, or another button such as =. The calculator's central control loop acts as a manager, routing work to the appropriate procedures. The actual work is accomplished by these helpers, or handler methods.

In this chapter, we will look at the same kind of architecture from a different viewpoint. Instead of focusing on the central control loop's role as a dispatcher, we will take that function largely for granted and look instead at control from the perspective of the handler methods. In other words, we will explore how one writes handlers for special circumstances, assuming that these handler methods will be called when they are needed. By the end of this chapter, we will turn to a system in which this is true without programmer effort, i.e., in which Java takes responsibility for ensuring that the handler methods are called when they are needed.
The basic idea of **event-driven programming** is simply to create objects with methods that handle the appropriate events or circumstances, without explicit attention to how or when these methods will be called. These helper methods provide answers to questions of the form, "What should I do when xxx happens?" Because xxx is a "thing that happens", or an **event**, these methods are sometimes called **event handlers**. As the writer of event handler methods, you expect that the event handlers will somehow (automatically) be invoked whenever the appropriate thing needs dealing with, i.e., whenever the appropriate event arises. [Footnote: Ensuring that those event handler methods will be called is a precondition for event-driven programming, not a part of it. We will return to the question of precisely how this can be accomplished later in this chapter.]

The result of this transformation is that your code focuses on the occasions when something of interest happens -- instead of the times when nothing much is going on -- and on how it should respond to these circumstances. An event is, after all, simply something (significant) that happens. This style of programming is called event-driven because the methods that you write -- the event handlers -- are the instructions for how to respond to events. The dispatcher -- whether central control loop or otherwise -- is a part of the background; the event handlers drive the code.

### Dispatch Revisited

Consider the case of an Alarm, such as might be part of an AlarmClock system. The Alarm receives two kinds of signals: SIGNAL_TIMEOUT, which indicates that it is time for the Alarm to start ringing, and SIGNAL_RESET, which indicates that it is time for the Alarm to stop. We might implement this using two methods, handleTimeout() and handleReset().

```java
public class Alarm
{
    Buzzer bzzz = new Buzzer();

    public void handleTimeout()
    {
        this.bzzz.startRinging();
    }

    public void handleReset()
    {
        this.bzzz.stopRinging();
    }
}
```

How do these methods get called? In a traditional control loop architecture, this might be accomplished using a dispatch loop. For example, we might make Alarm an Animate and give it its own AnimatorThread. The job of the dispatch loop would be to wait for and processes incoming
(timeout and reset) signals. This AnimateAlarm's act() method might say:

```java
public class AnimateAlarm extends AnimateObject {
    Buzzer bzzz = new Buzzer();
    public void handleTimeout()
    {  
        this.bzzz.startRinging();
    }
    public void handleReset()
    {  
        this.bzzz.stopRinging();
    }
    public void act()
    {  
        int signal = getNextSignal();
        switch (signal) {
        case SIGNAL_TIMEOUT:  
        this.handleTimeout();
        break;
        case SIGNAL_RESET: 
        this.handleReset();
        break;
        // Maybe other signals, too....
        }
    }
}
```

Of course, the real work is still done by the handleTimeout() and handleReset() methods. The job of the dispatch loop (or other calling code) is simply to decide which helper (handler) method needs to be called. The dispatcher -- this act() method -- is only there to make sure that handleTimeout() and handleReset() are called appropriately.

Figure #. An active Alarm object, invoking its own methods.
Simple Event Handling

What would happen if we shifted the focus to the helper procedures? What if we made the dispatch code invisible? Imagine writing code (such as this Alarm) in which you could be sure that the helper methods would be called automatically whenever the appropriate condition arose. In the case of the Alarm, we would not have to write the act method or switch statement above at all. We would simply equip our Alarm with the appropriate helper methods -- `handleTimeout()` and `handleReset()` -- and then make sure that the notifier mechanism knew to call these methods when the appropriate circumstances arose. This is precisely what event-driven programming does.

A Handler Interface

We have said that event-driven programming is a style of programming in which your code provides event handlers and some (as yet unexplained) event dispatcher invokes these event handler methods at the appropriate time. This means that the event dispatcher and the object with the event handler methods will need a way to communicate. To specify the contract between the event dispatcher and the event handler, we generally use an interface specifying the signatures of the event handler methods. This way, the event dispatcher doesn't need to know anything about the event handlers except that they exist and satisfy the appropriate contract.

In the case of the alarm, this interface might specify the two methods we've described, `handleTimeout()` and `handleReset()`:

```java
public interface TimeoutResettable
{
    public abstract void handleTimeout();
    public abstract void handleReset();
}
```

![Diagram of an Alarm that handles two event types.]

Of course, we'll have to modify our definition of Alarm to say that it implements TimeoutResettable:

```java
public class Alarm implements TimeoutResettable
{
    Buzzer bzzz = new Buzzer();
}
```
public void handleTimeout()
{
    this.bzzz.startRinging();
}

public void handleReset()
{
    this.bzzz.stopRinging();
}

Note that this is a modification of our original Alarm, not of the AnimateAlarm class. The TimeoutResettable Alarm need not be Animate. In fact, if it is truly event-driven, it will not be.

This TimeoutResettable Alarm definition works as long as some mechanism -- which we will not worry about just yet -- takes responsibility for dispatching handleTimeout() and handleReset() calls as appropriate. That dispatcher mechanism can rely on the fact that our Alarm is a TimeoutResettable, i.e., that it provides implementations for these methods. The dispatcher that invokes handleTimeout() and handleReset() need not know anything about the Alarm other than that it is a TimeoutResettable.

An Unrealistic Dispatcher

How might our TimeoutResettable Alarm be invoked? There are many answers, and we will see a few later. For now, though, it is worth looking at one simple answer to get the sense that this really can be done.

A simple -- and not very realistic -- event dispatcher might look a lot like the act method of AnimateAlarm. To make it more generic, we will separate that method and encapsulate it inside its own object. We will also give that object access to its event handler using the TimeoutResettable interface. Major differences between this code and AnimateAlarm are highlighted. Of course, the dispatcher doesn't have its own handler methods; its constructor requires a TimeoutResettable to provide those.

    public class TimeoutResetDispatcher extends AnimateObject
    {
        private TimeoutResettable eventHandler;

        public TimeoutResetDispatcher( TimeoutResettable eventHandler )
        {
            this.eventHandler = eventHandler;
        }

        public void act()
        {
            int signal = getNextSignal();
            switch (signal)
            {
                case SIGNAL_TIMEOUT:
                    this.eventHandler.handleTimeout();
                    break;
                case SIGNAL_RESET:
                    this.eventHandler.handleReset();
                    break;
            }
        }
    }
The details of this dispatcher are rather unrealistic. For one thing, it is extremely specific to the type of event, and extremely general to its event handler dispatchees. More importantly, in event-driven programming it is quite common not to actually see the dispatcher.

But dispatchers in real event-driven programs play the same role that this piece of code does in many ways. For example, the dispatcher doesn't know much about the object that will actually be handling the events, beyond the fact that it implements the specified event-handling contract. This dispatcher can invoke `handleTimeout()` and `handleReset()` methods for any `TimeoutResettable`, provided that the appropriate `TimeoutResettable` is provided at construction time. Different dispatchers might dispatch to different `Alarms`. In fact, timeout and reset are sufficiently general events that other types of objects might rely on them.

### Sharing the Interface

![ImageAnimation example](image)

**Figure #.** An `ImageAnimation` is a single component that displays a sequence of images, one at a time. For example, these frames, displayed in an `ImageAnimation`, would give the impression of a clock whose hands move.

Another object that might be an event-driven user of timeouts and resets -- and be controlled by the `TimeoutResetDispatcher` -- is an image animation. An image animation is a series of images, displayed one after the other, that give the impression of motion. In this case, we use the timeout event to cause the next image to be displayed, while reset restores the image sequence to the beginning. `ImageAnimation` simply provides implementations of these methods without worrying about how or when they will be invoked.

```java
public class ImageAnimation implements TimeoutResettable {
    private Image[] frames;
    private int currentFrameIndex = 0;

    // To be continued...
}
```

The image array `frames` will hold the sequence of images to be displayed during the animation. When the `ImageAnimation` is asked to paint (or display) itself, it will draw the `Image` labeled by `this.frames[this.currentFrameIndex]`. By changing `this.currentFrameIndex`, we can change what is currently displayed. When we do change `this.currentFrameIndex`, we can make that change apparent by invoking the `ImageAnimation`'s `repaint()` method, which causes the `ImageAnimation` to display the image associated with `this.frames[this.currentFrameIndex]`.

We omit the setup code that loads the Images into frames and handles other construction details.

The next segment of code is the timeout event handler, the helper method that is called when a timeout occurs. What should the `ImageAnimation` do when a timeout is received? Note that the

---

http://www.cs101.org/ipij/events.html
question is not how to determine whether a timeout has occurred, but what to do when it has. This is the fundamental premise behind event-driven programming: the event handler method will be called when appropriate. The event handler simply provides the instructions for what to do when the event happens. When a timeout occurs, it is time to advance to the next frame of the animation:

```java
public void handleTimeout()
{
    if (this.currentFrameIndex < (this.frames.length - 1))
    {
        this.currentFrameIndex = this.currentFrameIndex + 1;
        this.repaint();
    }
}
```

This code checks to see whether there are any frames left. If the animation is already at the end of the sequence, the execution skips the if clause and -- since there is no else clause -- does nothing. Otherwise -- if there's a next frame -- the execution increments the current frame counter, setting up the next frame to be drawn. Then, it calls `this.repaint()`, the method that causes the ImageAnimation to be redrawn. Recall that the ImageAnimation paints itself using the image that is associated with `this.frames[this.currentFrameIndex]`.

What about a reset? What should the ImageAnimation do when it receives the signal to reset? Handling a reset event is much like handling a timeout, but even simpler. The ImageAnimation simply returns to the first image in the sequence:

```java
public void handleReset()
{
    this.currentFrameIndex = 0;
    this.repaint();
}
```

No matter what, we reset the current frame index to 0, then repaint the image animation with the new frame. Note also that the next timeout will cause the frame to begin advancing again.

The code to actually repaint the image, which we have not shown here, makes `this.frames[this.currentFrameIndex]` appear. As a result, `handleTimeout()` works by changing the index to the next frame (until the end of the animation is reached); `handleReset()` restarts the image animation by restoring the index to the beginning index of `this.frames` once more.

Both Alarm and ImageAnimation are objects written in event-driven style. That is, they implement a contract that says "If you invoke my event handler method whenever the appropriate event arises, I will take care of responding to that event." Alternately, we think of the contract as saying "When the event in question happens, just let me know." When building both Alarm and ImageAnimation, the question to ask is, "What should I do when the specified event happens?"

**Real Event-Driven Programming**

We have seen other examples of event-driven coding style. In this section, we briefly review these and recast them in light of event-driven programming’s central question, "What should I do when xxx happens?" After reviewing these examples, we turn to look at the relationship of event providers to event handlers.

**Previous examples**

In chapter 9, we saw how an Animate’s act() method is repeatedly invoked by an AnimatorThread.
This `act()` method is in effect an event handler. It answers the question, "What should I do when it is time for me to act?" The Animate doesn't know who is invoking its `act()` method or how that invoker decided that it was time to act. It simply knows that it is, and how to respond to that knowledge, i.e., how to act(). The `act()` method may be invoked by an AnimatorThread instruction follower, executing at the same time as other parts of the system. It might equally well be invoked by a TurnTakerAnimator that controls a group of Animates and gives one Animate at a time a turn to act(). This latter approach might make sense, for example, in a board game where each player could move only when it was that player's turn.

Similarly, we saw how a Runnable object has a `run()` method that can be invoked in an event-driven style. This is commonly done when the `run()` method is invoked by starting up a new Thread. In this case, the Runnable's `run()` method is invoked when the Thread is started. From the perspective of the Runnable, its `run()` method is automatically invoked whenever it is time for the Runnable to "do its thing". In a self-animating object like a Clock, `run()` might be an event-handler-like method that is called by something "outside" (in this case, the Thread) when it is time for the Clock to begin execution.

The StringTransformer's `transform` methods of Interlude 1 were yet other examples of an event-driven style. These event handler methods simply answer the question, "What should I do when this StringTransformer is presented with a String to transform?" or "How do I respond to such a request?" These objects provide customized implementations for transforming strings. The decision of when to invoke these methods are outside the control of their owning objects.

In each of the cases described above, the event producer -- the thing that knows that it is time for a handler method to be invoked -- and the event handler -- which responds to the occurrence -- communicate fairly directly. For example, the TimeoutResetDispatcher polls (or explicitly asks) for signals and then directly invokes the event handler methods of its TimeoutResettable.

**The Idea of an Event Queue**

Event-driven programming by its very nature allows a more distant relationship between event producers and event consumers. Since the producer disavows responsibility for handling the event, it doesn't need to know or care who is taking on that responsibility; it merely needs to indicate that the event has arisen. The event handler doesn't really care where the event came from; it just needs to know that it will be invoked whenever the event has happened. This dissociation between event producers and event consumers is one of the potential benefits of programming in an event-driven style.

Systems that take advantage of this opportunity to separate event producers from event handlers generally contain an additional component, called the **event queue**, that serves as an intermediary. It is important to understand how the event queue can be used and the role that it plays as an intermediary between event producers and event handlers. Unless you are building your own event-driven system from the ground up, it is not important that you be able to build it. Generally, an event queue is provided as a part of any event-based system, and the major event-based systems in Java are no exception.
The role of the event queue is to serve as a drop-off place for events that need to be handled, sort-of like a To Do list. When an object produces behavior that constitutes an event, it reports that event to the event queue, which holds on to the event. The report of the event may be as simple as an indication that something happened ("Timeout!") or as complex as a complete description of the state of the world at the time that the event happened (e.g., the complete Wall Street Journal report on the stock market crash). What is important is that the event queue stores (remembers) this event report.

In addition to receiving event reports, the event queue also has an active instruction-follower that removes an event (typically the oldest one) from the queue and notifies any interested event handler methods. This is the queue-checker/dispatcher. An event queue also needs some way to figure out who to notify when an event has happened. In the cases that we explore in this chapter, there is always a single event queue per handler object, so it is always that object to which events are reported. In the next chapter, we will discuss a system that allows finer-grained control.

Consider the TimeoutResettable event handlers described above. A timer might generate the timeout events and deposit them into the queue. It would then return to its own business, keeping time and paying no more attention to the event queue. A separate instruction follower, the event dispatcher, would discover the timeout event in the queue and invoke the handleTimeout() method of the relevant party. The structure of this "queue cleaner" would be very similar to the TimeoutResetDispatcher we saw above.

Properties of Event Queues

This mechanism allows for a separation between the event producer and the event handler. The instruction-follower that puts an event into the queue -- the one who generates the event -- is not necessarily the instruction follower who performs the handler method (i.e., handles the event). Instead, one or more dedicated instruction followers have the task of processing events deposited into the queue, invoking the event handler method(s) as needed. Event suppliers need to know only about the event queue, not about the event handler methods.

Note that it is the event queue dispatcher's Thread (or instruction follower) that actually executes the steps of the event handler method. (Method invocation does not change which instruction follower is executing.) As a result, when you are writing event handlers, it is important that the event handler code complete and return (relatively) quickly; for example, it should not go into an infinite loop. [Footnote: A Runnable's run() method is an exception to this, because the Thread that executes run() has nothing to go back to doing. When run() completes, the execution of that Thread stops.] If the event dispatcher invoked an event handler that did not return, the dispatcher would be unable to process other events waiting in the queue.

You will almost never have to deal with an event queue explicitly unless you write your own event-driven system from scratch. Most programmers who write event-driven programs do not ever touch the event queue that underlies their systems. Instead, like many other aspects of event-driven programming, event queueing is generally a part of the hidden behavior of a system. However, there's nothing particularly mysterious about it. An event queue's contract provides an enqueue (add to the queue) operation and a dispatcher that actually invokes the event handler methods.[Footnote: In the next chapter, we will see that some event queues also provide an event listener registry service. This is not necessary in the event systems of this chapter, where there is a single event queue per handler object, but provides yet another layer of flexibility.]

In Java, the graphical user interface toolkit provides an event queue to handle screen events such as mouse clicking and button pressing. That event queue is fairly well hidden under the abstractions of the toolkit, so that you may not realize that it is an event queue at all. In the next chapter, we will explore that more complex system, which is used for most events in Java's windowing toolkit. That
system decouples the event handler from the object to whom the event happens, allowing one object to provide the handler for another's significant events. This is known as event delegation.

**Graphical User Interfaces: An Extended Example**

So far, we have left open the question of where and how events get generated. This is because in the most common kind of event system that you are likely to encounter -- a windowing system for a graphical user interface -- you do not deal with event generation directly. Instead, Java takes care of notifying the appropriate objects that an event of interest has occurred. When you are writing graphical user interfaces in Java, you will write event handlers without ever having to worry about when, where, and how the appropriate events are produced.

Before we can begin to talk about event handling in graphical user interfaces, we need to look briefly at what graphical user interfaces are and how they are built in Java. A graphical user interface -- sometimes called a GUI, pronounced "gooey" -- is a visual display containing windows, buttons, text boxes, and other "widgets". It is common to interact with a graphical user interface using a mouse, though a keyboard is often a useful adjunct. Graphical user interfaces became the standard interface for personal computers in the 1980s, though they were invented much earlier.

![Figure #. A sample graphical user interface.](http://www.cs101.org/ipij/events.html)

**java.awt**

Java provides a few different ways of making graphical user interfaces. In this section, we will take a look at the package java.awt. This package contains three major kinds of classes that are useful for making GUIs. The first of these is java.awt.Component and its subclasses. These are things that appear on your screen, like windows and buttons. The immediately following subsection explores this component hierarchy. The second major GUI class is the class java.awt.Graphics, which is involved in special kinds of drawing. We will return to java.awt.Graphics at the end of this chapter. The final group of classes are the event classes: the java.awt.Event class together with the classes in the java.awt.Event package. We'll come back and look at AWT Events in the event delegation chapter. Here we'll deal only with one (pseudo-)event, painting. In the remainder of this section, we are going to focus on Components and, in a bit, Graphics.

The event that we will be concerned with here is painting. That is, this is the event that occurs when a window or other user interface object becomes visible, is resized, or for other reasons needs to be redrawn. This event happens to a Component. In order to handle this event you need to know what the current state of the drawing is, including both its coordinate system and what if anything is currently visible. That information is held by a Graphics. So when the event happens, it takes a form roughly paraphrased as "paint yourself on this screen". The event handler belongs to a Component -- the "self" to paint -- and it takes a single argument, a Graphics -- the "screen" on which to paint.
Components

A component is a thing that can appear on your screen, like a window or a button. The parent of all component classes is `java.awt.Component`. The Component class embodies a screen presence. You can't have a vanilla Component, though; you can only have an instance of one of its subclasses. (In fact, `java.awt.Component` is an abstract class. See the sidebar in Chapter 7 for further detail on abstract classes.)

Although you can't instantiate Component directly, Component has several useful subclasses. One group of these is the set of stand-alone widgets that let you interact with your screen in stereotyped ways. There are many GUI widgets built in to `java.awt`. These include Checkbox, Choice, List, Button, Label, and Scrollbar. In addition, there are several Menu variants that don't extend Component directly, but also provide useful widgets. Each of these widgets is pretty well able to handle its GUI behavior -- showing up, disappearing, allowing selections to be made, etc. In the event delegation chapter, we will see how to use these GUI components to allow the user to communicate with your application; for example, to have something smart happen when a selection is made. (This involves customizing these widgets' event handlers.)

Another set of components are called Containers. These Components extend `java.awt.Container` (which itself is an abstract class extending `java.awt.Component`). Containers are components that can have other components inside them. For example, a `java.awt.Window` (which is a kind of component) can have a `java.awt.Scrollbar`.

In this chapter, we will confine ourselves to one simple component behavior: painting itself. To do this, we will use a generic Component, called Canvas, that you can instantiate. The `java.awt.Canvas` class doesn't do anything special, but you can either use it as a generic component or extend it to get specialized behavior. We will make a Canvas that paints itself with a special picture.

![Figure #. Standard screen coordinates, showing the origin, directions of increasing horizontal (x) and vertical (y) coordinates, and two other sample points.](image)

Graphics

A `java.awt.Graphics` (sometimes called a "graphics context") is a special kind of object that knows how to make pictures appear. A Graphics uses a coordinate system to keep track of locations within it. The origin of this coordinate system -- the point (0,0) -- is in the upper left-hand corner. Moving right from this point involves increasing the first (x) coordinate, so (100, 0) is 100 pixels to the right of the origin, along the top edge of the Graphics. [Footnote: A pixel, short for picture element, is the smallest visible unit on your computer's screen. A higher resolution display is one that has more pixels in the same amount of space, i.e., one with smaller pixels. Java Graphics are delineated in pixels.] Moving down increases the second (y) coordinate, so (0,50) is 50 pixels below the top of the Graphics, along its left-hand side. (100,50) is a point that is not on either the top or left edge; it is 100 pixels to the right and 50 pixels down.
Each Graphics has methods such as `drawLine`, `fillOval`, and `setColor` that allow you to create pictures. For example, if you had a Graphics named `g`, `g.fillOval(100,100,10,10)` would make it display a 10-pixel by 10-pixel circle with its upper left-hand corner at position 100, 100. If you called `g.setColor(Color.red)` first, the circle would be red. A complete list of the methods of a `java.awt.Graphics`, together with a brief description of each, can be found in the `java.awt.Graphics` reference.

A Graphics is not the kind of object that you are likely to create or have hanging around. You will probably never run into the Graphics associated with GUI widgets or containers. However, each time that your Canvas needs to redisplay itself, it will be handed a Graphics context with which to do that redisplaying. So there will be times when your code will be given a Graphics to use.

**The Story of paint**

Painting (itself) is what a GUI component does when it becomes visible. For example, if a window is (partially) covering a component and then the window is moved, the component needs to make itself look right again. Java takes care of automatically determining that this should happen and asks the component to paint itself.

Every `java.awt.Component` has an event-driven paint method. This method does not say when the component should be painted, nor why, nor on what. This method has nothing to do with determining that painting is necessary. Instead, this method is the set of instructions that describe how to paint the Component. It is the answer to the question, "What should I do when it is time to paint myself (on the provided Graphics screen)?" It is the job of whatever calls the paint method to determine whether and when the Component needs to be painted.

The paint method of a Component is passed a Graphics object. This is the Graphics which contains, among other things, the coordinate frame within which drawing on this Component should take place. It also contains a variety of utilities that will make things actually appear within the Component. Just as you don't have to determine when or whether paint should be invoked, you don't need to provide the Graphics object. Like magic, when paint is invoked, the Graphics object will be there.

Each paint method contains the specific instructions that that component needs to make itself appear. For example, a Button's paint method makes the button label appear on the button. A Window's paint method not only makes the Window appear, it also makes sure that the paint method of each of the components contained in the Window gets called as well.

When the paint method is invoked, it is equipped with a single argument, a Graphics. If what the Component does to display itself is, for example, to draw shapes, this Graphics (the argument passed in to the Component's paint method) is what actually does the drawing.

Your job, when implementing a paint method, is to make use of this provided Graphics (and any other information that the object may have) in order to make the correct picture appear. You supply the instructions to be executed. To paint me, make a big red dot. Or, to paint me, print my name. Or, to paint me, paint each of the Components that appear inside me.

Suppose that you want to have your Component contain a rectangle in the upper left-hand corner. A `Graphics` has a `drawRect` method which does just that. When your component's paint method is called, it should ask whatever `Graphics object` is supplied to it to `drawRect( int x, int y, int width, int height)`. [Footnote: `drawRect` takes four arguments: the upper left hand coordinates and the size coordinates. All measurements are in pixels -- tiny boxes that make up your screen -- and the origin -- the point (0,0) -- is in the upper left-hand corner of the component. These are called "screen coordinates". `Graphics objects` have lots of other drawing methods, too. See the

http://www.cs101.org/ipij/events.html
java.awt.Graphics documentation for a comprehensive listing.

For example, if paint were called with a Graphics named g, the instructions might read

```
g.fillRect(0,0,20,20);
```

to draw a square in the upper left-hand corner of the Component. The whole method would read

```
public void paint( Graphics g )
{
    g.fillRect(0,0,20,20);
}
```

A Component's paint method is an event handler. This means that the Component's paint method is
the set of instructions describing the Component's response to a request to redisplay itself. It triggers
every time Java finds that something has happened that requires the component to redisplay itself.

**Painting on Demand**

When we say that paint is an event handler method, what we mean in part is that your code doesn't
want to call paint directly. Instead, paint is called automatically by the Java runtime system any time the
Component needs to redisplay itself. This could happen, for example, if a window were covered up
and then uncovered: when the uncovering event occurs, the window needs to repaint itself. Each of
the components, containers, and widgets in java.awt has an event-driven paint method. Note,
however, that there's no Paintable interface; paint is a method of Component and is inherited by
every class that extends Component.

The paint method takes a Graphics context as an argument. You cannot, in general, supply the
appropriate Graphics context to a Component; but since you don't call paint, you don't need to
supply the Graphics. Instead, Java's behind the scenes bookkeeping takes care of this. (Remember,
paint(Graphics g) is used in event-driven style; that is, it is called by Java, not by your program.)

Your code cannot call paint directly. It is an event handler method and it uses an event queue; only
the queue manager can call paint. But sometimes you will know that it is necessary for a GUI object
to repaint itself. For example, in the code above the image animation needed to repaint itself each
time the currentFrameIndex changed. Since you can't call the component's event handler directly,
each Component provides another method, called repaint(), that you can call. If you call the
component's repaint() method, it will ask Java to send it a new paint event.

If you do ever need to tell the system that you want your component to be painted, you need to
arrange for Java to provide the appropriate information to your class. You can do this by calling the
component's repaint() method. Unlike paint, which takes a Graphics as an argument, repaint
takes no parameters. (This is good, because you don't generally have a Graphics around to give
to paint. This is another thing that Java keeps track of automatically.) You don't have to implement
repaint(); java.awt.Component.repaint(), which you will inherit, queues up a new paint
(Graphics g) request (even supplying the appropriate Graphics) behind the scenes. Remember:
**You never call paint, and you never implement repaint.** To cause a painting to happen, call
repaint(); to explain how to paint your component, implement (override) the paint(Graphics g)
method -- and don't worry about the Graphics, it will be automatically supplied to you!

**Events and Polymorphism**

One advantage of using an event-driven style is that your code can focus on how to respond to things
that happen. It does not have to spend a lot of time figuring out whether things happen or deciding
what has happened and who should deal with it. (Of course, event-driven code relies on an event
dispatcher, which does have to deal with these things, but often either one is available -- as in the
GUI case -- or a fairly simple and generic one will do.)

A second advantage of the event-driven style is that, when used in concert with an event queue (as in
Java's AWT), it separates the generator of the event (e.g., the window motion) from the handler of
the event (the component that is uncovered). This means that these two pieces of the system can be
designed independently. All they have to do is to agree on the event protocol that they will use (in
this case, repaint() and paint(Graphics g)). How each one fulfills its side of the contract -- how the
component decides to paint itself, for example -- is something that the rest of the system doesn't have
to worry about.

A corollary benefit, then, is that different kinds of components can handle the same event in very
different ways. We saw this early in this chapter where the same pair of events -- timeout and reset --
were used to run both an alarm and an image animation. In these two objects, the timeout event
meant very different things. The alarm handled a timeout by turning on its buzzer; the image
animation switched to the next image each time a timeout occurred.

The GUI painting system that we have described uses this polymorphism to great advantage. When a
component like a Canvas is asked to paint itself in a Graphics, it may draw a simple picture using the
Graphics supplied. When a widget like a Button is asked to paint itself, it creates labeled region of
the screen appropriate for clicking into. A Checkbox may paint itself as a square, with or without an
X in it depending on whether the Checkbox is checked. A container such as a Window not only
paints itself, it also asks each of the components contained inside it to paint themselves. The Window
doesn't need to know anything about how these components appear; it simply asks them to paint
themselves in the way that they know best.

Chapter Summary

- By hiding the central control loop, we shift emphasis from explicit dispatch to event handler
  methods.
- Event driven programming separates things that happen from how they're handled.
- Each object is free to implement the same event handler in a different, customized way.
- In Java's AWT, certain GUI events are automatically dispatched by the Java runtime.
- The root of the GUI component hierarchy is java.awt.Component. Although
  java.awt.Component is an abstract class, it has many useful subclasses, including
    o widgets such as Checkbox, Choice, List, Button, Label, and Scrollbar.
    o Containers, Components that can hold other Components.
    o Canvas, a generic Component that you can customize.
- Every java.awt.Component has a paint(Graphics g) method that is called by Java when
  Java needs to make the Component (re)appear on the screen.
    o By overriding or implementing a paint(Graphics g) method, you are describing how
      your custom component should handle requests to paint itself.
    o You don't generally call a component's paint method. (Among other things, you don't
      have a Graphics to pass it.) If you want to redraw a GUI component, you can call its
      repaint() method.

Exercises

1. Define a TimeoutResettable that simply prints to the Console whenever an event happens. The message printed should differ depending on which event occurs. Implement it in a purely event driven style, i.e., assuming that something else will manage the event dispatching.

2. Describe a scenario in which an event occurs to the object in the previous exercise. Explain the sequence of action.

3. Define a class that extends java.awt.Canvas and has an unfilled circle with its upper left hand corner at 100, 100.

   (Bonus) What happens if you make the Canvas very small? Can you modify your class to keep the circle centered on the Canvas? You can use Canvas's getSize() method, which returns a java.awt.Dimension with directly manipulable height and width fields.

4. Define a class that extends java.awt.Canvas and paints itself like a black-and-white checkerboard. You may assume that the dimensions of the Canvas are 400x400.

   (Bonus) Make the checkerboard red and black.

5. Define a class that extends java.awt.Canvas and has two different painting behaviors. (For example, it could paint a black circle or a red square.) This class should also have a changeMe method. Each time its changeMe method is called, it should redisplay itself using the other behavior (e.g., it should switch between a black circle and a red square). Hook this up to a Timer (from Chapter 8).

You can test your Canvases using the cs101.awt.DefaultFrame class included in the code supplement to this book.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Event Delegation and java.awt

Chapter Overview

- How do I separate an entity's core behavior (model) from its on-screen appearance (view)?
- How do intermediate (listener) objects couple together system components that don't know about one another?

In the simple event model of the previous chapter, each visible component provides an event handler method (e.g., `paint`) that is invoked every time that the appropriate event is triggered (e.g., by uncovering a window or by an explicit call to `repaint()`). The component doesn't (necessarily) have an always-active animacy (Thread); instead, it is woken -- invoked by the event dispatcher instruction follower -- whenever an appropriate event occurs.

In the previous chapter, we saw how event driven programming focuses a system's design on what to do when certain events happen. The mechanism that recognizes and dispatches these events fades into the background. We saw how this approach is used to implement painting in java.awt components. In that system, each Component handles its own events. In this chapter, we will look at a more complicated two-layer model which further separates the event producer from the event consumer. This mechanism, which relies on an explicit listener registration protocol, is at the heart of the event handling system in Java's AWT versions 1.1 and later.

The problem of GUI design is illustrative of larger design issues. The event-delegation approach described in this chapter arises from our desire to separate what happens in the GUI (such as clicking a button) from the behavior that this causes (such as playing a song). To make this work, we connect GUI objects (such as Buttons) to application objects (such as SongPlayers) indirectly, through special EventListener objects. The EventListener records the appropriate connection between GUI events and application behavior, keeping these details out of both GUI and application components. This allows significant flexibility: a single application behavior may be invoked by many different GUI events; one GUI event may give rise to many application behaviors; or the relationship between GUI events and application behavior may be remapped by a running program, for example.

This kind of indirect coupling through a Listener object is a useful technique in a wide range of applications.

Model/View: Separating GUI Behavior from Application Behavior

In the previous chapter we explored event-driven programming as a way of focusing on the important things that happen in a program. An event handler is a method that responds to some important circumstance, or event. It answers the question, "What should I do when xxx happens?" It shifts the emphasis from figuring out what has happened and deciding what to do (the dispatcher) to the actual code...
that handles the event, whenever it may arise. Event driven programming is the idea that an object simply provides an event handler method -- instructions to follow -- and does not worry about how or when those instructions are executed. Somehow, an instruction-follower will invoke this method -- and follow its instructions -- when appropriate.

Java's AWT graphical user interface toolkit uses event-driven programming to coordinate the display of GUI objects on your computer screen. Each java.awt.Component implements its own paint(Graphics g) method, which supplies the instructions for making that Component appear in the coordinate space described by g. As in all event-driven programs, the event handler paint method does not worry about when, why, or whether it is time to paint. When the paint method is invoked, it means that the need for painting has arisen -- the event has occurred -- and the paint method's execution simply responds to that event.

In AWT painting, the need-to-paint event happens to a particular Component. When a need-to-paint event arises, AWT makes it clear who is responsible for handling that event: the Component that needs to be painted. But there are many other kinds of events for which the question, "Who should handle this event?" does not have such an obvious answer. This chapter is about more general mechanisms that let programmers answer that question in a more flexible way, separating the Component to which the event happens from the object that handles the happening.

In many cases, the appearance of a GUI object and its underlying behavior may actually be implemented by two different Java Objects. For example, the GUI object that implements a set of radio buttons may be a Panel containing a number of Checkboxes. This is called the view: what the mechanism looks like, its screen appearance. In addition, when the appropriate buttons are pressed, a song may be played. This is called the model: how the mechanism behaves. The view -- in this case, the Panel -- is responsible for keeping track of the on-screen appearance of the CheckBoxes (with their help, of course). The Panel need not be responsible for playing the song, though. The model, which provides the song-playing behavior, may in fact be implemented by a different object. Logically, we want to separate out the GUI appearance (and GUI behavior, e.g., buttons looking pressed or not pressed) from the underlying application behavior. Java's AWT event delegation mechanism lets us do just that.[Footnote: The event delegation mechanism described in this chapter is used in Java's AWT version 1.1 and later and also in the Java Swing toolkit. In Java's AWT version 1.0, all event handling was done using a system closer to that of chapter 15.]
The Event Queue, Revisited

In the previous chapter, we saw that we can separate the generator of an event from the actual invocation of an event handler through the use of an event queue. The event queue is a place where an event producer can "drop off" the information that an event had occurred. For example, code can call a Component's repaint() method. This adds a painting request to the event queue. Paint requests can also get added to the queue by screen events, such as a Window moving to uncover a Component or a new Window being asked to show(). Inside the queue, it doesn't matter how the event got added. A separate active event dispatcher looks at the requests in the queue and figures out which event handlers need to be called when. The event dispatcher picks up an event (or, in the case of repaint requests, perhaps several requests) and invokes the appropriate method (e.g., paint(Graphics g)).

In the case of painting, you can imagine that there is one event queue per Component. The dispatcher doesn't need to figure out what code to call; all of the requests in that queue are for the associated Component. When a need-to-paint request arises, Java ensures that that Component's paint(Graphics ) method is called. The Component doesn't have to do anything more than provide a (possibly inherited) implementation for this method.

All GUI events -- not just painting -- happen to particular Components. The mouse is clicked inside a particular Component. Only one Component at a time can be listening to the keyboard.(Being the Component that is listening to the keyboard is called "having the focus"). So when an event occurs, it will still get added to the queue belonging to the Component with which it is associated.

But suppose that we want to separate even event ownership from the responsibility for handling the event. Suppose, for example, that clicking a radio button (GUI Component) causes another object -- a SongPlayer -- to play a song. If responsibility for handling the event doesn't necessarily belong to the Component -- if we are separating the Component view from a distinct Object implementing the model -- the event queue's dispatcher needs to figure out who to notify that the event has occurred. We need a mechanism for associating the events that happen (and the objects to which they happen) with interested parties that are willing to handle those events. We call these interested parties **listeners**. The system by which a separate event-handling object listens for events that occur to another (GUI Component) object is sometimes called **event delegation**.

Java solves the "who to notify" problem by introducing the idea of listener registration. You can think of this as being something like subscribing to a newspaper clipping service or personalized online news service. When you subscribe to such a service, you give the service a list of topics that you're interested in. This is registering your interest with the event queue, or listening. The service maintains a list of subscribers along with their interests. These are the registered listeners. Each time that a new article comes in, it is added to the pile of clippings to be considered. This is putting an event into the queue. An employee of the clipping service picks up a clipping (typically the oldest one) and checks to see who might be interested. If the article matches your interests, the clipping service sends you a copy. This is dispatching to the event handler methods.
Events -- such as mouse clicks or being uncovered when a Window moves -- still happen to individual Components. But -- for many such GUI events -- each `java.awt.Component` has its own event queue that can dispatch to the appropriate registered event handlers. These event handlers need to know about and register with the Component whose events they want to listen for; they need to tell the event queue which events they are interested in handling. The Component maintains a list of listeners who will handle its events.

Registering a listener is like leaving a (specialized) request with the clipping service: If any articles about Indonesian coffee come, please send them to Working Joe, and if any mouse motion events occur, please send them to the mouse motion listener that's waiting for them.

### Reading What the User Types: An Example

Imagine that we want to have the user type her name into a GUI widget. When she does so, we will print a friendly greeting. This section walks through this example, providing a pragmatic introduction to the actual AWT mechanisms required to implement event delegation.

The code that follows assumes that it appears in a method within a class within a file that imports `cs101.awt.DefaultFrame`, `java.awt.TextField`, and `java.awt.event.ActionListener`, and `java.awt.event.ActionEvent`. In general in this chapter we will omit package names unless they are needed for clarity.

### Setting Up a User Interaction

The first thing that we need to do is to create a place where the user can type her name. Java provides an AWT widget that is useful for just such occasions, a TextField.

```java
TextField nameField = new TextField( "Type your name here" );
```

This line creates a TextField, a rectangular box containing text. The constructor argument is the text initially displayed in this box.

```
Type your name here
```

```java
nameField.setEditable( true ); // Make it possible for the user
                            // to type into the TextField.
nameField.selectAll();    // Highlight the original text so that
                          // what the user types replaces it.
```

The first of these lines makes it possible for the user to type in the TextField. The second highlights all of the text in the TextField, so that what the user types will replace the text displayed there.

```
Type your name here
```

```java
new DefaultFrame( nameField ).init();    // Create a Frame around the TextField.
```
Finally, this line creates a cs101.awt.DefaultFrame, an awt Window in which a single Component can be displayed. DefaultFrame is a restricted kind of Frame, but has the advantage that it takes care of certain housekeeping details for you. DefaultFrame's init() method actually makes the window appear on the screen. See the sidebar on DefaultFrame for details.

Now suppose that the user types her name into the TextField box, replacing the highlighted text previously displayed. If the user ends her name by typing the return key, this causes an action event to be registered on the TextField. In other words, something has happened and we are ready to invoke the appropriate event handler.

Now, we are ready to print our greeting. For example, we might say

```
Console.println( "Hello, " + reference_to_nameText.getText() );
```

Each TextField has a getText() method that returns the String displayed in the TextField at the time of the getText() invocation. So, if we execute code along these lines, the text

```
Hello, Galadriel
```

should appear on the Java Console. There are, of course, a few issues:

1. Where does this code appear? That is, who is handling the event, and in what method?
2. How does that event handler access the TextField called nameText (in order to ask it to getText())?

This is where Java's event delegation system comes in.

**cs101.awt.DefaultFrame**

A cs101.awt.DefaultFrame is a cs101 utility provided to make it easy to put up a window containing a single Component. The DefaultFrame takes care of sizing, activating the window's close box, causing the window to appear on the screen, etc.
If c is a Java component, it can be made to appear on the screen using

```java
new cs101.awt.DefaultFrame( c ).init();
    // Create a Frame around the component.
```

The first half of this statement is an object construction expression that creates a DefaultFrame around c. The second half of the statement invokes this DefaultFrame's init() method, which is useful for its side effect: it displays Component inside the DefaultFrame, i.e., in its own window. Of course, you can use a more complex version of this code that names the new DefaultFrame, allowing you to use it elsewhere in your program, if you wish:

```java
cs101.awt.DefaultFrame frame = new cs101.awt.DefaultFrame( c );
frame.init();
    // Create a Frame around the component.
```

The class cs101.awt.DefaultFrame extends java.awt.Frame, documented in the AWT Quick Reference appendix to this book. For the complete code implementing cs101.awt.DefaultFrame -- which is straightforward -- see the online supplement to this book.

**Listening for the Event**

The event generated by Galadriel's return is associated with the TextField called nameField. That TextField is like a clipping service, and a new item of potential interest -- the action taken by Galadriel -- has just arrived. Now, Java needs to determine who is interested in nameField's action events.

Who might be interested? There is a special interface, called ActionListener, that describes the contract to be implemented by any object interested in handling action events. Here is the definition of the ActionListener interface:

```java
public interface ActionListener extends EventListener {
    public void actionPerformed( ActionEvent ae );
}
```

The actionPerformed method is an event handler, so its implementation will answer the question, "What should I do when an action is performed?" In this case, the answer is to print out the text currently displayed by the TextField in which Galadriel typed her name. The object whose actionPerformed method is invoked is not responsible for deciding whether, when, or why the actionPerformed method should be called. It is only responsible for behaving appropriately when the event handler method is called.

We can build an action listener by providing a class that implements this interface. The implementation of actionPerformed in this class is an answer to the question, "What should I do when an action is performed?"

```java
public class FieldHandler implements ActionListener {
    private TextField whichText;
    public FieldHandler( TextField whichTextToHandle )
```
This class actually keeps track of which TextField it wants to associate itself with. We can create a particular FieldHandler associated with nameText using the construction expression

```java
new FieldHandler( nameText )
```

Now, when this FieldHandler's actionPerformed method is invoked -- when the action happens -- the FieldHandler will use nameText's getText() method to print a greeting to Galadriel.

Of course, we might want to hang on to that FieldHandler once we've created it....It will come in handy in another few paragraphs.

### Registering Listeners

So far, so good. However, we haven't specified how the FieldHandler gets notified about the event in the first place. Of course, part of the story is that Java's event manager identifies that a carriage return has been hit in the TextField and generates an appropriate ActionEvent. But this event happens to the TextField; how does the FieldHandler get hold of it?

The answer is that Java needs to be notified that the FieldHandler is interested in this TextField's action events. To return to our earlier analogy, the FieldHandler needs to subscribe to the TextField's action event clipping service.

This is accomplished with the TextField's addActionListener method, which takes an ActionListener as an argument. The addActionListener method tells Java that the ActionListener argument addActionListener is wants to know about any ActionEvents that occur to this TextField. For example,

```java
ActionListener nameHandler = new FieldHandler( nameText );
nameText.addActionListener( nameHandler );
```

[Footnote: or simply nameText.addActionListener( new FieldHandler( nameText ) );]

registers the ActionListener called nameHandler as a listener for any ActionEvents that occur to nameText.

Now, when Galadriel finishes typing, an action event will not only be generated but also forwarded to nameHandler to handle.
Recap

The code that creates this situation is distributed over the paragraphs above. Here is the entire setup code. It might, for example, appear in a main method or in the constructor of an entity that provided the name-greeting behavior described at the beginning of this section.

```java
// Set up the TextField.
TextField nameField = new TextField( "Type your name here" );
nameField.setEditable( true ); // Allows user typing.
nameField.selectAll(); // Highlights current text.

// Now create and register the ActionListener
ActionListener nameHandler = new FieldHandler( nameText );
nameText.addActionListener( nameHandler );

// Finally, create a Frame around the TextField.
new DefaultFrame( nameField );
```

The only additional code required is the FieldHandler definition:

```java
public class FieldHandler implements ActionListener {

    private TextField whichText;

    public FieldHandler( TextField whichTextToHandle )
    {
        this.whichText = whichTextToHandle;
    }

    public void actionPerformed( ActionEvent ae ) {
        Console.println( "Hello, " + this.whichText.getText() );
    }
}
```

Specialized Event Objects

In Galadriel's example, we encountered an object whose type was ActionEvent. It appears as a parameter in the actionPerformed method of ActionListener. In that example, we blithely ignored the ActionEvent -- as one often does in an action Performed method -- but this begs the question of what that object is and why it appears. In this section, we'll look at ActionEvent and other similar event objects, and explore cases in which these event objects have important roles to play.

In the previous chapter, we looked at an event handler method called paint. That method needed to be supplied with a fairly specific kind of object, a Graphics, before it could do anything. In contrast, other handler methods of the previous chapter -- such as handleTimeout() and handleReset() -- needed no arguments at all. The event handlers in this chapter do need some information, but that information is of a fairly generic (though specializable) type. The information supplied to one of these AWT event handlers is a special Java object called an AWTEvent. Such an object inherits from java.awt.AWTEvent (which is itself a java.util.EventObject). The subclasses of java.awt.AWTEvent live in a separate package, called java.awt.event.
In a general GUI, what kinds of things can happen? The mouse can be moved and clicked and dragged, the keys can be pressed, windows can be closed, menu items can be selected, text can be entered, and many, many more things can happen. A listing of the major event types used in this book may be found in the AWT Quick Reference appendix in the AWT Events segment. For example, a mouse click generates a MouseEvent, while clicking in the close box of a window generates a WindowEvent and clicking a button (or typing return in a text field) causes an ActionEvent.

Some kinds of events, like ActionEvents, are notable mostly for happening. For example, when a Button is clicked, an ActionEvent is generated. If you know what Button was clicked to generate the ActionEvent, you really know everything worth knowing about the ActionEvent. (If you don't know what Button was clicked, you can find out by asking the ActionEvent; see below.) An ActionEvent is also generated when the return key is typed in a TextField (as we have seen), indicating that the text is complete. In this case, you need to know both which TextField and, perhaps, what text was typed. But once you know what TextField generated the ActionEvent, you can ask the TextField for its text. So the internal structure of an ActionEvent is not likely to be of much interest.

Different kinds of events have methods that provide access to the different kinds of information that you'd want if you were dealing with a mouse click or a window close. These event methods are summarized in the AWT Events segment of the appendix AWT Quick Reference. For example, a MouseEvent has a few methods that are especially worth noting. If the MouseEvent is labelled mickey, then

- mickey.getX() returns an int specifying the mouse's location at the time of the MouseEvent (in pixels starting at the upper left-hand corner of mickey's screen-space).
- mickey.getY() similarly returns mickey's y coordinate.
- If you prefer to get both coordinates at once, you can retrieve a java.awt.Point object using mickey.getPoint().

Every AWTEvent also has a getSource() method. This method returns the Object to whom the event happened. For example, we could have replaced the actionPerformed method of our FieldHandler class with the definition

```java
public void actionPerformed( ActionEvent ae ) {
  TextField theField = (TextField) ae.getSource();
  Console.println( "Hello, " + theField.getText() );
}
```

This text uses the TextField that is the source of the action event, rather than the TextField that is handed to the FieldHandler constructor, as the target of the getText() method.[Footnote: In this case, we could simply eliminate the constructor, making the FieldHandler definition look like this:

```java
public class FieldHandler implements ActionListener {

  public void actionPerformed( ActionEvent ae ) {
    TextField theField = (TextField) ae.getSource();
    Console.println( "Hello, " + theField.getText() );
  }
}
```
Some AWTEvents, such as MouseEvent, are ComponentEvents. Every ComponentEvent also has a `getComponent()` method that returns the same thing as its `getSource()` method, but typed as a Component.

A variety of useful event types and their methods are documented in the AWT Events segment of the AWT Quick Reference appendix.

**Listeners and Adapters: A Pragmatic Detail**

Every AWTEvent type has an associated Listener type.[Footnote: Except PaintEvent, which uses the mechanism described in the previous chapter rather than the listener registration system described here.] This means that when the AWT event occurs -- the mouse is clicked or the key is pressed, etc. -- there's a type of object equipped to handle that event. (Actually, MouseEvent is an exception, as it has two associated listener types: MouseListener, which handles clicks, entry and exit, presses and releases, and MouseMotionListener, which handles drags and moves. Most event types only have one Listener.)

The ActionListener defined above will do the trick quite nicely for our TextField. The ActionListener interface only had a single method to implement. Other listener interfaces are more complex, though. For example, the MouseListener interface defines five methods:

```java
public interface MouseListener extends EventListener {
    public void mouseClicked( MouseEvent mickey );
    public void mouseEntered( MouseEvent mickey );
    public void mouseExited( MouseEvent mickey );
    public void mousePressed( MouseEvent mickey );
    public void mouseReleased( MouseEvent mickey );
}
```

If you want to be able to respond to mouse clicks, you will need to implement a class that has an appropriate mouseClicked method. But the MouseMotionListener interface specifies a contract with five distinct methods. If clicks are the only kind of MouseEvent that you want to respond to, it would be rather annoying to have to implement each of the other four methods just to be able to write the one (mouseClicked) that we need. Our class definition might say

```java
public class MouseHandler implements MouseListener {

    public void mouseClicked( MouseEvent mickey ) {
        // Interesting code goes here...
    }
    public void mouseEntered( MouseEvent mickey ) {}
    public void mouseExited( MouseEvent mickey ) {}
    public void mousePressed( MouseEvent mickey ) {}
    public void mouseReleased( MouseEvent mickey ) {}
}
```
Not very concise or beautiful, but necessary if we are to implement the interface directly. After all, an interface is a contract and implementing the interface means fulfilling the whole contract, not just a part of it.

To avoid this ugliness, `java.awt.event` gives us a more concise way of saying the same thing. There is a class called `MouseListener` that implements `MouseListener`, providing all of the (non-interesting but also non-abstract) method bodies required. We can just extend `MouseListener` in our class, eliminating the need to implement all of the extra (extraneous) methods:

```java
public class MouseHandler extends MouseAdapter {
    public void mouseClicked( MouseEvent mickey ) {
        // Overrides MouseAdapter's mouseClicked method.
        // Interesting code goes here...
    }
}
```

Much nicer!

Each of the listener interfaces that declares more than one method has a corresponding adapter class. These are listed in the AWT Listeners and Adapters segment of the AWT Quick Reference appendix.

**Inner Class Niceties**

Let's return to the TextField handler class from the Galadriel example, above. There are still some improvements in functionality that we can make.

We might, for example, make our own class -- our own specialized `TextField` -- that is born with its own `FieldHandler`:

```java
public class HandledTextField extends TextField {
    public HandledTextField() {
        ActionListener nameHandler = new FieldHandler( nameText );
        nameText.addActionListener( nameHandler );
    }
}
```

Now each HandledTextField is born with its own FieldHandler. This is similar to AnimateObject's creating its own AnimatorThread, rather than expecting someone else to create the AnimatorThread on its behalf.

Using inner classes,[Footnote: See chapter 12 for details.] we can make this innovation do even more work for us. Inner classes are a relatively advanced feature of Java, and they add only to the aesthetics of this program, not to its functionality. They do provide a little bit more protection for code from unanticipated use, a feature that we can exploit. After all, a FieldHandler as we have defined it is not really of much general interest. We can embed the definition of that class inside the HandledTextField class definition, hiding it from the rest of the world and simultaneously taking advantage of inner class's privileged access to their containing instance's state.
Using inner classes, we can write:

```java
public class HandledTextField extends TextField {

    public HandledTextField() {
        ActionListener nameHandler = new FieldHandler();
        nameText.addActionListener( nameHandler );
    }

    private class FieldHandler implements ActionListener {

        public void actionPerformed( ActionEvent ae ) {
            Console.println( "Hello, "
                + HandledTextField.this.getText() );
        }
    }
}
```

Since `FieldHandler` is defined inside `HandledTextField`, it has access to its containing instance directly (through `HandledTextField.this`), and we can eliminate the constructor argument (and the constructor itself!) for `FieldHandler`. Pretty neat, huh?

Chapter Summary

- EventListeners are interfaces promising particular sets of event handler methods. There are Listeners for groups of related AWT event types, such as mouse motion events, in the package java.awt.event.
- That package also includes adapter classes to make implementing these interfaces easier.
- Listeners are connected to AWT components using a component's `addEventClassListener()` (registration) method.
- `java.awt.AWTEvent` and its subclasses are data repositories that record relevant information about individual (GUI) events.
- Each event handler method takes one of these Event objects as an argument, in much the same way that `paint()` requires a `Graphics`. Like `paint()`, the event handlers of an EventListener are called by the system, not by your code.
- Inner classes provide a nice way of packaging the definitions of subsidiary classes (such as EventListeners) inside other class definitions.

Exercises

1. Define a class that implements `java.awt.event.MouseAdapter` and extends the `mouseClicked(MouseEvent)` method by printing the coordinates of the point on which the mouse had clicked. You may also want to make use of the class `java.awt.event.MouseAdapter`. (Bonus: also print the components of the previous mouse click.)
2. Now define a class that extends `java.awt.Canvas` and sends its mouse events to your `MouseListener`.

3. Define a class that implements `java.awt.event.WindowListener` and extends the `windowClosing()` method by printing "Nah, nah, you can't kill me!" (Alternately, you can do the potentially more useful thing and (1) call the object's `dispose()` method and (2) call `System.exit(0)`.) What class do you think would be useful when implementing `WindowListener`?

4. Define a class that extends `java.awt.Canvas` and looks like a (black and white) Japanese flag, i.e., it has a circle at (100,100). Make the circle change color when the mouse is over your `Canvas`. (Hint: mouse enter, mouse leave.)

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's [Rethinking CS101](http://www.cs101.org/ipij/) Project at the [Computers and Cognition Laboratory](http://www.cs101.org/ipij/) of the [Franklin W. Olin College of Engineering](http://www.cs101.org/ipij/) and formerly at the [MIT AI Lab](http://www.cs101.org/ipij/) and the [Department of Electrical Engineering and Computer Science](http://www.cs101.org/ipij/) at the [Massachusetts Institute of Technology](http://www.cs101.org/ipij/).

Questions or comments:
<webmaster@cs101.org>
Client-server interaction patterns

Chapter Overview

- Who is responsible for getting something from one entity to another?
- What tradeoffs are involved in this decision?

This chapter concerns the ways in which responsibility for (information) transfer can be allocated between the provider and the recipient and the implications of these design decisions. When the service provider takes responsibility for the transfer, it maintains control of its own workload but may overwhelm the recipient. When the recipient initiates the request, the dual situation is in effect. The participants in this relationship are called servers and clients, and client/server architectures are common in modern software design.

What is a client-server interaction?

Sometimes, one (computational) entity has something that another (computational) entity needs. For example, a baker may have cookies, and you may be hungry. In this case, the entity that has the thing--the baker--is called a server and the entity that needs the thing--you--is called a client. Although these terms are often used without further explanation, you can see from this description that a client and a server are defined with respect to some (computational) need, or service (like a cookie).

In the computational world, a server is often something that provides a particular service to other computers connected by a network. For example, it is common for an organization to have a lot of disks on which its members' information is stored, and to have a single machine responsible for providing access to this storage space. This machine is called a disk server or a file server. Another machine in the same organization might control the public html access for that organization's world-wide web pages. That machine would be the organization's web server. Yet another machine might be in charge of electronic mail for the organization: the mail server.

In each of these cases, the service is described by what is provided. But it is also important to characterize how the service is provided, in what form, when, by whom, and to whom, whether it is provided once or repeatedly, whether it is provided to one client at a time or to many clients simultaneously, and who is responsible for initiating the transaction. For example, an important message may be transmitted by certified mail, or it may be communicated by announcement over a public address system. These two services may communicate the same information, but they do so in dramatically different ways.

Postal Services: An Example
In this section, we'll look more closely at a particular real-world, non-computational service: the postal system. In doing so, we will see that most of the major properties of client-server interactions are present in familiar transactions.

The main service provided by the postal system is the transmission of physical letters and packages. In this sense, it is perhaps the original mail server. Its clients are the people who send and receive mail through the system. The post office (or, more generally, the postal system) is the server. It may be obvious that the recipient of mail is a client of the postal system: A recipient gets mail delivery from the postal system. Perhaps less intuitively, the sender of the mail is also a client (of another service of) the postal system: the sender relies on the post office to provide the service of transmitting the letter. We will return to this point later.

Figure #. The post office provides parcel transmission and parcel delivery services. Its clients are senders and recipients, respectively. The post office is a server providing the service of transmission of packages.

This example is actually quite rich, illustrating several points about service providers.

1. **A server can provide a variety of services**

We have already seen two interrelated services that the postal system provides: mail transmission (to mail senders) and mail delivery (to mail recipients).

Actually, each of these is itself an abstraction of several different services. For example, letter transmission is a somewhat different service than parcel transmission. The post office charges different rates depending on the weight and size of the item to be delivered. Similarly, the postal system provides multiple qualities of service for similar items. A letter can be sent air mail or surface mail, overnight or second day or standard delivery. A parcel can be sent first class or book rate. Each of these services is a specialized form of mail transmission, with different costs, time behaviors, and guarantees. All can be described as the same general mail transmission service, but each has slightly different behavior.

Some service specializations don't fit neatly under the same abstraction. For example, a letter can be sent certified; a return receipt can be requested. Requesting a return receipt even changes the contract between the client -- the mail sender -- and the postal system so that their interaction pattern is different. In traditional mail transmission, the client gives the item to the post office and the interaction ends. (Of course, the post office is still obliged to carry out its end of the deal, delivery.) When a return receipt is requested, the transaction does not end here. Instead, delivery involves the post office's obtaining a signature from the recipient. This signature needs to be transmitted back to the sender; only then is the original transmission service complete. This amounts to the postal equivalent of a callback. (See the chapter on Intelligent Objects.)

The same server that provides these transmission and delivery services -- the post office -- also provides a number of other services, some of which may not even seem related. For example, the United States Postal Service sells stamps and money orders. By special arrangement (i.e., the rental of a post office box) it will hold your mail for you. Some post offices will even provide you with a passport. This last service is one provided by the post office acting on behalf of the Passport Agency.
When we talk about a server, then, it is important to distinguish what service that server is providing. In general, it is impossible to talk about a client or server without (at least implicitly) referring to the service provided.

2. You can have more than one provider of a given service

Among the services provided by the post office is the selling of money orders. But if you want to transmit money through the mail, you can also do so using a check. The check is a service provided by a bank, not by the postal system. So, for sending money through the mail, you can turn to a variety of service providers. Each will have its own set of properties: cheap or expensive, secure or less so, available on demand or only during certain hours, etc.

Even considering only the delivery service that is the postal system's "core" service, there are still alternative providers. In the United States, package and letter delivery is provided by United Parcel Service, Federal Express, DHL, and many other vendors. Each of these service providers -- servers -- has a different performance profile. For example, some of these parcel delivery servers are faster, provide "better" service, include a variety of guarantees, cost more or less, etc. At different times, you may wish to select a particular server because its properties best match your needs. But even when multiple servers provide the same (or similar) services, any one service instance -- such as sending one particular parcel -- is likely to go through only one provider of a particular service. For example, when you mail your mother's birthday present, you will pick one carrier to deliver it.

3. Services can be layered

We have seen that the post office provides both transmission and delivery services. These two services together can be used as the basis for other service models.
to ship the shirt. In this case, shirt-procurement is **layered** on top of parcel delivery, i.e, relies on parcel delivery to accomplish the transaction.

Real services often work this way. In fact, network services -- the way that one computer communicates with another -- involve many layers of services. When we look more closely at network services in chapter 21, we will examine only the highest levels of these services. We will use network transmission to build still more sophisticated services -- such as a web server or a chat program -- in exactly the same way that the mail order company relies on the postal service to deliver its shirt.

4. **Roles are relative to a service**

We have seen how a single server can provide many different services (as the post office does) and that a single service may be provided by many different servers (like the various parcel delivery servers). We have also seen how layering makes it possible for one service to be built out of others. Each of these observations provides further illustration that the role "server" (or "client") is not an absolute one, but is meaningful only relative to a particular service.

We can't, for example, properly say that the post office is a server. We have to specify what service the post office is providing to whom. Of course, we sometimes skip this information when we think that it is obvious from context. But properly, every server is the server of a particular service interaction; every client is a client with respect to a service interaction.

This is particularly important when we're talking about sophisticated interactions in which a single entity can be simultaneously a client and a server. (Not of the same service interaction, of course.) So, for example:

- The mail order company is simultaneously the server of "shirt purchasing" (I'm the client) and the client of the post office's delivery service.
- In most standard retail transactions, the retailer is simultaneously the client of the wholesaler (who sells the retailer the goods) and the server to the general public (like me).
- Many interactions are two-way, like barter. For example, one farmer may supply eggs to a second; the second may provide the first with milk. Each farmer is a service provider (server) as well as a service consumer (client).

Note that in any relationship, an entity can either be a client or a server of a particular service instance, but not both.

In computational systems, we typically reserve the term server for ongoing service providers, i.e., persistent entities that can be repeatedly called upon to provide services. That is, servers are full-blown computational entities, not simply program segments.

**Implementing client-server interactions**

As we have seen, a client-server interaction is one in which the server has something at the beginning and the client has it at the end. This "thing" might be quite abstract -- permission to access some data, for example, or the property of being subscribed to a mailing list -- but the idea is that the client wants it and the server can provide it.
In this section we will focus on the question of who initiates the service and the implications of this decision for client-server interactions. There are many different services provided by many different servers, and many different mechanisms to support these services. These include simple procedure call, the use of channels to transmit requests, even aspects of event-driven programming. The issue of who takes responsibility for service initiation exists no matter what service mechanism is used to implement the interaction, and the tradeoffs described here apply to each of these implementations as well.

**Client pull**

If a client needs something (or some service) from a server, perhaps the easiest way for this transfer to happen is for the client to go and get -- or pull -- the thing from the server. We do this all the time. For example, this is what happens when we go to the grocery store.

- The client requests the service as it is needed.
- The server handles each request as it comes in.

The following icon represents a client pull client: In this icon, information flows from right to left. The client (the circle) initiates the transfer of information, requesting it from the server and retrieving it. Here is a client pulling from a (passive) server:

Getter methods are very simple versions of client pull. In a getter method, one entity asks another for something; the method return completes the pull request. When direct method invocation is not available -- as when communicating over a channel, or network -- a pull request usually consists of two separate messages: one from the client, requesting the service, and one from the server, completing it.

---

**Locating the Server**

In order for a client pull interaction to work, the client must know where to find the server. This can be accomplished either by telling the client about the server when the client is created or by providing a standard place to look. For example, I might first ask the phonebook where the grocery store is, then get what I need from the grocery store. This interaction involves two separate client pulls and is depicted in the next figure. Structurally, this is the way that computers locate each other on the internet.
Interactive Programming In Java

Client Pull Tradeoffs

There are many advantages to client pull. The server doesn't have to do anything unless a request is pending. The client gets only what it needs, when it needs it. The client exercises control over the interaction, so the interaction (theoretically) happens when and where the client is best able to make use of the service.

On the other hand, there are disadvantages, too.

- The burden is on the client. (You have to go to the grocery store. Sometimes, you may miss out on a special because you get to the store at the wrong time.)
- The server may be deluged with requests and unable to keep up. (The grocery store may run out of something.)
- The network may be full of requests, since each client is sending these requests separately. (The check-out line may be long.)
- In general, more energy will be expended. For example, there's likely to be a lot more (network) traffic. (Each client arrives in his/her own car. Sometimes there are traffic jams in the parking lot.)
- The server's load may be patchy and unpredictable -- too busy one moment, unused the next -- making inefficient use of the machine and its resources. (Adequate inventory and staffing levels may be hard to identify, making the grocery store an inefficient business.)

Client pull works well when client requests are highly variable but overall not too great a load on the server. It allows each client to do its grocery shopping precisely when it needs to. When it doesn't work, though, the grocery store can be quite a mess!

Server push

An alternate architecture that addresses some of these problems is for the server to take initiative. In this case, it can simply deliver -- or push -- the information to the client when it is ready. This is sort-of like the fruit-of-the-month club. Every month, the fruit-of-the-month club delivers a box of fresh, ripe fruit to your house.

\[
\text{fruit-of-the-month club} \quad \text{me} \quad \text{put}
\]

Figure #. The fruit-of-the-month club delivers a box of fruit to me each month, without my having to do anything. This is server push.
In direct method invocation, setter methods are server pushes. Although these methods technically complete with a return, no value is returned; in a put, only one-way communication is necessary. In channel- or network-based interactions, a push is often implemented as a single communication.

We can represent a server push server with the icon and a server push interaction with a (passive) client with the icon . Again, information flows from left to right. In this case, however, the entity with the information initiates the service.

Registering with the Server

Before the fruit-of-the-month club can provide me with regular deliveries, however, I may need to register my interest with the club. This is often done as a one-shot communication that precedes the regular (server push) delivery. Many subscription services -- like the fruit-of-the-month club, magazine subscriptions, or other periodic deliveries -- require a registration before the recurrent server push can begin.

Server Push Tradeoffs

There are numerous advantages to server push approaches:

- The server gets to control who gets what when. This means that it can manage its resources and keep its load even (or at least predictable): clients with names A-G this week, H-M next; oranges this month because they're in good supply.
- If the server is supplying multiple clients with the same -- or similar -- information, there may be economies of scale. For example, the server may only have to package up the information once to send to multiple clients.
- The client doesn't have to do anything to make this happen. The service just shows up whenever the server thinks it appropriate.

But this model doesn't always work perfectly, either. Why not? Let's consider what can go wrong:

- Deliveries might come at a bad time. Imagine that a whole month's shipment of fruit arrives the day after you've left town for a week. By the time you get back, the fruit will have spoiled. This happens in computational systems, for example, when the server goes too fast for the client, and the client has to spend all of its time handling the server's shipments. (Sometimes, this happens even if all that the client is doing is throwing the server's information away: trash can pile up and become overwhelming. In other cases, the client can't throw the information away, because the next delivery depends on the previous one in some crucial way.)
- Even though the server maintains control, it can still get too busy and deliveries can get back-logged. Some clients might need more frequent attention. Such a client might not get what it needs often enough, or even in time. The fruit-of-the-month club might be reliable, but not all computational servers are.
- Sometimes, the server doesn't know that (or when ) the client wants or needs something.

Both the pros and cons of this approach can be summed up by the following:

- The client has very little control over what it gets when.
The server has lots of control, but also has to do all of the work.

One popular way of doing animation in the early days of the web involved having the web server regularly push the next image in the animation sequence. This often swamped clients -- web browsers and the machines running them -- making it difficult for their users to do anything at all and giving server push a(n undeservedly) bad name.

The Nature of Duals

Server push and client pull are opposites of a special sort. The positive aspects of one are the negative aspects of the other. In general, they are like mirror images. Pairs of opposites like this are called duals, and they have some special properties. For example, you can generally take almost any statement expressed in terms of these dual operations (and their associated dual terms, such as client and server) and replace each operation with its dual without changing the truth or falsity of the statement:

- **Client pull** gives the **client** a lot of flexibility, but the **server** doesn't have much control over its workload.

**dual statement:**

- **Server push** gives the **server** a lot of flexibility, but the **client** doesn't have much control over its workload.

Of course, it's not quite this simple -- it's not too hard to find statements that you can't turn around this way -- but client pull and server push are duals, which mean's that there's a fundamental symmetry in the ways that they work.

Pushing and Pulling Together

It is possible to build a system that uses multiple -- chained -- server pushes to produce its result. In this case, the client of one push becomes the server of another push:

For example, a farmer may push produce to the wholesaler -- taking it to market when it is ready -- while the wholesaler in turn may deliver it to the retailer when it becomes available.

![Diagram](http://www.cs101.org/ipijpush.html)

**Figure #.** You can chain together server pushes: the farmer sells to the wholesaler, who sells to the retailer, who sells to the customer.
Similarly, a chain of client pulls -- requests for services -- allows one client to pull from a server that may in turn request assistance from another service: Requesting a book on interlibrary loan follows this process.

\[ \text{me} \quad \text{branch library} \quad \text{main library} \quad \text{county library} \]

Figure #. You can also chain together client pulls: I reserve a book at my branch library, which asks the main library, which sends out the request to the entire county system, which finally finds the book.

Note, however that each of these pictures involve the chaining of similar service models. It is less simple to put a client pull client together with a server push server (or a server push client with a client pull server). (Iconically, there's no way to connect \( \text{to} \) or \( \text{to} \).) To make these transactions possible, we need to introduce additional machinery. You cannot connect server pushes to client pulls (or client pull to server pushes -- there's that dual thing again!) without putting something different in the middle.

**Passive repository**

A passive repository is essentially just a the server side of a client pull combined with the client side of a server push. In other words, it's the passive recipient of information provided to it, and the passive provider of information when requested. It corresponds to the "drop box" where a spy might leave information for his spy master. The server -- or spy -- can drop off the information any time it wants. The client -- or spy master -- can come by and pick up the information whenever it is convenient. Iconically, this is just a \( \). It has the important property that it can be used to connect a server push (\( \) with a client pull (\( \)), making a functioning system: \( \).
What happens if the server pushes a new value before the client pulls the previous one? One possibility is that the passive repository actually contains a queue, i.e., keeps track of each of the items it's been given and provides them to the client pull client, either one at a time or all at once when the client requests them. (The first case, in which the repository provides the values one at a time, works much like an event queue -- see chapter 15 -- or channel -- see chapter 21.)

Alternately, the repository can keep track only of the last thing deposited. In this case, the repository will work well as long as the server updates the repository often enough that the client is assured of reading a relevant value. If the client doesn't check the repository often enough, though, the client runs the risk of missing some values.

A passive repository can be implemented using a single piece of shared data. For example, the server push may use a setter method, while the client pull uses a corresponding getter method. The data may be simple -- a single value or object -- or complex, capable of holding many values, like a queue.

Of course, there are both benefits and disadvantages to the use of a passive repository. Advantages include the flexibility to allow the active components to act whenever it may be convenient for them. But problems may arise:

- If the server pushes too much more often than the client pulls -- the spy drops off a lot of documents, but the spymaster doesn't claim them -- the repository can fill up.
- If server doesn't push often enough for the client -- or if the client pulls too frequently for the server -- the client may receive out-of-date (stale) information or no information at all. (The spymaster can't pick up documents faster than the spy delivers them.)

The use of a passive repository works best when the client and server need to operate relatively independently, but run at about the same rate.

**Active constraint**

An active constraint is the dual of a passive repository. If a passive repository couples a server push server and a client pull client, then an active constraint couples a server push client and a client pull server. Each of these is a passive component -- the active component both pulls (from the passive server) and pushes (to the passive client):
Imagine a diner in a fancy restaurant. As soon as the diner puts down his fork, the fork disappears from the table, reappearing at the dishwasher. How does this happen? The active constraint -- in this case, the busboy -- gets (pulls) the fork from the diner and gives (pushes) it to the dishwasher.

![Diagram](image)

**Figure #.** An active constraint pulls information from the passive server and supplies it to the passive client.

This process requires no initiation of action on the part of either the client or the server. Instead, each of them goes about their business, responding only when the active constraint explicitly asks for something (or provides something). The intermediate entity -- the active constraint -- does all of the work to make this transfer happen.

Like server push and client pull, passive repository and active constraint are duals. A server push server can be connected to a client pull client by a passive repository. A client pull server (i.e., a passive server) can be connected to a server push (passive) client by an active constraint. In fact, a passive repository is the client side of a server push attached to the server side of a client pull. By duals, an active constraint should be the server side of a server push and the client side of client pull -- and it is!

**Chapter Summary**

- A service is something provided by one entity to another. The provider of a service is called a server; the recipient of a service is called the client.
- An entity is a server or a client *with respect to a particular service*. Services can be layered or chained.
- Client pull describes the situation in which a client initiates a service request. This is like shopping at a grocery store, with all the attendant advantages and disadvantages.
- Server push describes the situation in which a server initiates the request. This is like subscribing to the fruit-of-the-month club.
- Client pull and server push are an example of duals.
- A server-push server and a client-pull client can be connected using a passive repository.
- A client-pull server and a server-push client can be connected using an active constraint.

**Exercises**
Real-world interactions are often complicated mixes of clients and servers. One way to tell who is (apparently) the server is that the client often pays for a server's services. Consider each of the following interactions and describe who is the client and who the server:

1. I buy a computer from a store.
2. I rent a car.
3. I rent a computer from a store.
4. I rent a computer from a service that (a) doesn't charge me but (b) requires that I read ads before using the computer.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Synchronization

Chapter Overview

- What happens when two entities want to use the same thing at the same time?

Synchronization is an issue that arises when multiple animacies share state. In Java, this means that there are multiple Threads directly or indirectly accessing some field of an object. These Threads may either be explicitly created or automatically generated as, e.g., the user interface Thread in java.awt.

When an object accesses state, it does so either to obtain or to set a value. If the access does not change the value, we call it a **read**. An access that changes the value of some state is called a **write**. If more than one thread can access a state, we call it **shared state**. Shared state can lead to problems if there are multiple accesses of the state at the same time and at least one of those accesses is a write. To avoid these problems, we can prevent sharing, we can prevent writing, or we can use specialized mechanisms or protocols to minimize conflict.

An Example of Conflict

When I was in high school, we took a class trip to Washington D.C. While we were there, we had a class photograph taken on the Capitol steps. Since there were a lot of us, they used a panning camera. The photographer started off pointing to the left, then scanned across the class until he got to the rightmost edge. The entire process took a minute or two.

The interesting part came when the photograph was developed. One of my classmates appeared in the upper left-hand corner of the picture. He also appeared in the upper right-hand corner! Here's how he did it: He started in the left edge of the group. As soon as the camera had moved past him -- during the minute or so that the photographer was scanning the group -- this classmate ran from one end of the group to the other. By the time that the photographer got to the right edge of the group, he had reached that side and was standing among the students there.

This is a synchronization failure. The problem is that the scanning of the group took time. Between the time that the scan started and the time that the scan completed, the student was able to change his position, so that the camera recorded him in both places. This is called a read-write conflict: the camera "read" a value that was incorrect. (My classmate's new position had already been recorded. A similar problem would arise if he'd run the other way -- then neither position would be recorded.)

A second example arises when two writes conflict. Say that our bank account contains $1000. We go to deposit $100. The ATM (automated teller machine) reads our current balance -- $1000 -- and goes off to calculate the new balance. At the same time, the bank's computer goes to give us our periodic 1% interest. It, too, reads our current balance ($1000) and sets out to compute our new balance from this.
In the meantime, the ATM finishes computing our new balance and stores it in the central accounting ledgers -- $1100. Finally, the banks' computer calculates our balance after interest -- 101% of $1000 is $1010 -- and writes that value to the central ledger. Unfortunately, the result is that after deposit and interest, we have a balance of $1010, not $1110.

These failures occur because there are two things going on at once -- a student running and a camera photographing, or two processes computing new balances -- that interact in inappropriate ways.

**Synchronization**

Synchronization is required when *two or more threads of control* (animacies) *access the same* (piece of) *state and that state changes*. Synchronization prevents one animacy from reading the state while the other might be changing it. In Java, it also ensures that each read is of an up-to-date version of the state.

Synchronization is only necessary when there can be a write to shared state.

**Java synchronized**

The primary means of ensuring mutual exclusion in Java is through *synchronized methods*.

**methods**

In Java, a method may be declared *synchronized*. In each object at most one *synchronized* method can run at any one time. We say that a *synchronized* method *obtains a lock* on its containing object before it can execute. Since there is only one lock for each object, this prevents any other *synchronized* method from running until this method completes: no other method can obtain a lock on the object until this method releases its lock. This one-animacy-running-at-a-time property is called *mutual exclusion*.

Locking an object only prevents access to other methods or code blocks that also require a lock on the same object. Locking an object does not prevent other (non-synchronized) methods of the object from running, nor does it prevent other use of the object.

**(blocks)**

Java has a second form of *synchronized* execution. A special *synchronized* statement type can be used to provide mutual exclusion on its body. Unlike *synchronized* methods, the *synchronized* statement (sometimes called a *synchronized* block) must explicitly specify the object it locks. The syntax of this statement is:

```java
synchronized ( objectReference ) {
    statements
}
```

Here, *objectReference* is some expression whose value is of an object type; the locked object is the expression's value. The *objectReference* expression should be one whose value does not change; otherwise, careless coding can easily lead to a failure of mutual exclusion.
What synchronization buys you

Consider the class photograph described above. If the photographer had had synchronization, he would have been able to tell us not to move -- and would have been able to enforce it -- until after the photograph was done. My classmate never could have appeared in the single picture twice. (Well, at least not without digital enhancement.)

The bank example is similar. Real ATMs lock the account during the transaction, so that the interest figuring process couldn't read the balance until the ATM was done. In this case, the two computations would not overlap and the correct final balance would be reached.

Safety Rules

Sometimes, a set of data is interdependent. For example, we might have two fields corresponding to a street address and a zip code. Changing an address might involve changing both of these fields. If the zip code is changed without a corresponding change to the street address, the data may be inconsistent or incoherent. Such a set of operations, which must be done as a unit -- i.e., either all of the operations are executed or none are -- in order to ensure consistency of the data, is called a transaction. The property of "doing all or none" is called atomicity. A system in which all transactions are atomic is transaction-safe.

The following rule suffices to ensure that your system is transaction-safe:

All (potentially changeable) shared data is accessed only through the synchronized methods of a single object; no interdependent piece can be accessed independently.

Note that this means that shared data cannot be returned by these methods for access by other methods. If shared data is to be returned, a (non-shared) copy must be made. Further, if interdependent values are to be returned (i.e., a portion of the shared data is to be used by other methods), all of the relevant values must be returned in a single transaction.

For example, the address and zip code of the previous example should not be returned by two separate method calls if they are to be assumed consistent.

```java
public class AddressData {

    private String streetAddress;
    private String zipCode;
    public AddressData( String streetAddress, String zipCode) {
        this.setAddress( streetAddress, zipCode );
        ....
    }

    public synchronized void setAddress( String streetAddress, String zipCode) {
        // validity checks
        ....
        // set fields
        ....
    }

    public synchronized String getStreetAddress() { // problematic!
        return this.streetAddress;
    }
}
```
If this class definition were used, e.g. for

    printMailingLabel( address.getStreetAddress(),
                      address.getZipCode() );

it would in principle be possible to get an inconsistent address. For example, between the calls to
address.getStreetAddress() and address.getZipCode(), it is possible that a call to
address.setAddress could occur. In this case, getStreetAddress would return the old street
address, while getZipCode() would return the new zip code.

Instead, getStreetAddress() and getZipCode() should be replaced by a single
synchronized method which returns a copy of the fields of the AddressData object:

    public synchronized SimpleAddressData getAddress() {  
      return new SimpleAddressData( this.streetAddress,
                                    this.zipCode );
    }

The SimpleAddressData class can contain just public streetAddress and zipCode fields,
without accessors. It is being used solely to return the two objects at the same time.

Deadlock

If you are not careful, it is not too difficult to get into a situation where multiple active objects each prevent
the other from running.

Consider two objects which each need to control both the chalk and the eraser in order to write on the
blackboard. The first uses the following algorithm:

1. Wait until the chalk is available, then pick it up.
2. Wait until the eraser is available, then pick it up.
3. Write (and erase).
4. Release the eraser.
5. Release the chalk.

The second uses the following algorithm:

1. Wait until the eraser is available, then pick it up.
2. Wait until the chalk is available, then pick it up.
3. Write (and erase).
4. Release the chalk.
5. Release the eraser.

If the two processes time things just right, it could be the case that they each complete their first steps before reaching their second. Now, the first process will be stuck waiting for the eraser (which the second process has), while the second will be stuck waiting for the chalk (which the first has). This situation — in which neither process can do anything, and both are stuck waiting — is called **deadlock**. (The processes in this case are effectively **dead**.)

There is an analogous situation that arises when both processes put down the objects they have and pick up the other object (repeatedly). In this situation, although both processes are still alive, neither is making any progress. This is called **livelock**.

The desirable property of a system that doesn't reach deadlock is **liveness**. In general, there is a tradeoff between safety and liveness, and a significant part of programming concurrent applications is designing to simultaneously maximize both.

**Obscure details**

This section is not for the faint of heart. While it is true, it is not pretty. Feel free to skip it.

**Synchronization and local copies of state**

In Java, each **Thread** may keep its own copy of shared state. This means that one copy may be inconsistent with another. Using **synchronized** forces a **Thread** to refresh all of its shared state, ensuring that it does not have a stale copy. Thus, even if timing constraints guarantee that only one **Thread** can access the state at a time, it may still be necessary to use **synchronized**. However, in this case the identity of the locked object is irrelevant; any **synchronized** method or block will do. (An alternate solution to this problem, though not to synchronization in general, is the **volatile** keyword on fields.)

**Synchronized blocks and lock object references**

It is the value returned by the expression (at the time that the lock is obtained), and not the expression itself, that is locked. For example, given the following class definition

```java
class SynchronizationFailure {
    Object foo = new Object();
    void failToSynchronize() {
        synchronized (foo) {
            foo = new Object();
            other statements
        }
    }
}
```

http://www.cs101.org/ipij/synchronization.html
the synchronized block does not provide proper mutual exclusion. Consider a particular SynchronizationFailure instance, popularObject. If Jack and Jill both call popularObject.failToSynchronize() with appropriate timing, here is what could happen:

1. Jack's call to failToSynchronize obtains a lock on popularObject.foo's current value, say object 1.
2. When the line foo = new Object(); is executed, popularObject.foo is assigned a new value, object 2.
3. Jack's call continues to execute other statements.
4. In the meantime, Jill calls popularObject.failToSynchronize(). When Jill's call reaches the synchronized block, it attempts to obtain a lock on popularObject.foo's current value, object 2. Although Jack's call is still inside the synchronized block, Jill's call is able to enter because it attempts to lock a different object from Jack's call.

Note that this failure can arise any time the value of the objectReference expression can change, even when it does not change inside the synchronized block. To avoid such failures, the synchronization expression (i.e., the objectReference on which the lock is obtained) should generally be an expression whose value does not change.

Chapter Summary

- Conflict can arise when multiple animacies access mutable state. For example, an entity may read an impossible value.
- This kind of conflict can be prevented by limiting state access to single animacy or by making all shared state immutable.
- When shared mutable state is desired, access can be controlled through Java's synchronization mechanisms.
  - Each object has its own "lock".
  - At most one animacy can hold this lock at any time.
  - A method may be declared synchronized. An animacy cannot execute a synchronized method until it holds the lock of the object to which the method belongs.
  - A block may be declared synchronized on a particular object. An animacy cannot execute a synchronized method until it holds the lock of the specified object.
- A transaction is a group of operations which must either be completely executed or not executed at all. Partial execution is not legal. A system is transaction-safe if all of its transactions are executed atomically, i.e., partial execution is not possible.
- In general, increasing (transaction-)safety means decreasing liveness, a program's ability to run towards completion.
  - Transactions that interfere with one another so that all execution stops are called deadlocked.
  - Sometimes transactions interfere so that execution continues, but no progress can be made towards completion. This is called livelock.
This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Network Programming

Chapter Overview

- How do entities on one computer communicate with entities on another computer?

Many modern applications involve multiple computers. This chapter introduces Java's primary mechanisms for making such interaction possible: communication channels, or streams, over which information can be transmitted. Transmission over these channels is often mediated by a Lector -- one who reads -- and/or a Scribe -- one who writes -- on behalf of an entity. Communication may occur across a network, between co-located entities, or with persistent storage resources such as a File on disk. The stream abstraction gives these diverse kinds of communication a uniform interface.

In this chapter, we present a series of Lector/Scribes, initially relying only on local resources, ultimately establishing and controlling a network connection. We conclude with a discussion of a multi-threaded server and a brief look at the role of a server in a network architecture.

A Readable Writeable Channel

Two entities often need to communicate with one another. We have seen how this can be accomplished using direct method invocation. But that kind of communication requires that the calling object know the identity of the method's owner. This is the equivalent of having a face-to-face conversation: You must know to whom you are speaking. In this chapter, we explore a more abstract communication mechanism that uses intermediate objects -- called streams -- to allow one entity to communicate with another indirectly. Stream communication is more like talking on the telephone. The same device can be used to interact with many different people -- or entities -- without requiring direct contact. Streams similarly provide a uniform interface that can be used to communicate indirectly with a wide variety of objects or resources.

A stream -- or, more properly, a stream of values -- is an abstractly defined resource containing a sequence of values that can be processed, one by one. Streams come in two flavors: input streams, which support the reading of values, and output streams, which support their writing. In other words, an input stream is a stream from which these values can be read, one by one, in order; an output stream is such a resource to which values can be written one by one. In this chapter, we will concentrate on stream-like objects and how they are used.

Tin Can Telephones

One way to think about a stream is to consider the tin can telephones many of us played with as children:
Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.

This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end. The end into which Tweedledum is speaking is like an output stream. The end to which Tweedledee is listening corresponds to an input stream. Anything Tweedledum writes to the output stream can be heard by Tweedledee (reading) the input stream. [Footnote: Note, though, that the communications medium itself -- the tin can telephone -- isn't an input stream or an output stream. The medium has two ends, one of which is an input stream, and the other of which is an output stream. In the case of the tin can telephone, these roles might be marked on the two cans: "Listen here" on one can, "Speak here" on the other. In the case of streams, there's less room for confusion. A stream is either an input stream or an output stream, and the two are not interchangeable.]

One nice property of this system is that the tin can telephone doesn't rely on face to face contact between the conversationalists. Using a stream, a Java entity can talk to all different kinds of things without needing to know much about those things. Communication relies on properties of the stream, rather than on properties of the thing at the other end. It also means that the communicators don't have to be directly in contact, as long as each holds one end of the tin can telephone, or stream. For example, Tweedledum can use the same kind of device to talk to the Jabberwock, even though Tweedledum knows far better than to approach the Jabberwock face to face.

The tin can telephone story so far works well when Tweedledum wants to communicate something to Tweedledee. But what happens when Tweedledee has something to say as well? We can accomplish this using the same approach. But in order to have simultaneous two-way communication, it is useful to have two tin can telephones. Then Tweedledum listens to one and talks into the other; Tweedledee listens to the one Tweedledum talks into, and talks into the one Tweedledum is listening to.

In the same way, Java streams often come in pairs. One -- the -dum to -dee route -- is an output stream for -dum and an input stream for -dee. The other -- -dee to -dum -- is output stream for -dee and an input stream for -dum. If you are standing at one end -- like Tweedledee -- you are holding one input stream (from which you can read) and one output stream (to which you can write).

**Streams**

In this book so far, we have talked about cs101.io.Console, a particular concrete resource that behaves like one end of the two-stream configuration. Console behaves like an input stream -- through its `readln()` method -- and also like a separate output stream -- through its `println(String)` method. The Channels used in interlude 1 are another example of stream-like behavior, though the methods provided for reading and writing are somewhat different from Console's. Like those channels, many kinds of Java streams are used to connect two entities.
Streams can also be used to interact with things that may be more outside of your program, such as a File -- information stored on your disk -- or a network connection. Each different kind of stream-like resource has a slightly different set of (read and write) methods, and each has a very different kind of behavior at the "other" end of the resource. Writing to the Console makes the information appear on your Java console, i.e., on your screen. Writing to a file stores the information away on your disk for later retrieval. Writing to a network connection sends the information to another computer. But from the read/write end -- from Tweedledee's perspective -- the stream makes these resources seem fundamentally similar.

A stream is a general way to think about each of these connections. Individual connections implement different methods to actually make stream communication possible. No matter the differences among them, these methods are generally used in stereotypical ways. The ways that reading -- extracting information from an input stream -- and writing -- depositing information in an output stream -- are used within a program is the topic of the first part of this chapter. Even the general properties of read and write operations are shared by most of these stream-like connections.

In the first part of this chapter, we will simply use the notation `inStream.read()` and `outStream.write(message)` whenever we are accessing a read or write method. All of the code in that portion of the chapter will work if the text `inStream.read()` is replaced with the text `cs101.io.Console.readln()` and the text `outStream.write(message)` is replaced with the text `cs101.io.Console.println(String)`. We use the more generic notation to indicate that other substitutions are equally possible. In the latter parts of this chapter, we will introduce other kinds of streams and talk about other code that might be used to replace `inStream.read()` and `outStream.write(Object)`.

**Using A Channel**

This section develops a very general set of classes capable of reading from and writing to a general read/write resource. It is tempting to call the thing that does the reading a Reader and the thing that does the writing a Writer. However, Java has reserved those names for other classes, described below. Instead, we will call the thing that does the reading a Lector (meaning "one who reads") and the thing that does the writing a Scribe ("one who writes"). We will define the interfaces for these classes first.

Although the details will vary from application to application and will depend in some part on the kind of resource from which you are reading or to which you are writing, the general pattern of interaction with a read/write resource is similar. In this section, we will develop a fairly general class whose instances are capable of reading from and writing to resources of this sort. In the succeeding sections, we will modify that class to tailor it to the kind of read/write resource represented by a network connection.

**For Writing**

Let's say that you have a thing, say, message. You want to write message to your output stream, `outStream`. Accomplishing this is as easy as it sounds:

```java
outStream.write(message);
```
Let's look more carefully at what is going on here. Consider the write end of a channel. This is a stream that implements a `write` method, such as `Console.println(String)`. To use this `outStream.write(Object)` method, all that we need to do is invoke it with the appropriate Object.

The write method of a resource like this one is just a server push client. It accepts information when the server writes it. To use such a write method, you must build code that acts as a server push server towards the write method. It must explicitly take action. That is exactly what happens when we invoke

```java
outStream.write(message);
```

**Flushing Out the Stream**

One detail is worth noting. Remember that the output stream is one end of something that has another end (like the tin can telephone). When an output stream's write method is invoked, it causes the thing written -- the message -- to pass into the stream. It does not, however necessarily cause this thing to be available at the other end of the stream. This is like speaking into a tin can telephone with a noticeable delay between the ends. It is quite possible that the message may be "inside" the stream. It will eventually appear at the other end, but not necessarily when the writer expects it to do so.

![Pic of balls stuck inside a pipe. Caption: When you write something to an output stream, it can get stuck "inside" the stream. A call to the stream's `flush()` method will push these objects through.]

If it is really important that the information you wrote to the stream not get stuck inside, you can use a special method, called `flush()`, to push the information through. When an output stream's `flush()` method is invoked, anything that's been written gets pushed along through the stream, so that it appears at the other end. If there are multiple things that have been written, they appear at the other end one by one, in the correct order; `flush()` doesn't change anything, it just gets things moving along.

Why might something get stuck inside a stream? Imagine that you have a carton that can hold twelve eggs. You go to the henhouse and pick up an egg. (You "write" the egg to the carton.) Back in the house, the cook is waiting for eggs to make breakfast. But it is silly for you to go back to the house with just one egg if the cook can't start until s/he has enough eggs for breakfast. So you pick up another egg and write it to the carton. You keep going until you have a full carton of eggs.

Streams can work the same way. They can wait until there is a group of information to be sent, then send the whole collection at once. Just as it saves you time to collect a carton full, it can make more efficient communication to wait for a full "packet" of information.

When you invoke a `flush()` method, that causes the information to be sent, regardless of how much is waiting. In the egg collecting example, it would make you go to the house even if you'd only collected two eggs so far. Then you'd have to go back to the henhouse and collect some more. Unnecessary `flush()`es are wasteful, just as in this example.

On the other hand, a judicious `flush()` every now and then can be beneficial. What if you were determined only to return to the house when you had a full dozen eggs? But say that today the hens layed only eight eggs. You might stay in the henhouse until tomorrow rather than return with a partially full carton. In this case, a `flush()` would be just the right thing: It would get the eight eggs you had collected where they needed to go, rather than waiting for the next four eggs (that might never come).
A Scribe Example

So far, we have seen how writing to an output stream works. We can encapsulate this knowledge inside a method that takes an object and sends it out over the output stream. The interface for an object supplying this behavior might read:

```java
public interface Scribe {
    public void send(MessageType m);
}
```

[Footnote: Note that we are being deliberately cagey about the type of object that can be written (or read). This is because that depends on the specific kind of stream that you're dealing with. Nothing in this section relies on the specific kind of stream or type of message.]

An example implementation of this method (to be encapsulated in an appropriate class) might be:

```java
public void send(String m) {
    Console.println(m);
}
```

In other words, a Scribe is an object that keeps track of its output stream and, on (send) request, writes the object to be sent to the stream.

A GenericScribeImpl class implementing this interface would need an output stream. It could then simply use that stream's write method on demand. If it is important that our writing not be delayed, we might add a `flush()` invocation to the send method as well.

```java
public class GenericScribeImpl implements Scribe {
    private OutputStreamType outStream;

    public GenericScribeImpl(OutputStreamType outStream) {
        this.outStream = outStream;
    }

    public void send(MessageType m) {
        outStream.write(m);
        outStream.flush(); // (maybe)
    }
}
```

Instances of this Scribe object are suitable for use in event-driven programs. For example, whenever something happens that needs to be communicated, the Scribe's send method could be invoked. It would then write the relevant communication to its output stream.

For example, if we have a TextField and a Scribe...
TextField textField = new TextField();
Scribe scribe = new GenericScribeImpl();

we might connect them by having the Scribe write out the text in the TextField each time the return key is hit. (Recall that hitting the return key in a TextField triggers an ActionEvent).

This can be accomplished using an actionPerformed method that says:

    public void actionPerformed( ActionEvent ae )
    {
        this.scribe.send( this.textField.getText() );
    }

[Footnote: We've omitted a few details from this example. First, the actionPerformed method is embedded in a ScribeListener class whose full definition is:

    public class ScribeListener implements ActionListener
    {
        private TextField textField;
        private Scribe scribe;

        public ScribeListener( Scribe scribe, TextField textField );
        {
            this.scribe = scribe;
            this.textField = textField;
        }

        public void actionPerformed( ActionEvent ae )
        {
            this.scribe.send( this.textField.getText() );
        }
    }

An instance of this ScribeListener class is then used to connect the TextField with the appropriate Scribe.

    textField.addActionListener( new ScribeListener( scribe, textField ) );
]

For Reading

Writing to an output stream is fairly straightforward. Reading from an input stream is somewhat more complicated. To help us read from an input stream, we will define a class called Lector: "one who reads".

The innermost portion of the Lector says something parallel to the Scribe. We certainly want to invoke the stream's read method:

    inStream.read()
Immediately, we are faced with the first complication. What should the Lector do with the message read from its input stream? There are many possibilities, depending on what you want your Lector to do. For example, the Lector could just let the user know that it has read the message (through the Java console):

```
Console.println( "Lector: just read " + inStream.read().toString() )
```

This line of code reads the message from the input stream, finds its printable equivalent using `toString()`, and then prints this version to the Java console. It is one example of a thing that we might want a Lector to do over and over again. We will return to this issue and see more complex solutions below.

The second difference between reading and writing is that the Lector must be an active object. The Scribe is automatically invoked whenever an object is available to be written. But the Lector must check to see for itself whether an object is available for reading. The input stream is passive. [Footnote: So is the output stream. But the Scribe is activated by the thing that asks it to send.]

The Lector must invoke the input stream's read method by itself. This means that an instruction follower has to come from the Lector itself. Not only that, but the instruction follower of the Lector may wind up spending a lot of time waiting for something to become ready to read. When there is no such value, the read request doesn't return. The instruction follower that executed it is simply stuck waiting. This is because reading is a blocking operation.

**Reading and Blocking**

The Lector invokes the input stream's read method -- asking for the next value -- whether or not there's a value ready to be read. It is this ready-or-not condition that poses the real issue. When there is no value to return, the Lector may get stuck waiting for one.

The read operation on almost any kind of stream is called a **blocking** read. This means that it will not return until the appropriate information becomes available. For example, if you type something on the Java console, ending with the return key, Console's `readln()` method will return this String. If you invoke `Console.readln()` again, it will return the next return-key-terminated String that you type. But what if you haven't (yet) typed another return-key-terminated String? In this case, the `readln()` method will not return. The method invocation continues until an appropriate String becomes available; the Console's `readln()` method waits for a carriage return. This waiting -- for the necessary information -- is called blocking.

Because stream reading methods almost always are blocking methods, they generally need to be invoked by a dedicated instruction follower, i.e., one that can sit around and wait until the read invocation can complete. The blocking read method itself is essentially a client pull server: it provides the information on request. To interact with a blocking read method, you must write a client pull client: active code that invokes the read method on a regular basis.

The fact that the Lector might get stuck waiting for an object to become ready -- that the read might block -- means that the Lector must have its very own dedicated instruction follower whose job is to wait for the read. This instruction follower can't be expected to get much of anything else done, because it might spend a long time waiting for the read invocation to un-block. We need a dedicated Thread -- instruction-follower -- who can afford to spend its time waiting. This is like sending one person to stand in line while the others
do something. You don't want to tie everyone up standing in line, and if you only have one person, you
can't afford to block (wait); you need to hire someone to wait for you.

We can resolve this issue by dedicating an instruction-follower to the read task. This is a job for an animate
object.

A Lector Example

We are now ready to write the Lector class. The Lector, like the Scribe, keeps track of a stream. But
instances of this class, unlike those of Scribe, are animate objects, each with its own AnimatorThread. A
Lector can afford to block each time it calls its input stream's read() method, because it has a dedicated
instruction follower. If the instruction follower's invocation of read() blocks, it is not a problem because
this instruction follower is not expected to be doing anything else other than reading from the input stream.

```java
public class Lector implements Animate {
    private InputStreamType inStream;
    private AnimatorThread mover;

    public Lector( InputStreamType inStream ) {
        this.inStream = inStream;
        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void act() {
        Console.println( "Lector: just read " + this.inStream.read().toString() );
    }
}
```

This code shows how a Lector can print the read message to the Console. But this isn't always what we'll
want to do when something is read from an input stream. For example, we might want to do a dispatch on
case, depending on what the input it reads is. This might involve some giant conditional with
`inStream.read()` as the switch expression. Or we might want to pass the new message around to
everyone we know, as in the broadcast server towards the end of this chapter.

This situation should sound vaguely familiar. Something happens: the Lector reads something from the
input stream. This is an event. There are many different ways that this event could be handled. In fact, it's
not clear that the Lector should do anything itself. Maybe what the Lector should do is to delegate this
responsibility to some other object. This could be done using the simple event handling of chapter 15 or
the more complex event delegation of chapter 16.

Paralleling chapter 16, let's define an interface for this separate event handler object:
public interface LectorListener
{
    public void messageRead( MessageType m );
}

An example LectorListener class -- one whose instances simply print their message to the Java console -- might be:

public class LectorPrinter
{
    public void messageRead( MessageType message )
    {
        Console.print( "Lector:  Just read:  ");
        Console.println( message.toString() );
    }
}

Now we'll need a way for the LectorListener to register with the Lector. We will assume just one LectorListener per Lector for now, though we could certainly do otherwise (e.g., using a Vector). The modifications are underlined.

public class GenericLector implements Animate
{
    private InputStreamType inStream;
    private AnimatorThread mover;

    private LectorListener ll;

    public GenericLector( InputStreamType inStream )
    {
        this.inStream = inStream;

        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void addLectorListener( LectorListener ll )
    {  
        this.ll = ll;
    }

    public void act()
    {
        this.ll.messageRead( this.inStream.read() );
    }
}
Encapsulating Communications

We have seen how to write the code for a generic Scribe, a class that manages writing to an output stream. We have also seen how to write a generic Lector that actively reads from an input stream. Often, it is useful to package these two functions together. In the single resulting class, we consolidate all management of communications with a single remote entity. This object may add functionality. It may, for example, do some packing or unpacking for us (if we don't want to and receive objects in the same form that we use them within our program). It may do other bookkeeping, for example recording what information comes in or timestamping it. Such a communications manager might also establish the streams initially, handle exceptions, and otherwise provide a single point of contact for the rest of the entities with which it interacts directly. From within its local community, this entity provides an interface to the remote entity.

These two classes can be combined into a single class:

```java
public class LectorScribe implements Scribe, Animate {
    private OutputStreamType outStream;
    private InputStreamType inStream;
    private AnimatorThread mover;
    private LectorListener ll;

    public LectorScribe( OutputStreamType outStream,
                         InputStreamType inStream )
    {
        this.outStream = outStream;
        this.inStream = inStream;
        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void act()
    {
        this.ll.messageRead( this.inStream.read() );
    }

    public void send( MessageType m )
    {
        this.out.write( m );
        this.out.flush(); // (maybe)
    }

    public void addLectorListener( LectorListener ll )
    {
        this.ll = ll;
    }
}
```
Note that this class will often have (at least) two instruction-followers active in it: the AnimatorThread named by this.mover and whatever Thread invokes this object's send method (from outside this class).

**Real Streams**

So far, we have been discussing input streams and output streams as hypothetical idealized objects. In Java, there are a series of classes that actually implement this stream behavior. In this section, we will look at the Java classes that implement stream behavior. All of the classes described in this section are defined in the package java.io unless otherwise specified. Further information on many of these classes are included in the [Java IO Quick Reference](http://www.cs101.org/ipij/net.html) appendix.

**Abstract Stream Classes**

Java actually has four abstract classes that implement stream behavior: two input stream types, from which you can read, and two output stream types, from which you can write. The input stream classes are called `InputStream` and `Reader`. The output stream classes are `OutputStream` and `Writer`. In this chapter, we use the term stream to refer generically to all four of these classes. Each of these classes is abstract, meaning that any instance of that class is actually an instance of some subclass. They are all defined in the package java.io.

A stream is a resource containing a sequence of values. The values in the resource underlying an `InputStream` or an `OutputStream` are stored as bytes, i.e., eight bit pieces of data. The values in the resource underlying a `Reader` or `Writer` are stored as chars, i.e., sixteen bit data. Certain contexts produce byte streams, while others produce char streams. You do not need to worry about the differences, but you do need to keep track of which one you have.

Every stream has a `public void close()` method. This method frees up the underlying resources that have been used to create this stream. When your program is done with a stream, it should call that stream's close() method. When your program shuts down, any open resources will be closed automatically; however, it is good practice to close your streams as soon as you are done with them.[Footnote: Although Java includes automatic garbage collection -- it will throw away your stream object if nothing in your program can possibly access it any more -- Java does not necessarily release the underlying system resource (i.e., the actual connection to a file or whatever else your stream is connected to) at that time.]

`InputStream`, `Reader` and their extensions support a variety of methods for reading. `InputStream`'s `read()` method returns a byte, while `Reader`'s `returns a char`. `OutputStream`, `Writer` and their extensions support methods for writing. The write method of `OutputStream` takes a byte as its argument.[Footnote: Actually, `OutputStream`'s write method takes an int, but it only writes the low order byte of that int to the stream.] The write method of `Writer` takes either a char or an int or a `String`.

Each of the abstract stream classes has several subclasses that provide additional behavior. For example, some of these classes provide a wider range of methods, such as `public Object readObject()` and `public void println(String)`. You will often find it more useful to use one of these extended classes. Those classes are discussed in the next sections; their details are summarized in the [Java IO Quick Reference](http://www.cs101.org/ipij/net.html) appendix.
Many stream methods also potentially throw an exception. The most common exception to be thrown by a stream method is `IOException`. For example, when you go to read from a stream, if the underlying resource has somehow been corrupted, the read method may throw `IOException`. When using a stream method, you will often need to catch this exception. `IOException` also has several more specific subclasses, each applicable to a particular failure condition.

**Decorator Streams**

Java uses a technique called decoration to add features to streams. For example, suppose that you have an `InputStream` but have decided that you'd really rather have a `Reader`. Java has a class called `InputStreamReader` that is a special kind of Reader. Specifically, `InputStreamReader`'s constructor takes an `InputStream` as an argument. The resulting `InputStreamReader` uses the same underlying stream resource as the `InputStream` argument, but the `InputStreamReader` is a Reader, not an `InputStream`:

```java
Suppose you have an InputStream called `in`, and execute

    Reader reader = new InputStreamReader(in);

    Now reader.read() returns the first char in the underlying input stream. The streams named `in` and `reader` use the same underlying input stream!
```

This pattern -- adding features by constructing a more sophisticated object around a simpler one -- is called decoration. Java streams make extensive use of decoration to add features. For example, you can now treat reader as you would any `Reader`, decorating it further using the appropriate constructors.[Footnote: There is, however, no way to make an `InputStream` from a `Reader` (or an `OutputStream` from a `Writer`).]

Some of the decorations that you might wish to apply to your stream include:

**Buffering**. This reads a larger group of data from the stream into some hidden storage, and then reads from that storage on demand. This is particularly useful when you are reading from a file or a network connection. Buffering is provided by the `BufferedInputStream` and `BufferedReader` classes. `BufferedReader` also has a particularly useful `readLine()` method that returns a whole `String`, up to but excluding the terminating newline.

**Data**. `DataInputStream` is a class whose instances provide a variety of read methods that allow you to read Java primitive data. These include `readInt()`, `readBoolean()`, etc. Note, however, that there is no corresponding `DataReader` class.

**Objects**. An `ObjectInputStream` is very much like a `DataInputStream` with the addition of a method for reading whole Java Objects: `readObject()`. However, only objects that implement the Serializable interface may be read from an `ObjectInputStream`.

There are similar decorations on the output side. An `OutputStream` can be used to create a `Writer` using `OutputStreamWriter`'s constructor: `new OutputStreamWriter( yourOutputStream )`. On the output side, buffering also enhances efficiency, especially when writing to a file or network connection. The `BufferedWriter` also has a `newLine()` method. There are also Data and Object `OutputStream` classes. Only `Serializable` objects can be written to an `ObjectOutputStream` or `ObjectWriter`.
Finally, there are a pair of classes called PrintStream and PrintWriter.[Footnote: You should use PrintWriter, rather than PrintStream, if you want to create an instance of this kind of output stream. PrintStream exists only for compatibility with certain objects already built in to Java.] These output stream classes have the special advantage that none of their methods throws IOException. Their methods are called print and println, rather than write, to indicate their non-exception-throwing status. There are print and println methods for essentially every type of Java primitive. Using an Object's toString() method, print and println can also print any kind of Java Object. This makes these output stream types very useful for writing messages, e.g. to the Console.

There are several other decorator stream types defined in the Java.io package. Many of those are designed for special purposes. A few are documented in the Java IO Quick Reference appendix of this book.

Stream Sources

Now that you know how to manipulate streams, you may be wondering where you can find one. Streams come from a variety of different sources, depending on the resources that they connect.

For example, there are a series of streams that communicate with Files. These are called FileInputStream, FileOutputStream, FileReader, and FileWriter. Their constructors take the name of the file to read from or write to. These streams allow information to be read from or written to persistent storage, such as a disk. Since disk interactions are relatively slow, it is common to combine several disk access operations using the appropriate kind of buffered stream.

Another source of streams is pipes. A PipedInputStream (or OutputStream, Reader, or Writer) can be used to communicate between two Java objects. To do this, you must create a matched pair (PipedInputStream and PipedOutputStream or PipedReader and PipedWriter), then use the connect() method of one piped stream to join it to its mate.

We will look more closely at networked streams -- streams that communicate between two computers -- below. There are also streams that read from or write to arrays or Strings.

There are two additional streams with which you are already familiar, though you do not know it. These are the streams called the standard input and standard output. They are the streams that connect to the Java console. So far, you have used these through the cs101.io.Console class. In fact, there are two streams corresponding to the methods of Console.

Both System.in and System.out are static fields of the class java.lang.System. System.in is the standard input (or "stdin") stream, while System.out is the standard output ("stdout"). There a third stream, System.err, the standard error stream ("stderr"), that also writes to the Java console by default. System.err is intended for error messages, while System.out is intended to output to the user.

The type of System.in is simply InputStream. The type of System.out, however, is java.io.PrintStream. A PrintStream supports textual output of most Java primitive types as well as objects. It also avoids most of the otherwise-ubiquitous IOExceptions.

Decoration in Action
As we have seen, the four abstract IO classes lack some basic useful features and methods. Frequently, you would really rather be using one of their non-abstract subclasses. For example, one very common reader is BufferedReader. Instances of the BufferedReader class support useful methods such as

```java
public int read() throws IOException;
public String readLine() throws IOException;
```

The first of these reads a single character from the input; the second reads an entire line of input. For a more complete list of BufferedReader methods, see the Java IO Quick Reference appendix.

To create a BufferedReader, you first need to have a Reader. Java doesn't come with any predefined Readers, but it does come with a built-in InputStreamReader: System.in, which reads from the Java console. The following (extremely useful) code assigns the name myIn to a BufferedReader that gets its input from System.in:

```java
BufferedReader myIn = new BufferedReader( new InputStreamReader( System.in ) );
```

In general, you can cascade the feature types, i.e., take any arbitrary stream and make another (more featureful) stream out of it. You begin with a particular stream, either based on an external (non-stream) structure or built in. Below, we will focus on streams created from a network interface.

**Network Streams: An Example**

In this final section, we will revisit the LectorScribe class that we defined above. Using what we have learned about actual Java streams, we will embellish that class so that it can be used to communicate with other computers running over the network. To do this, we will need to add the machinery of network communications: sockets. In Java, sockets and other network communication classes are implemented in a package called java.net. These classes are also covered in the Java IO Quick Reference appendix of this book.

**Starting from Streams**

The following code reproduces the LectorScribe class, above, with a few minor modifications. First, we have used actual java.io stream classes (in this case, Reader and Writer) as the stream types. We have also specified String as our MessageType and added some error messages when IOExceptions are caught.

```java
public class LectorScribe implements Scribe, Animate
```
private Writer out;
private Reader in;
private AnimatorThread mover;
private LectorListener ll;

public LectorScribe( Writer out, Reader in )
{
    this.out = out;
    this.in = in;
    this.mover = new AnimatorThread( this );
    this.mover.start();
}

public void act()
{
    try
    {
        this.ll.messageRead( this.in.read() );
    }
    catch (IOException e)
    {
        System.err.println( "Oops, I couldn't read a line!" );
    }
}

public void send( String m )
{
    try
    {
        this.out.write( m );
        this.out.flush(); // (maybe)
    }
    catch (IOException e)
    {
        System.err.println( "Oops, I couldn't write a line!" );
    }
}

public void addLectorListener( LectorListener ll )
{
    this.ll = ll;
}

Recall the logic of this code: A LectorScribe is responsible for reading from and writing to its streams. This involves continually monitoring the input with an active Thread (in the act() method) as well as being responsive to requests to write to the output (when send( String ) is called from another Thread).

Decorating Streams
But what if you are given an InputStream and an OutputStream rather than a Reader and a Writer? In this case, we might add another constructor to this class, one which decorates these byte streams with their char equivalents. Only the additional constructor is reproduced here.

```java
public LectorScribe( OutputStream out, InputStream in )
{
    this( new OutputStreamWriter( out ),
           new InputStreamReader( in ) );
}
```

Recall that a this() constructor invokes another constructor of the same class. Invoking new LectorScribe( System.in, System.out ) results in the invocation of

```java
new LectorScribe( new OutputStreamWriter( System.out ),
                   new InputStreamReader( System.in ) )
```

This would create a LectorScribe that writes on demand to the standard output stream and continually reads from the standard input stream.

**Sockets and Ports**

Where might you get input and output streams in the first place? The answer depends on what these streams are supposed to connect you to. For example, if you are reading from (or writing to) a file, you could use the FileInputStream/FileOutputStream or FileReader/FileWriter class pairs. In this section, we will explore streams that connect you to other computers. Java contains a standard library package called java.net that provides most of the infrastructure for making network connections.

Think back to the tin can telephone example. What we really want is a sort of a place on the other computer that we can connect to: someplace to "plug in" the tin can telephone. Computers have a number of such things, called *ports*, but you won't see them if you look at the back of a computer. Instead, a port is a virtual place to plug in a special kind of connection, called a *socket*.

A *socket* is an abstraction of actual network connections and protocols. It contains two streams: one for input, one for output. In other words, it is the virtual equivalent of a two-way pair of tin can telephones.

To establish a socket connection, you need to run a program at each end (i.e., one program on each of the two computers that the socket will connect).

- One of these programs "listens" for connection requests; this is called the server because it is providing the service of enabling socket connections. The server provides this service on a particular port of its machine. That is, the server needs to know which port to be watching to see whether anyone is trying to connect.
- The other program is called the client, and it contacts the server to open a socket connection. The client program needs to specify what machine to contact, typically using the name of that machine, and also what port on that machine to try to connect to.

Remember that the terms client and server are relative to a particular service. In this case, the server is providing the service of listening for socket connections, while the client is making use of that service. Once the socket is in place, though, it looks exactly the same from both ends.
Using A Socket

Using this idea of sockets, we can now read and write across the network. In Java, a socket is implemented by an instance of the class `java.net.Socket`. Suppose that we have one of these Java Sockets and want to read from and write to it using a LectorScribe.

We already know how to create a LectorScribe if we are given either a Reader and a Writer, or an InputStream and an OutputStream. A Java `Socket` has a method to access each of its streams: `getInputStream()`, which returns an `InputStream`, and `getOutputStream()`, which returns an `OutputStream`. If we had a socket -- one end of a virtual two-way tin can telephone -- we could access its input and output streams using these methods. We can accomplish this using yet another LectorScribe constructor:

```java
public LectorScribe( Socket sock ) throws IOException
{
    this( sock.getOutputStream(), sock.getInputStream() );
}
```

In creating a LectorScribe for this `Socket`, we simply extract the streams and use them to create a LectorScribe on an output and an input stream. Using the remainder of the LectorScribe code above, we have a simple program that takes a `Socket` as an argument and transmits what it reads and writes over the `Socket` to the user via the Java console. Note, however, that this constructor risks throwing an `IOException`. This is because the `Socket` might be corrupt and the streams might not be accessible.

A final note on `Socket`s: Like a stream, a `Socket` has a `close()` method. You should make a point of closing your `Socket` when you're done with it.

Opening a Client-Side Socket

Now we have code to read and write from a `Socket`, we need to figure out where to get a `Socket` in the first place. As described above, we can get a `Socket` by connecting to a server -- a machine that is listening for connection requests -- on a particular port. We need to know what machine to connect to, specified by a String corresponding to its hostname, such as "www-cs101.ai.mit.edu". We also need to know on what port the server is listening for our connection. The port is specified by an integer. By convention, ports numbered below 1024 are reserved for "standard" protocols. Otherwise, you have fairly free choice of ports.

A `java.net.Socket` is created by calling its constructor with a `String` corresponding to the hostname of the machine you want to connect to and an `int` representing the port on that machine where something is listening for connections. So, if we had this information, we could use the following LectorScribe constructor:

```java
public LectorScribe( String hostname, int port ) throws IOException
{
    this( new Socket( hostname, port ) );
}
```

[Footnote: Note that the constructor for `Socket()` may throw `IOException`.]
This would enable us to say, e.g.,

    new LectorScribe( "www-cs101.ai.mit.edu", 8080 )

If we put this expression into our public static void main method, running this program would create a program that connects the user to the machine www-cs101.ai.mit.edu on port 8080. Anything the user types would be sent to that port on that machine, and anything that www-cs101.ai.mit.edu writes to port 8080 would be printed on the Java console. This is the complete program!

Opening a Single Server-Side Socket

Of course, to make the client side of this program work, something has to be listening on the appropriate port of the appropriate machine. What code should we run on www-cs101.ai.mit.edu to listen on port 8080?

The port listener code requires another class from the package java.net. This one is somewhat misleadingly named ServerSocket. A Java program uses a `java.net.ServerSocket` to listen for connections. To create a ServerSocket, you need to specify what port to listen on. Remember that this is the local port -- the port on the machine this code is running on -- and you are not making any connections, just waiting for someone else to contact you. (If someone throws you a pair of tin cans, you should catch them and use them to communicate.

The port number on which you listen is arbitrary, but it must match the port number on which the client will try to connect. (The client should also use the hostname of the computer on which this ServerSocket is running.) Remember that the port number should be at least 1024.

We will need to add two constructors to LectorScribe. The first simply creates the ServerSocket and invokes the LectorScribe constructor that takes a ServerSocket as an argument:

    public LectorScribe( int port ) throws IOException
    {
        this( new ServerSocket( port ) );
    }

The action is really in this second constructor. This constructor says "listen on your port". The method

    public void Socket ServerSocket.accept() throws IOException;

is a blocking method that returns a Socket when a connection has been made:

A ServerSocket's accept() method returns a Socket. Specifically, it waits until some program tries to connect to that port, then returns its own side of that connection.

    public LectorScribe( ServerSocket serv ) throws IOException
    {
        this( serv.accept() );
    }
This complete LectorScribe is now ready to run on both sides of the network. By having one main program -- on a computer named \textit{yourComputerName} -- run

\begin{verbatim}
    new LectorScribe( 4321 )
\end{verbatim}

and the other run

\begin{verbatim}
    new LectorScribe( yourComputerName, 4321 )
\end{verbatim}

you can create a simple two-way chat program. The number 4321 is, of course, an arbitrary choice, but both programs must use the same number.

The complete LectorScribe code is included in the code supplement (as \texttt{LectorScribe.html}).

\textbf{A Multi-Connection Server}

The \texttt{accept()} method, like an input stream's \texttt{read()} method, blocks until there is a connection ready to accept. So, like a \texttt{read()}, \texttt{accept()} -- and this method -- may wait for a very long time before returning. This means that it may be useful to have the \texttt{accept()} invocation run in its own Thread. We can write a variant on the LectorScribe by separating the connection listening from the rest of the program.

In fact, we may want to go further. A single application can have several connections active at once. There is no problem with having multiple connections running over the same port. A port is simply a place where a ServerSocket can be listening for connection requests. For these reasons, it is common to write a more sophisticated kind of server than a simple LectorScribe.

Essentially, the LectorScribe that we have seen so far is run on a Socket, not on a ServerSocket. An additional class is used solely to listen on the Socket. This class needs to have its own instruction follower, so it is an animate object. When it accepts a connection -- yielding a Socket -- it simply creates a LectorScribe on that Socket.

\begin{verbatim}
public class MultiServer implements Animate {

    private ServerSocket serv;
    private AnimatorThread mover;

    public MultiServer( int port ) throws IOException {
        this.serv = new ServerSocket( port );
        this.mover = new AnimatorThread( this );
        this.mover.start();
    }

    public void act() {
        try {
            new LectorScribe( this.serv.accept() );
        } catch (IOException e)
\end{verbatim}
Server Bottlenecks

The server architecture that we have just described puts one computer in the middle of a network. This is sometimes called a **hub-and-spoke** architecture, since all connections run through the central server, or hub. There are advantages and disadvantages to this architecture. One of the major potential disadvantages is that the server can be overwhelmed if it receives more traffic than it can handle. In this case, the server has become a **bottleneck**, the difficult point where congestion must be relieved. The good news is that in a single-server model, upgrading the server is likely to significantly improve system performance.

Hub-and-spoke architecture is very common in networks. When increased reliability is needed, there are variant architectures that reduce the reliance on a single potential point of failure. The most extreme of these is one in which every computer connects with every other computer (on an as-needed basis). This amounts to a whole lot of LectorScribes talking with each other, without the added MultiServer code. This kind of architecture is called **peer-to-peer** communication, because neither of the participants is particularly more important. In that case, one plays the role of the server and the other the client only to establish the socket connection; after that, the two machines are equivalent.

A common variant on the hub and spoke, in which each server is in turn the client of a super-server (which may itself be a client...) makes for more efficient routing. This is called a **hierarchical** architecture. It is the basis of, for example, computer name lookup (also called **domain name service**) on the Internet.

Chapter Summary

- An InputStream is a Java abstraction describing an entity from which Things can be read; an OutputStream is an entity to which Things can be written.
- Streams can be used for I/O on the console, files and network connections, as well as certain Java objects like arrays and strings.
- Streams can have features like buffering, filtering, or automatic data formatting. These features can be cascaded using the appropriate stream class's constructor.
- Every Java instantiation has a PrintStream called System.out and an InputStream called System.in.
- objectInputStream and ObjectOutputStream are stream types that can be particularly useful for sending objects across the network.
- A Socket is one side of a network connection. It has an InputStream and an OutputStream. You can create a Socket by specifying the hostname and port to which you wish to connect.
- A ServerSocket is something that can accept connection requests on a particular port. You can create a ServerSocket, by specifying which port to listen on. A ServerSocket's accept() method returns a Socket object each time a new connection is made.
• A multithreaded server is an entity that creates a new self-animating object to handle each connection accepted by its ServerSocket.

• Such a server can be a hub for a network, but when it is overloaded, it can also be a communications bottleneck.

Exercises

Q. Write code to open a file and read it, one line at a time, printing each line to the standard output.

Q. Modify the LectorScribe so that it shuts down gracefully. That is, when a stream throws an exception -- e.g., when there is nothing more to read -- it should close its streams and its Socket.

Q. Modify the MultiServer so that it keeps track of the LectorScribes that it has created. Add something to the act() method of the MultiServer that sends a message over the output stream of each LectorScribe when a new connection is accept(ed. ("Congratulations on your new sibling!")

Q. Create a new kind of LectorListener event handler that notifies the MultiServer whenever one of its LectorScribes reads something from its input stream.

Q. Combine the answers to the previous two questions so that, when a message is read by one LectorScribe, it is broadcast to all of the LectorScribes' output streams. Bonus: Can you avoid sending the message to the initiating client?

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Applets

An applet is a piece of Java code that can be run under certain network browsers (and appletviewer, a Java program). Applets are embedded in html and invoked by viewing the page (or running appletviewer on the page). Every applet extends java.applet.Applet, which in turn extends java.awt.Panel. When an applet is invoked, an instance is created (i.e., its constructor is called). No arguments are supplied to the constructor; instead, there is html syntax for providing parameters to applets. At applet creation time, three methods are called in sequence:

1. the applet's constructor
2. the applet's public void init() method.
3. the applet's public void start() method.

Each of these is provided by java.applet.Applet, but can be overridden by the subclass. The init method will be called exactly once. The start method may be called repeatedly, e.g., each time the applet scrolls off of and then back on to the page. Applets also inherit stop and destroy methods (both public void, no parameters) which are called when the applet temporarily disappears or is permanently removed, respectively. It is conventional to start and stop any Threads that the applet uses in the applet's start and stop methods. In this sense, start serves some of the role of public static void main( String[] ) in standalone Java applications. (Other parts of that role may be played by init or even by the constructor.)

The primary differences between applets and standalone applications are:

- An instance of the applet is always created, and its constructor, init, and start methods are always run. (These are the only things guaranteed to run, but both stop and destroy may also be called.) In addition, because an Applet instance is a Panel instance, a visible component is created, awt events are (potentially) handled, etc.
- When a standalone application is invoked, only public static void main( String[] ) (and code called by it) is run.

Other than this information, applets are largely outside the scope of this course.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>

```
WHILE (TRUE) {
    ECHO
 }
```
Java.awt Quick Reference

- AWT Components
- Component
- Canvas
- Widgets and their Event Types
- Basic Widgets
- ItemSelectable Widgets
- Text Widgets
- Container
- Panel and Frame
- Dimension, Point, and Rectangle
- Graphics
- AWT Events
- ActionEvent and ActionListener
- AWT Listeners and Adapters

AWT Components

An awt component is a visible gui entity. The root of the component hierarchy is the class java.awt.Component.

The class java.awt.Component is abstract. Its methods include:

- public void paint(Graphics g), an event-handler method supplying detailed instructions as to how to paint the Component.
- public void repaint(), a user-invoked method requesting a paint.

Specific widgets extending Component include

- Button, which has a label and can respond to being pressed.
- Label, a non-editable piece of text.
- TextField, a single line (potentially editable) text box.
- TextArea, a multi-line (potentially editable) text box.
Checkbox, which can be checked or unchecked. If a Checkbox is part of a CheckboxGroup, only one Checkbox in the CheckboxGroup may be checked at any time.

Choice, a popup menu, which contains a set of items. One of these items may be selected.

List, a Component with multiple Strings, some of which are selectable.

Most of the activity of these widgets is accomplished through the use of specialized event handlers, as described in the chapter on Event Delegation.

Two other Components deserve special mention:

1. Canvas, which does nothing by itself, but is often extended.
2. Container, an abstract Component capable of holding other Components inside it.

There are several varieties of Container, including

1. Panel, an instantiable Container.
2. Frame, a top level (outermost) Container

Component

This abstract class is the root of the visible AWT classes. All of the other classes extend it and inherit its methods. However, few subclasses rely on the full generality of Component and most of these methods are unused in most of Component's subclasses. If you want to exploit the behavior of Component, it is common to extend Canvas, the generic instantiable Component.

java.awt.Component

- abstract
- extends Object
- To cause the Component to be (re-)displayed on the screen, call its repaint() method:
  - public void repaint();
  - public void repaint( long time );
  - public void repaint( int x, int y, int width, int height );
  - public void repaint( long time, int x, int y, int width, int height );
- To give instructions for how the Component ought to look when it is time for it to appear, override its paint( Graphics g ) method:
  - public void paint( Graphics g );
- Every Component that is not a Window is inside another, called its parent:
  - public Container getParent();
- If you want to know how big the Component is...
  - public Dimension getSize();
- Component event types:
public synchronized void addComponentListener( ComponentListener l );
public synchronized void addFocusListener( FocusListener l );
public synchronized void addKeyListener( KeyListener l );
public synchronized void addMouseListener( MouseListener l );
public synchronized void addMouseMotionListener( MouseMotionListener l );
public synchronized void removeComponentListener();
public synchronized void removeFocusListener();
public synchronized void removeKeyListener();
public synchronized void removeMouseListener();
public synchronized void removeMouseMotionListener();

* Override these to specify a different size from the default for your Component
  * public Dimension getMaximumSize();
  * public Dimension getMinimumSize();
  * public Dimension getPreferredSize();

* Used for double-buffering:
  * public Graphics getGraphics();

There are many, many other methods available in java.awt.Component. However, the vast majority of these (and even several of the ones listed here) are not relevant to the material covered in this book. Check the on-line Java API documentation for details.

**Canvas**

java.awt.Canvas

A Canvas is an instantiable Component. It has no additional behavior beyond that inherited from Component. It is often extended and customized, particularly by overriding its paint() method or supplying specialized event listeners.

* extends Component
* public Canvas();
* Canonical usage:
  * subclass Canvas, create instance of subclass, add this (subclassed) Canvas to Container:
    class SpecialCanvas extends Canvas{ ... }
  * common to override paint()
  * common to addMouse(Motion)Listener

**Widgets and their Event Types**

Button, Checkbox, Choice, List, TextArea and TextField, are each types of GUI widgets. Each is a subclass of java.awt.Component and a member of the package java.awt.
<table>
<thead>
<tr>
<th>Component Name</th>
<th>Description</th>
<th>Main Event Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.Button</td>
<td>Clickable button with label. Clicking on this component generates an ActionEvent. Label with on/off mark. Clicking this item causes its state (checked/unchecked) to change.</td>
<td>java.awt.event.ActionEvent</td>
</tr>
<tr>
<td>java.awt.Checkbox</td>
<td>If a Checkbox is part of a CheckboxGroup, at most one Checkbox in the group can be selected.</td>
<td>java.awt.event.ItemEvent</td>
</tr>
<tr>
<td>java.awt.Choice</td>
<td>Popup with a list of labels from which a single item can be selected</td>
<td>java.awt.event.ItemEvent</td>
</tr>
<tr>
<td>java.awt.Label</td>
<td>A non-editable text item.</td>
<td>none</td>
</tr>
<tr>
<td>java.awt.List</td>
<td>List of labels, each of which may be selected or not. Clicking an item toggles (flips) its state.</td>
<td>java.awt.event.ItemEvent</td>
</tr>
<tr>
<td>java.awt.TextArea</td>
<td>A multi-line text box.</td>
<td>java.awt.event.TextEvent</td>
</tr>
<tr>
<td>java.awt.TextField</td>
<td>Box into which a single line of text may be typed. Hitting the return key causes an ActionEvent.</td>
<td>java.awt.event.ActionEvent</td>
</tr>
</tbody>
</table>

The major methods of each widget type are listed in separate sidebars, below.

### Basic Widgets

**java.awt.Label**

- **constructors**
  - public Label();
  - public Label( String text );
  - public Label( String text, int alignment );
- **alignment management:** Symbolic constants and getter/setter for Label text alignment
  - public static final int CENTER, LEFT, RIGHT
  - public int getAlignment();
  - public synchronized void setAlignment( int alignment );
- **text management:** What should the Label say?
  - public String getText();
  - public synchronized void setText( String text );
- **Canonical usage:**
  - create Label, add Label to Container:
    Label l = new Label(text); container.add(l);

**java.awt.Button**
● constructors
  ● public Button();
  ● public Button( String label );
● label management: What text should appear next to the Button?
  ● public String getLabel();
  ● public synchronized void setLabel( String label );
● ActionListener management: Who needs to know when this Button is pressed?
  ● public synchronized void addActionListener( ActionListener l );
  ● public synchronized void removeActionListener( ActionListener );
● Canonical usage:
  ● create Button, addActionListener to Button, add Button to Container:
    Button b = new Button(label); cb.addActionListener(listener); container.add(b);

**Item Selectable Widgets**

These widgets each contain multiple items, one or more of which may be selected at any time. Each implements an interface specifying certain behavior. The methods of this interface are not repeated for each of the implementing classes below.

**java.awt.ItemSelectable (interface)**

● Listener management: Who needs to know when one of the items is selected or deselected?
  ● public void addItemListener( ItemListener l );
  ● public void removeItemListener( ItemListener l );
● public Object[] getSelectedObjects; returns null if none currently selected.

**java.awt.Choice (a.k.a. dropdown list)**

Has a set of indexed String items. Generates ItemEvents.

● implements ItemSelectable
● constructor
  ● public Choice();
● item management:
  ● public synchronized void add( String item );
  ● public synchronized void addItem( String item );
  ● public synchronized void insert( String item, int index );
  ● public synchronized void remove( String item );
  ● public synchronized void remove( int index );
  ● public synchronized void removeAll();
  ● public String getItem( int index );
public int getItemCount(); returns how many there are currently.

item selection management: Which item is currently selected?
public synchronized void select( int index );
public synchronized void select( String item );
public int getSelectedIndex();
public synchronized String getSelectedItem();

Canonical usage:
create Choice, add items to Choice (one by one), addItemListener to Choice, add Choice to
Container:
Choice c = new Choice(); /* repeatedly */ c.add( label ); c.addItemListener(listener);
container.add(c);

java.awt.Checkbox

Has a label, a state (clicked or not), and possibly a CheckboxGroup. Generates ItemEvents.

- implements ItemSelectable
- constructor
  - public Checkbox();
  - public Checkbox( String label );
  - public Checkbox( String label, boolean state );
  - public Checkbox( String label, boolean state, CheckboxGroup group );
  - public Checkbox( String label, CheckboxGroup group, boolean state );
- label management: What text should appear next to the Checkbox?
  - public String getLabel();
  - public synchronized void setLabel( String label );
- state management: True is checked, false is unchecked
  - public boolean getState();
  - public void setState( boolean state );
- group management: Is this checkbox part of a group of mutually exclusive alternatives?
  - public CheckboxGroup getCheckboxGroup();
  - public void setCheckboxGroup( CheckboxGroup group );
- Canonical usage:
  - create Checkbox, addItemListener to Checkbox, add Checkbox to Container:
    Checkbox cb = new Checkbox(label); cb.addItemListener(listener); container.add(cb);
  - OR create Checkbox in CheckboxGroup, addItemListener to Checkbox, add Checkbox to
    Container:
    Checkbox cb = new Checkbox(label, group); cb.addItemListener(listener); container.add(cb);

java.awt.CheckboxGroup
• Not a Component!
  • extends Object implements Serializable
• constructor
  • public CheckboxGroup();
• Given a group, you can get the currently selected Checkbox:
  • public Checkbox getSelectedCheckbox();

## Text Widgets

These three widget types provide varying kinds of text display and editing. TextField is by far the simplest, especially as it relies on ActionEvents triggered only when editing is "complete", e.g., when the user hits return. TextEvents allow finer-grained access to the user's editing.

### java.awt.TextComponent

Parent class for TextArea, TextField; less commonly used directly.

• TextListener management: Who should listen to random text changes. Note: it is more common to use an ActionListener with a TextField
  • protected transient TextListener textListener;
  • public void addTextListener( TextListener l );
  • public void removeTextListener( TextListener l );
• Manipulating selected (highlighted) text:
  • public synchronized String getText();
  • public synchronized int getSelectionStart();
  • public synchronized int getSelectionEnd();
  • public synchronized void select( int startIndex, int endIndex );
  • public synchronized void selectAll();
  • public synchronized void setSelectionStart( int index );
  • public synchronized void setSelectionEnd( int index );
• Basic text manipulation:
  • public synchronized String getText();
  • public synchronized void setText( String text );
• Where is insertion point?
  • public int getCaretPosition();
  • public void setCaretPosition( int index );
• Can user edit text?
  • public boolean isEditable();
  • public synchronized void setEditable( boolean state );
java.awt.TextField

A single line of text, with facility for hiding (e.g., as password). Primary event type is ActionEvent, not TextEvent.

- extends TextComponent
- constructors
  - public TextField();
  - public TextField( String text );
  - public TextField( int columns );
  - public TextField( String text, int columns );
- Size in columns, i.e., how wide can this line of text be. Note also interacts with component size.
  - public int getColumns();
  - public void setColumns( int columns );
  - public Dimension getMinimumSize( int columns );
  - public Dimension getPreferredSize( int columns );
- If echoChar is set, text typed into the TextField will appear as echoChar. This is useful if the information typed is secret, e.g., a password.
  - public void echoCharIsSet();
  - public char getEchoChar();
  - public void setEchoChar( char echoChar );
- ActionListener is TextField's main event handler. It is triggered when the return (or enter) key is pressed.
  - public synchronized void addActionListener();
  - public synchronized void removeActionListener();
- Canonical usage:
  - create TextField with default size, addActionListener to TextField, setEditable, add TextField to Container:
    TextField tf = new TextField(columns); tf.addActionListener(listener); tf.setEditable(true); container.add(tf);

java.awt.TextArea

A full scrollable block of text. Inherits much of its behavior from TextComponent.

- extends TextComponent
- constructors
  - public TextArea();
  - public TextArea( String text );
  - public TextArea( int rows, int columns );
  - public TextArea( String text, int rows, int columns );
public TextArea ( String text, int rows, int columns, int scrollbars );

Size in columns and rows, i.e., how wide and high can this block of text appear. Note also interacts with component size.

public int getRows();
public void setRows( int rows );
public int getColumns();
public void setColumns( int columns );
public Dimension getMinimumSize( int rows, int columns );
public Dimension getPreferredSize( int rows, int columns );

Scrollbar appearance management:

public static final int SCROLLBARS_BOTH, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE, SCROLLBARS_VERTICAL_ONLY;
public int getScrollbarVisibility();

Text management (beyond TextComponent's methods):

public synchronized void append( String text );
public synchronized void insert( String text, int index );
public synchronized void replaceRange( String text, int startIndex, int endIndex );

Container

This abstract class is the root of the parent (container) AWT classes. All of the other container classes extend it and inherit its methods. Only classes extending Container can be a parent to another Component.

Container has four important subclasses:

java.awt.Panel is a generic instantiable Container. It provides no additional functionality, but is often used directly or extended to create a Container instance.

java.applet.Applet is a specialized Panel that can be used inside an applet viewer or web browser. See the appendix on Applets for further information.

java.awt.Window is a top level Container, i.e., a Container that does not itself need to be Contained. However, Window contains no platform-specific niceties (such as resizability), so it is rarely used directly.

java.awt.Frame is a subclass of Window that is commonly used in its place.

java.awt.Container

abstract
extends Component and so inherits all of its methods
protected Container();
- Contained Component management. Position is dictated by this Container's LayoutManager. In this book, we stick to the default LayoutManager.
  - public void add( Component c );
  - public void add( String name, Component c );
  - public void add( Component c, int index );
  - public Component getComponent( int index );
  - public Component getComponentAt( int x, int y );
  - public Component getComponentAt( Point p );
  - public Component getComponentCount();
  - public Component[] getComponents();
  - public void remove( int index );
  - public void remove( Component c );
  - public void removeAll();
  - public void removeContainerListener( ContainerListener l );

- Special event handler:
  - public void addContainerListener( ContainerListener l );
  - public void removeContainerListener( ContainerListener l );

There are many, many other methods available in java.awt.Container as well. Check the on-line Java API documentation for details.

**Panel and Frame**

A Frame is a top-level Window. A Panel is a generic Container. Every component must be inside a Container except a top-level (Window) Container such as a Frame.

**java.awt.Frame**

- extends Window
- implements MenuContainer
- constructors
  - public Frame();
  - public Frame( String title );
- The title is displayed on the Frame's titlebar:
  - public String getTitle();
  - public synchronized void setTitle( String title );
- Make the Frame as small as it can be while still holding all of its contained Components
  - public void pack(); *inherited from Window*
- Make the Frame visible:
  - public void show(); *inherited from Window*
• public boolean isShowing(); *inherited from Window*

• Is the user allowed to resize the Frame?
  • public boolean isResizable();
  • public synchronized void setResizable( boolean resizable );

• What to do when you're done with the Frame and its contained Components:
  • public synchronized dispose();

• Special event handler (includes window closing events)
  • public synchronized void addWindowListener( WindowListener l );
  • public synchronized void removeWindowListener( WindowListener l );

• Canonical usage:
  • create Frame, create and add Components, add WindowListener, pack Frame, show Frame
    Frame f = new Frame();
    Component c = new ComponentSubclass(); f.add( c ); /* repeat this line */
    f.addWindowListener( listener ); f.pack(); f.show();
  • OR subclass Frame, create instance of subclass

**java.awt.Panel**

• extends Container

• constructors
  • public Panel();
  • public Panel( LayoutManager lm );

• Canonical usage:
  • create Panel, create and add Components, add Panel to Container
    Panel p = new Panel();
    Component c = new ComponentSubclass(); p.add( c ); /* repeat this line */
    container.add( p );
  • OR subclass Panel, create instance of subclass

**Dimension, Point, and Rectangle**

A dimension represents length and width; a point represents x and y coordinates. A rectangle is represented in terms of its upper lefthand corner and its height and width, i.e., combining a Point and a Dimension.

**java.awt.Dimension**

• implements Serializable

• constructors:
  • public Dimension();
  • public Dimension( Dimension d );
public Dimension( int width, int height );
publicly accesible fields (!!)
   public int height;
   public int width;
A nicer way to access fields:
   public Dimension getSize();
   public void setSize( Dimension d );
   public void setSize( int width, int height );

java.awt.Point

   implements Serializable
   constructors:
      public Point();
      public Point( Point p );
      public Point( int width, int height );
   publicly accesible fields (!!)
      public int x;
      public int y;
   A nicer way to access fields:
      public Point getLocation();
      public void setLocation( Point p);
      public void setLocation( int width, int height );
      public void translate( int x, int y );

java.awt.Rectangle

   extends java.awt.geom.Rectangle2D
   implements Serializable
   constructors:
      public Rectangle();
      public Rectangle( Dimension d );
      public Rectangle( int width, int height );
      public Rectangle( int x, int y, int width, int height );
      public Rectangle( Point p );
      public Rectangle( Point p, Dimension d );
      public Rectangle( Rectangle r );
   publicly accesible fields (!!)
      public int height;
- public int width;
- public int x;
- public int y;

- A nicer way to access fields:
  - public Dimension getSize();
  - public void setSize( int width, int height );
  - public void setSize( Dimension d );
  - public double getHeight();
  - public double getWidth();
  - public Point getLocation();
  - public void setLocation( int x, int y );
  - public void setLocation( Point p );
  - public double getX();
  - public double getY();
  - public Rectangle getBounds();
  - public void setBounds( int x, int y, int width, int height );
  - public void setBounds( Rectangle r );

- Geometric predicates:
  - public boolean contains( Point p );
  - public boolean contains( Rectangle r );
  - public boolean intersects( Rectangle r );
  - public boolean isEmpty();

- Geometric computations:
  - public Rectangle intersection( Rectangle r );
  - public Rectangle union( Rectangle r );

### Graphics

A Graphics is the "screen" object on which all primitive drawing takes place. Graphics support a huge number of methods. You will almost always use the Graphics passed into a paint method when it is invoked by Java.

**java.awt.Graphics**

- abstract
- extends Object
- constructor
  - protected Graphics();
- Make pictures on this Graphics:
public abstract void clearRect( int x, int y, int width, int height );
public void draw3DRect( int x, int y, int width, int height, boolean raised );
public abstract void drawArc( int x, int y, int width, int height, int startAngle, int arcAngle );
public abstract boolean drawLine( int startX, int startY, int endX, int endY );
public abstract void drawOval( int x, int y, int width, int height );
public abstract void drawPolygon( int[] xCoords, int[] yCoords, int numCoords );
public void drawPolygon( Polygon p );
public abstract void drawPolyline( int[] xCoords, int[] yCoords, int numCoords );
public void drawRect( int x, int y, int width, int height );
public abstract void drawRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight );
public abstract void drawString( String string, int x, int y );
public void fill3DRect( int x, int y, int width, int height, boolean raised );
public abstract void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle );
public abstract void fillOval( int x, int y, int width, int height );
public abstract void fillPolygon( int[] xCoords, int[] yCoords, int numCoords );
public void fillPolygon( Polygon p );
public abstract void fillRect( int x, int y, int width, int height );
public abstract void fillRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight );

A Graphics draws in one color at a time. These methods access and change the currently active Color:

- public abstract Color getColor();
- public abstract void setColor( Color color );
- public abstract void setXORMode( Color color );

A Graphics displays text in one Font at a time. These methods access and change the currently active Font:

- public abstract Font getFont();
- public FontMetrics getFontMetrics();
- public abstract FontMetrics getFontMetrics( Font font );
- public abstract void setFont( Font font );

Copy whatever is on this Graphics to a new Graphics.

- public abstract Graphics create();
- public Graphics create( int x, int y, int width, int height );

Get rid of a Graphics you no longer need (only if you've created it!)

- public abstract void dispose();

You can manipulate java.awt.Images; see the online documentation for Java for details.

- public abstract boolean drawImage( Image image, int x, int y, ImageObserver observer );
- public abstract boolean drawImage( Image image, int x, int y, int width, int height, ImageObserver observer );
- public abstract boolean drawImage( Image image, int x, int y, Color background, ImageObserver observer );
- public abstract boolean drawImage( Image image, int x, int y, int width, int height, Color background, ImageObserver observer );
- public abstract boolean drawImage( Image image, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer );
- public abstract boolean drawImage( Image image, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color background, ImageObserver observer );

**AWT Events**

There are many different kinds of events in the package java.awt.event. Each is a subclass of java.awt.event.AWTEvent. It is unlikely that you would ever need to create an awt event. Instead, you are likely to write Listeners that handle these Events.

**java.awt.AWTEvent**

The most important method of the class java.awt.AWTEvent is

- public Object getSource();

which returns the Object to which the event occurred. Because all other awt events extend AWTEvent directly or indirectly, they, too, have getSource() methods. Their getSource() methods will generally return a Component (or an instance of one of its subclasses).

Other event objects with fields worth noting are summarized in the following table:

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Notable Event Methods</th>
</tr>
</thead>
</table>
| ActionEvent | public String getActionCommand(); *can be used to disambiguate source.*  
|             | public int getModifiers(); *indicates alt/ctrl/shift/meta keys pressed*  
|             | public Component getComponent(); *same as getSource(), but typed correctly*  
|             | public Point getPoint();  
|             | public int getX();  
|             | public int getY();  
|             | public int getClickCount();  
| MouseEvent  | public boolean isAltDown();  
|             | public boolean isControlDown();  
|             | public boolean isMetaDown();  
|             | public boolean isShiftDown();  
|             | public int getModifiers(); |
ItemEvent

public Object getItem(); \textit{returns selected item}

public ItemSelectable getItemSelectable(); \textit{same as getSource(), but typed correctly}

public int getStateChange(); \textit{returns ItemEvent.SELECTED or DESELECTED}

WindowEvent

public Window getWindow(); \textit{same as getSource(), but typed correctly}

ComponentEvent

public Component getComponent(); \textit{same as getSource(), but typed correctly}

public Component getChild(); \textit{who was added or removed}

public Container getContainer(); \textit{who it was added to/removed from. same as getSource(), but typed correctly}

WindowEvent

public Component getComponent(); \textit{same as getSource(), but typed correctly}

ComponentEvent

public Component getChild(); \textit{who was added or removed}

ContainerEvent

public Container getContainer(); \textit{who it was added to/removed from. same as getSource(), but typed correctly}

### ActionEvent and ActionListener

\texttt{java.awt.event.ActionEvent} and \texttt{java.awt.event.ActionListener}

Although \texttt{ActionEvent} does have some methods, it is most common simply to register the occurrence of an \texttt{ActionEvent}, especially if the \texttt{ActionListener} is only listening to the \texttt{ActionEvents} of a single \texttt{Component}. The \texttt{ActionEvent}'s \texttt{getSource()} method can always be used to disambiguate the source of \texttt{ActionEvents} if necessary.

The interface \texttt{java.awt.event.ActionListener} has a single method:

- \texttt{public abstract void actionPerformed( ActionEvent e );}

To handle the action events generated by a \texttt{Button} or \texttt{TextField}, you will need to write a class that implements \texttt{java.awt.event.ActionListener} and its \texttt{actionPerformed} method.

### AWT Listeners and Adapters

An Adapter provides trivial implementations of its corresponding Listener's methods. Generally, you should extend the Adapter class (if available) and override any methods you wish to handle. If you will be overriding all of the methods, you may wish to implement the Listener interface directly. You must implement the interface directly in the cases where no adapter is available.

All events are public abstract void.

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Listener Interface</th>
<th>Adapter Class</th>
<th>Listener/Adapter methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
<td>--</td>
<td>actionPerformed( ActionEvent e ); mouseClicked( MouseEvent e ); mouseEntered( MouseEvent e ); mouseExited( MouseEvent e ); mousePressed( MouseEvent e ); mouseReleased( MouseEvent e );</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener</td>
<td>MouseAdapter</td>
<td>mouseDragged( MouseEvent e ); mouseMoved( MouseEvent e );</td>
</tr>
<tr>
<td>Event</td>
<td>Listener</td>
<td>Adapter</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>ItemStateChanged</td>
<td>ItemEvent e</td>
<td>itemStateChanged(ItemEvent e);</td>
<td></td>
</tr>
<tr>
<td>windowActivated</td>
<td>WindowEvent e</td>
<td>windowActivated(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>Window gains focus, etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>windowClosed</td>
<td>WindowEvent e</td>
<td>windowClosed(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>successfully closed Window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>windowClosing</td>
<td>WindowEvent e</td>
<td>windowClosing(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>user requested Window close</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>windowDeactivated</td>
<td>WindowEvent e</td>
<td>windowDeactivated(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>Window loses focus, etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>windowIconified</td>
<td>WindowEvent e</td>
<td>windowIconified(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>windowOpened</td>
<td>WindowEvent e</td>
<td>windowOpened(WindowEvent e);</td>
<td></td>
</tr>
<tr>
<td>componentHidden</td>
<td>ComponentEvent e</td>
<td>componentHidden(ComponentEvent e);</td>
<td></td>
</tr>
<tr>
<td>componentMoved</td>
<td>ComponentEvent e</td>
<td>componentMoved(ComponentEvent e);</td>
<td></td>
</tr>
<tr>
<td>componentResized</td>
<td>ComponentEvent e</td>
<td>componentResized(ComponentEvent e);</td>
<td></td>
</tr>
<tr>
<td>componentShown</td>
<td>ComponentEvent e</td>
<td>componentShown(ComponentEvent e);</td>
<td></td>
</tr>
<tr>
<td>ComponentEvent</td>
<td>ComponentListener</td>
<td>ComponentAdapter</td>
<td></td>
</tr>
<tr>
<td>componentHidden</td>
<td>ComponentEvent e</td>
<td>componentAdded(ContainerEvent e)</td>
<td></td>
</tr>
<tr>
<td>componentMoved</td>
<td>ComponentEvent e</td>
<td>componentRemoved(ContainerEvent e)</td>
<td></td>
</tr>
<tr>
<td>componentResized</td>
<td>ComponentEvent e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>componentShown</td>
<td>ComponentEvent e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>ContainerListener</td>
<td>ContainerAdapter</td>
<td></td>
</tr>
<tr>
<td>focusGained</td>
<td>FocusEvent e</td>
<td>focusGained(FocusEvent e);</td>
<td></td>
</tr>
<tr>
<td>focusLost</td>
<td>FocusEvent e</td>
<td>focusLost(FocusEvent e);</td>
<td></td>
</tr>
<tr>
<td>TextEvent</td>
<td>TextListener</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>textValueChanged</td>
<td>TextEvent e</td>
<td>textValueChanged(TextEvent e);</td>
<td></td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener</td>
<td>KeyAdapter</td>
<td></td>
</tr>
<tr>
<td>keyPressed</td>
<td>KeyEvent e</td>
<td>keyPressed(KeyEvent e);</td>
<td></td>
</tr>
<tr>
<td>keyReleased</td>
<td>KeyEvent e</td>
<td>keyReleased(KeyEvent e);</td>
<td></td>
</tr>
<tr>
<td>keyTyped</td>
<td>KeyEvent e</td>
<td>keyTyped(KeyEvent e);</td>
<td></td>
</tr>
</tbody>
</table>

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Java.io Quick Reference

These appendices are intended to provide a quick reference to that part of Java that is likely to be useful to a student reading this book. No additional reference should be necessary to understand what is contained in the body of this book. However, the class documentation here is not intended to be complete or exhaustive. For a comprehensive listing of the methods and other properties of these classes, read the Java API documentation.

- InputStream and Reader
- OutputStream and Writer
- Sources of Streams
- InputStreamReader and OutputStreamWriter
- Files
- Pipes
- Streams that Add Features
- Buffering
- Primitive Data
- Object Streams and Serialization
- Other Useful Streams
- IOExceptions

InputStream and Reader

Both java.io.InputStream and java.io.Reader are abstract classes, that is, there are no instances of InputStream (or of Reader) that are not also instances of a subclass of InputStream (or of Reader).

Each InputStream or Reader (generically known as a stream) represents an ordered sequence of readable Things. If a read request is made and no Thing is in the stream, the read request blocks until the next Thing becomes available.

The difference between an InputStream and a Reader is that an InputStream is at base a stream of bytes, while a Reader is at base a stream of chars.

InputStream supports the following methods:

- public int read() throws IOException; reads and returns the next byte from the stream
public int available() throws IOException; \textit{returns the number of bytes that can be read without blocking}

public void close() throws IOException; \textit{should be called when you are done with the stream}

Reader supports the following methods:

- public int read() throws IOException; \textit{reads and returns the next char from the stream}
- public boolean ready() throws IOException; \textit{returns true if the next read() will not block}
- public void close() throws IOException; \textit{should be called when you are done with the stream}

If you have an InputStream, you can create a Reader using an InputStreamReader.

\textbf{OutputStream and Writer}

Both java.io.OutputStream and java.io.Writer are abstract classes. Each of these classes represents a resource to which Things can be written.

The difference between an InputStream and a Reader is that an InputStream is at base a stream of bytes, while a Reader is at base a stream of chars.

InputStream supports the following methods:

- public void write( int b ) throws IOException; \textit{writes a byte to the stream}
- public void flush() throws IOException; \textit{makes sure that any writes have actually happened}
  - \textit{without this method, sometimes writes get queued up....}
- public void close() throws IOException; \textit{should be called when you are done with the stream}
  - \textit{includes a call to flush()}

Writer supports the following methods:

- public void write( int c ) throws IOException; \textit{writes a character to the stream}
- public void write( String s ) throws IOException; \textit{writes a String to the stream}
- public void flush() throws IOException; \textit{makes sure that any writes have actually happened}
  - \textit{without this method, sometimes writes get queued up....}
- public void close() throws IOException; \textit{should be called when you are done with the stream}
  - \textit{includes a call to flush()}

If you have an OutputStream, you can create a Writer using an OutputStreamWriter.

\textbf{Sources of Streams}

There are several different ways to generate an InputStream. One is to use System.in, the "standard input" stream built in to every running Java program. Other ways involve using some resource to read from.
**java.lang.System.in** is an InputStream that is available to every Java program.

- **FileInputStream** is a class whose constructor opens a file for reading.
- **PipedInputStream** is a class that can be used to create a stream between two running Java Threads.
- **Sockets** have InputStreams that (potentially) connect multiple computers.
- **Other InputStream types** include those that read from a ByteArray or from a Sequence of other InputStreams.

Like an InputStream, a Reader can be generated from a system resource. A Reader can also be generated from an InputStream.

- **InputStreamReader's constructor** takes an InputStream and creates a Reader that reads from that underlying stream.
- **FileReader** is an InputStreamReader that can directly produce a Reader from a File or filename.
- **PipedReader** can create a stream between two Threads.
- **Other Reader types** include StringReader and CharArrayReader.

There are also several different ways to generate an OutputStream. There are two built-in OutputStreams, **System.out**, the "standard output", and **System.err**, the "standard error" stream. Other OutputStreams can be constructed from resources:

- **java.lang.System.out** and **java.lang.System.err** are OutputStreams available to every Java program.
- **FileOutputStream** is a class whose constructor opens a file for writing, creating it if necessary.
- **PipedOutputStream** is a class that can be used to create a stream between two running Java Threads.
- **Sockets** have OutputStreams that (potentially) connect multiple computers.
- **An OutputStream can also write** to a ByteArray.

Writers mimic OutputStreams in the same way that Readers mimic InputStreams. A Writer can be generated from any OutputStream, or from a system resource directly.

- **OutputStreamWriter's constructor** takes an OutputStream and creates a Writer that reads from that underlying stream.
- **FileWriter** is an OutputStreamWriter that can directly produce a Writer from a File or filename.
- **PipedWriter** can create a stream between two Threads.
- **Other Writer types** include StringWriter and CharArrayWriter.

**InputStreamReader and OutputStreamWriter**

If you have an underlying byte stream and want a character stream, making one is as simple as calling the appropriate constructor. InputStreamReader has the methods described above for Reader; OutputStreamWriter implements the methods described for Writer.
Makes a Reader out of an InputStream

- constructor
  - public InputStreamReader( InputStream in )

`java.io.OutputStreamWriter`

Makes a Writer out of an OutputStream

- constructor
  - public OutputStreamWriter ( OutputStream out )

**Files**

The File class is a platform-independent way to refer to directories and Files by name.

The corresponding stream classes are fairly unexceptional. Use the Reader/Writer classes to read and write text, the InputStream/OutputStream to manipulate raw data.

It is generally a good idea to use buffering when reading from or writing to a file. When reading from a file, the value -1 will be returned when the end of the file is reached.

To create a File, use the first constructor of FileOutputStream or FileWriter.

`java.io.File`

A platform-independent way to refer to directories and files. See also FileOutputStream to create a File.

- implements Serializable
- constructors
  - public File( String path );
  - public File( String path, String name );
  - public File( File directory, String name );
- constants: platform-dependent directory and path separators are preset for you.
  - public static final String pathSeparator;
  - public static final char pathSeparatorChar;
  - public static final String separator;
  - public static final char separatorChar;
- predicates:
  - public boolean canRead();
  - public boolean canWrite();
  - public boolean exists();
- public boolean isAbsolute();
- public boolean isDirectory();
- public boolean.isFile();

"selectors":
- public String getName();
- public String getParent();
- public String getPath();

get information about the file or directory:
- public long lastModified();
- public long length();
- public String[] list(); // lists the files in the directory

manipulate the file:
- public boolean delete();
- public boolean mkdir();
- public boolean renameTo( File newName );

java.io.FileInputStream
- extends InputStream
- constructors
  - public FileInputStream( String name ) throws FileNotFoundException;
  - public FileInputStream( File file ) throws FileNotFoundException;

java.io.FileReader
- extends InputStreamReader
- constructors
  - public FileReader( String name ) throws FileNotFoundException;
  - public FileReader( File file ) throws FileNotFoundException;

java.io.FileOutputStream
- extends OutputStream
- constructors
  - public FileOutputStream( String name ) throws IOException;
  - public FileOutputStream( String name, boolean append ) throws IOException;
  - public FileOutputStream( File file ) throws IOException;

java.io.FileWriter
- extends OutputStream Writer
constructors
• public FileWriter( String name ) throws IOException;
• public FileWriter( String name, boolean append ) throws IOException;
• public FileWriter( File file ) throws IOException;

Pipes

Pipes are useful for communication between Threads. A PipedInputStream must be connected to a PipedOutputStream, either at construction or using the connect() method. Similarly, a PipedReader must be connected to a PipedWriter.

java.io.PipedInputStream
• extends InputStream
• constructor
  • public PipedInputStream( PipedOutputStream stream ) throws IOException;
  • public PipedInputStream();
• additional method
  • public void connect( PipedOutputStream stream ) throws IOException;

java.io.PipedOutputStream
• extends OutputStream
• constructor
  • public PipedOutputStream( PipedInputStream stream ) throws IOException;
  • public PipedOutputStream();
• additional method
  • public void connect( PipedInputStream stream ) throws IOException;

java.io.PipedReader
• extends Reader
• constructor
  • public PipedReader( PipedWriter stream ) throws IOException;
  • public PipedReader();
• additional method
  • public void connect( PipedWriter stream ) throws IOException;

java.io.PipedWriter
• extends Writer
Sockets

The classes Socket and ServerSocket are part of java.net. They are another source of streams, in this case streams that bridge across the network.

java.net.Socket

The class java.net.Socket represents a virtual connection to another machine. A Socket has an InputStream and an OutputStream.

- constructors
  - public Socket( String host, int port ) throws UnknownHostException, IOException;
  - public Socket( InetAddress address, int port ) throws IOException;
  - public Socket( String host, int port, InetAddress localAddress, int localPort ) throws IOException;
  - public Socket( InetAddress address, int port, InetAddress localAddress, int localPort ) throws IOException;

- stream methods:
  - public InputStream getInputStream();
  - public OutputStream getOutputStream() throws IOException;

- when done with the Socket:
  - public synchronized void close() throws IOException;

- other selectors:
  - public InetAddress getInetAddress();
  - public InetAddress getLocalAddress();
  - public int getLocalPort();
  - public int getPort();

java.net.ServerSocket

A Socket can be used to connect to a ServerSocket.

- constructors
  - public Socket( int port ) throws IOException;
- listen for a connection:
  - public Socket accept() throws IOException;
- when done with the ServerSocket:
  - public void close() throws IOException;
- other selectors:
  - public InetAddress get InetAddress();
  - public int getLocalPort();

**java.net.InetAddress**

A constructorless class that represents an internet address.

- pseudo-constructors
  - public static InetAddress getLocalHost() throws UnknownHostException; //what machine are you running on?
  - public static InetAddress getByName( String host ) throws UnknownHostException; //what is host’s address?
  - public static InetAddress[] getAllByName( String host ) throws UnknownHostException; //more useful if host has many addresses.
- selectors for humans
  - public String getHostName();
  - public String getHostAddress();

The package Java.net also contains classes for manipulating urls and http (i.e., the web) directly.

**Streams that Add Features**

The InputStream classes listed above create InputStreams. Other InputStream classes have constructors that take any InputStream and produce a new InputStream with additional functionality. Similar classes exist for Readers, OutputStreams, and Writers.

- You might want to read or write bigger chunks from the stream. This can be done with BufferedInputStream, BufferedReader, BufferedOutputStream or BufferedWriter.
- You might want to read or write a variety of Java primitive types. To do so, use DataInputStream or DataOutputStream.
- You might want to read or write a variety of Java primitive *and* Object types (or simply Object types). The appropriate classes are ObjectInputStream and ObjectOutputStream
- Other stream types (not documented here) include
  - Pushback, which allow you to return something you've read
  - Filter, which provides general infrastructure for only letting part of the stream through.
- See also LineNumberReader and PrintStream, below.
**java.io.BufferedInputStream**

 Reads a larger chunk of data from the underlying stream and stores it in a buffer, then on individual read() calls reads from this buffer as long as data is available there. Often used when reading from disk or from the network.

 - extends FilterInputStream
 - constructors
   - public BufferedInputStream( InputStream in );
   - public BufferedInputStream( InputStream in, int bufferSize );
 - Certain InputStream methods are synchronized:
   - public synchronized int available() throws IOException;
   - public synchronized int read() throws IOException;

**java.io.BufferedOutputStream**

 Writes to a buffer, then when the buffer is full (or when flush() is called) writes the whole thing to the underlying stream at once. Often used when writing to disk.

 - extends FilterOutputStream
 - constructors
   - public BufferedOutputStream( OutputStream out );
   - public BufferedOutputStream( OutputStream out, int bufferSize );
 - Certain OutputStream methods are synchronized:
   - public synchronized void write( int b ) throws IOException;
   - public synchronized void flush() throws IOException;

**java.io.BufferedReader**

 Reads a larger chunk of data from the underlying stream and stores it in a buffer, then on individual read() calls reads from this buffer as long as data is available there. Often used when reading from a File.

 - extends Reader
 - constructors
   - public BufferedReader( Reader in );
   - public BufferedReader( Reader in, int bufferSize );
 - Added method for reading a whole line
   - public String readLine() throws IOException;
     - note: returned String does not include the end-of-line (carriage return or linefeed) character

**java.io.BufferedWriter**

 Writes to a buffer, then when the buffer is full (or when flush() is called) writes the whole thing to the underlying stream at once. Often used when writing to disk.
- Added method for writing the end-of-line character
  - public void newLine() throws IOException;

## Streams that Add Features

The InputStream classes listed above create InputStreams. Other InputStream classes have constructors that take any InputStream and produce a new InputStream with additional functionality. Similar classes exist for Readers, OutputStreams, and Writers.

- You might want to read or write bigger chunks from the stream. This can be done with BufferedInputStream, BufferedReader, BufferedOutputStream or BufferedWriter.
- You might want to read or write a variety of Java primitive types. To do so, use DataInputStream or DataOutputStream.
- You might want to read or write a variety of Java primitive and Object types (or simply Object types). The appropriate classes are ObjectInputStream and ObjectOutputStream.
- Other stream types (not documented here) include
  - Pushback, which allow you to return something you've read
  - Filter, which provides general infrastructure for only letting part of the stream through.
- See also LineNumberReader and PrintStream, below.

### Buffering

Buffering is a way of combining multiple reads or multiple writes into a single action. It is primarily used to increase efficiency, not to obtain additional functionality. However, BufferedReader is independently useful because it has a readLine() method that reads in a whole line of text; BufferedWriter has a corresponding newLine() method.

#### java.io.BufferedInputStream

Reads a larger chunk of data from the underlying stream and stores it in a buffer, then on individual read() calls reads from this buffer as long as data is available there. Often used when reading from disk or from the network.

- extends FilterInputStream
- constructors
  - public BufferedInputStream( InputStream in );
  - public BufferedInputStream( InputStream in, int bufferSize );
- Certain InputStream methods are synchronized:
  - public synchronized int available() throws IOException;
  - public synchronized int read() throws IOException;

#### java.io.BufferedOutputStream
Writes to a buffer, then when the buffer is full (or when flush() is called) writes the whole thing to the underlying stream at once. Often used when writing to disk.

- extends FilterOutputStream
- constructors
  - public BufferedOutputStream( OutputStream out );
  - public BufferedOutputStream( OutputStream out, int bufferSize );
- Certain OutputStream methods are synchronized:
  - public synchronized void write( int b ) throws IOException;
  - public synchronized void flush() throws IOException;

**java.io.BufferedReader**

Reads a larger chunk of data from the underlying stream and stores it in a buffer, then on individual read() calls reads from this buffer as long as data is available there. Often used when reading from a File.

- extends Reader
- constructors
  - public BufferedReader( Reader in );
  - public BufferedReader( Reader in, int bufferSize );
- Added method for reading a whole line
  - public String readLine() throws IOException;
  - note: returned String does not include the end-of-line (carriage return or linefeed) character

**java.io.BufferedWriter**

Writes to a buffer, then when the buffer is full (or when flush() is called) writes the whole thing to the underlying stream at once. Often used when writing to a File.

- extends Writer
- constructors
  - public BufferedWriter( Writer out );
  - public BufferedWriter( Writer out, int bufferSize );
- Added method for writing the end-of-line character
  - public void newLine() throws IOException;

**Primitive Data**

To read and write primitive data types, Java provides two classes with appropriate methods.

**java.io.DataInputStream**
Useful for reading Java primitive types.

- extends FilterInputStream
- implements DataInput
- constructor
  - public DataInputStream( InputStream in ) throws IOException;
- Additional read methods:
  - public boolean readBoolean() throws IOException;
  - public byte readByte() throws IOException;
  - public char readChar() throws IOException;
  - public double readDouble() throws IOException;
  - public float readFloat() throws IOException;
  - public int readInt() throws IOException;
  - public long readLong() throws IOException;
  - public short readShort() throws IOException;

`java.io.DataOutputStream`

Useful for writing Java primitive types.

- extends FilterOutputStream
- implements DataOutput
- constructor
  - public DataOutputStream( OutputStream out ) throws IOException;
- Synchronizes an inherited method:
  - public synchronized void write( int b ) throws IOException;
- Additional write methods:
  - public void writeBoolean( boolean b ) throws IOException;
  - public void writeByte( int b ) throws IOException;
  - public void writeBytes( String s ) throws IOException;
  - public void writeChar( int c ) throws IOException;
  - public void writeChars( String s ) throws IOException;
  - public void writeDouble( double d ) throws IOException;
  - public void writeFloat( float f ) throws IOException;
  - public void writeInt( int i ) throws IOException;
  - public void writeLong( long l ) throws IOException;
  - public void writeShort( int s ) throws IOException;

**Objects**
To read and write Objects as well as primitive data types, Java provides two additional classes that support all of the methods of DataInput/OutputStream, plus additional support for Object reading and writing.

Note that an Object to be written or read must implement the Serializable interface. Note also that Object streams do a lot of additional work in packaging/unpackaging Objects to read or write them. If you are using a stream in a time-critical way -- such as to send information between two players of a fast-paced video game -- you may wish to send primitive data, such as ints, rather than Objects encapsulating that data, such as Points.

**java.io.ObjectInputStream**

Useful for reading Serializable Objects as well as Java primitive types.

- extends InputStream
- implements ObjectInput
- constructor
  - public ObjectInputStream( InputStream in ) throws IOException, StreamCorruptedException;
- Object read method:
  - public Object readObject();
- Plus the primitive data methods listed in DataInputStream

**java.io.ObjectOutputStream**

Useful for writing Serializable Objects as well as Java primitive types.

- extends OutputStream
- implements ObjectOutput
- constructor
  - public ObjectOutputStream( OutputStream out ) throws IOException;
- Object write method:
  - public void writeObject( Object o );
- Plus the primitive data methods listed in DataOutputStream

### Object Streams and Serialization

If you want to read and write Objects as well as primitive data types, you should use Java's ObjectInput and OutputStream classes. These classes support all of the methods of DataInput/OutputStream, plus additional support for Object reading and writing.

Note that an Object to be written or read must implement the Serializable interface.
java.io.Serializable

- Must be implemented by an object to be written to or read from a file.
- Has no methods.
- If the class involves (non-transient) fields with Object types, these fields must be Serializable.
- For complex objects (including those with Threads), additional measures may need to be taken.

java.io.ObjectInputStream

Useful for reading Serializable Objects as well as Java primitive types.

- extends InputStream
- implements ObjectInput
- constructor
  - public ObjectInputStream( InputStream in ) throws IOException, StreamCorruptedException;
- Object read method:
  - public Object readObject();
- Plus the primitive data methods listed in DataInputStream

java.io.ObjectOutputStream

Useful for writing Serializable Objects as well as Java primitive types.

- extends OutputStream
- implements ObjectOutput
- constructor
  - public ObjectOutputStream( OutputStream out ) throws IOException;
- Object write method:
  - public void.writeObject( Object o );
- Plus the primitive data methods listed in DataOutputStream

Other Useful Streams

java.io.PrintWriter

This is the simplest class for writing. It can write any type, with or without a line terminator following. None of its methods throw exceptions. System.out and System.err are PrintStreams.

- extends FilterOutputStream
- methods to print everthing:
interactive programming in java

- public void print( boolean b );
- public void print( char c );
- public void print( int i );
- public void print( long l );
- public void print( float f );
- public void print( double d );
- public void print( String s );
- public void print( Object obj );

- method to end the line:
  - public void println();

- plus a println method identical to each print method, above.

- and flush(), close(), etc. inherited from OutputStream

**java.io.PrintWriter**

Unfortunately, you can't make a PrintStream. PrintWriter is similar.

- extends Writer
- constructors
  - public PrintWriter( Writer out );
  - public PrintWriter( Writer out, boolean flushOnPrintln );
  - public PrintWriter( OutputStream out );
  - public PrintWriter( OutputStream out, boolean flushOnPrintln );

- The additional methods of a PrintWriter are as listed above for PrintStream

**java.io.LineNumberReader**

In case you want to know what number line you're reading.

- extends BufferedReader
- constructor
  - public LineNumberReader( Reader in );
- additional methods:
  - public int getLineNumber();
  - public void setLineNumber();

**java.io.SequenceInputStream**

This is useful when you want to pick up reading from one stream as soon as another one runs out of input.

- extends InputStream
• constructors
  • public SequenceInputStream( InputStream s1, InputStream s2 );
  • public SequenceInputStream( Enumeration e );

**IOExceptions**

The following classes are defined in the package java.io, and each extends java.io.IOException.

• CharConversionException
• EOFException
• FileNotFoundException
• InterruptedIOException
• InvalidClassException
• InvalidObjectException
• NotActiveException
• NotSerializableException
• ObjectStreamException
• OptionalDataException
• StreamCorruptedException
• SyncFailedException
• UnsupportedEncodingException
• UTFDataFormatException
• WriteAbortedException

The following additional IOExceptions are defined in the package java.net

• BindException
• ConnectException
• MalformedURLException
• NoRouteToHostException
• ProtocolException
• SocketException
• UnknownHostException
• UnknownServiceException

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and
formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>
Java Charts

About Java Charts

The tables in this file give literal specifications for different Java constructs. For example, the Program File Chart lists exactly what things can be in a .java file, in what order, and with what syntax. If your code doesn't match these specifications, it is not legal Java and will not compile.¹

See also Java Rules.

Contents

- About Java Charts
- Program File
- Interface Declaration
- Class Declaration
- Field Declaration
- Method Declaration
- Expressions
- Statements
- Disclaimers, Notes, Amendments, etc.

© 1999 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook from Morgan Kaufmann Publishers, Inc. It is a part of the course materials developed as a part of Lynn Andrea Stein’s Rethinking CS101 project at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments: <cs101-webmaster@ai.mit.edu>

Glossary

A

abstract method
   A method with no body; a method signature followed by a semicolon.

abstraction
   The use of a general contract as a placeholder for specific details. A technique for separating use from implementation.

alternative
   In a conditional statement, the optional sub-statement to be executed if the boolean test expression's value is false.

ampersand
   & Used in conjunction.

applet
   A Java program capable of running embedded in a web browser or other specialized applet environment. Contrast application.

application
   A Java program capable of running "standalone". Contrast applet.

animacy
   A Java Thread that enables concurrent execution, e.g., of a self-animating object. See the chapter on Self-Animating Objects.

animate object
   See self-animating object.

array
   A structure for holding many Things of the same type.

argument
   A value supplied to a method when it is invoked. During the execution of the method body, this value is named by the matching method parameter.

arithmetic operator
   An operator that computes one of the arithmetic functions. See the chapter on Expressions.

assertion
   A statement of what must be true. In Java, assertions are generally found in comments.

assignment
   The association of a name with a value. See the chapter on Things, Types, and Names.

Also the operator in such an assignment. See the chapter on Expressions.
assumption
Something taken for granted by a computer **program** or its **designer**. Often a **requirement** on the environment in which a program operates. When an assumption is violated, the program may not behave properly, so it is especially important for a **software engineer** to **document** the assumptions of the program.

asterisk
* Sometimes called **star**. Used as the multiplication **operator** and, together with a **slash**, to delineate certain **comments**.

B

backslash
\ Used in **character escapes**.

bang
See **exclamation point**. Also a loud noise, often made by a **child**.

base type
In an **array**, what (type) it is an array **of**.

base case
In a **recursive definition**, the case that does not rely on the thing being defined. Contrast **recursive case**.

batch
Things that happen while you wait....and wait....and wait. Or, better yet, things that you leave behind and pick up later. Contrast **real time**.

binary operator
An **operator** that takes two **operands**. See the chapter on **Expressions**.

binding
See **name binding**.

bit
A single binary digit.

bitwise operator
An **operator** that computes a **bit**-by-bit function such as bitwise complement. See the chapter on **Expressions**.

block
A sequence of **statements** contained between **braces**. See the section on **Blocks** in the chapter on **Statements**.

blocking
Waiting for information to become available, especially in a **read**.

body
The body of a **method**, **class**, or **interface**, i.e., either a **method body**, a **class body**, or an **interface body**.

boolean
A true-or-false **value**. In Java, represented by the **primitive type** **boolean** and by the **object type** **Boolean**. See the sidebar on **Java Primitive Types** in the chapter on **Things, Types, and Names**.

boolean expression
An **expression** whose **type** is **boolean**.
boot, boot up
   Start up (a computer or program).
bottom up, bottom-up design
   An approach to design that starts by identifying the simplest, most concrete things in your system and proceeds by combining them. Contrast top down.
brace
   { or } Used to enclose bodies or blocks.
bracket
   [ or ] Used in array expressions.
bug
   An error in a program. Contrast feature.

C
call
   See invocation.
call path
   The sequence of method invocation (instructions followed by a Thread) that led up to the currently executing method body. Unless execution exits abruptly, each of these invocations will return, one at a time, in this order, along the reverse of the call path, i.e., the return path.
carriage return
   One of two line-ending characters. (The other is line feed.) So named after an archaic device called a typewriter in whose early models the carriage (i.e., paper-bearing part) literally needed to be returned to the other side of the typewriter at the end of each line.
case-sensitive
   Distinguishing between upper and lower case letters.
cast expression
   An expression involving a type and an operand whose value is the same as its operand but whose type is the type supplied. Contrast coercion.
catastrophic failure
   An exceptional circumstance so incapacitating that your program cannot hope to prevent or deal with it. At this point, the only hope is in recovery.
catch statement
   A particular kind of Java statement, typically used with exceptions, that receives a thrown object. See the chapter on Exceptions.
character
   A single letter, digit, piece of punctuation, or piece of white space. In Java, represented by the primitive type char, using Unicode notation, and occupying sixteen bits, and by the object type Char. See the sidebar on Java Primitive Types in the chapter on Things, Types, and Names.
character escape
   A special sequence indicating a character other than by typing it directly. Especially useful for non-printing characters, such as carriage return.
child
   Offspring, inheritee, extender. The opposite of a parent.
A (user-definable) **type** from which new objects can be made. See the chapter on [Classes and Objects](#).

class body
The portion of a class **definition** containing the class's **members**. The portion of a class definition enclosed by `{ }`. See the chapter on [Classes and Objects](#) and the [Java Chart on Classes](#).

class object
The **object** representing the **class** itself, i.e., the factory. Itself an **instance** of class `java.lang.Class`.

client
With respect to some **service**, the (computational) **entity** that needs that service. Contrast **server**.

client pull
A communication pattern in which the **client** initiates the **service**. Contrast **server push**.

code
An excerpt from a **program**. Formally, source code is compiled (typically into executable code).

coefficient
Treating an **object** of one **type** as though it were of another type. Contrast **cast**. See the chapter on [Expressions](#).

colon
: Used in the ternary conditional **expression** and after **labels**.

column
Text embedded in a program in such a way that the Java compiler ignores it. Intended to make it easier for people to read and understand the code.

comparator
An **operator** in an **expression** of **boolean** **type**.

completeness
A promise made by a system that it will supply all true (or relevant) information. Trivially (and not very interestingly) accomplished by providing all information, whether true or not. Contrast **soundness**.

component
A **member**, especially a **field**.

compiler
The utility that transforms your Java code into something that can be run on a Java **virtual machine**.

compile time
The time at which a **program** is **compiled**. Compile time information refers to information that is available by reading the (partial) **source code** of the **program**. Contrast **run time**, information available when the program is actually **executing**.

compound assignment
A shorthand **assignment operator** (or **expression**) that also involves an **arithmetic** or **logical** operation.

concatenation
The gluing together of two **strings**.
In a **conditional statement**, the **boolean expression** whose **value** governs whether the **consequent** or the **alternative** is executed.

**conditional**

A compound **statement** whose execution depends on the **evaluation** of a **boolean expression**. Consists of a **condition**, a **consequent**, and an optional **alternative**. In Java, often an `if` statement. Contrast **sequence**, **loop**.

**conjunction**

The **logical operator** `&&` (and).

**concurrent**

Literally or conceptually at the same time. Contrast **sequential**.

**consequent**

In a **conditional statement**, the sub-statement to be executed if the **boolean** test **expression**'s value is true.

**console**

See [Java console](http://www.cs101.org/ipij/glossary.html).

**constant**

A **name** associated with an unchanging **value**. Typically declared `final`.

**constant expression**

Any **expression** whose **value** can be determined at **compile time**, i.e., independent of any **execution** of any **Threads** of any **program**. Typically either a **literal** or a **name** declared `final`.

**constructor**

The code which specifies how to make an **instance** of a **class**. Its name matches the name of the class. A constructor is a class **member**. See the chapter on [Classes and Objects](http://www.cs101.org/ipij/glossary.html).

**controller**

(In a **GUI**) How the **view** is connected to the **model**. In java.awt, this is not usually a separate object.

**D**

**data**

**Values**, as opposed to executable code. Things that might be associated with **names** such as **variables**, **parameters**, or **fields**. See also **state**.

**data repository**

A kind of **object** whose primary purpose is to store **data**. See the chapter on [Designing with Objects](http://www.cs101.org/ipij/glossary.html).

**debug**

To attempt to eliminate **bugs** from your program.

**declaration**

A **statement** associating a **name** with a **type**. Once the name has been declared, it can be used to refer to Things of the associated type. See the chapter on [Things, Types, and Names](http://www.cs101.org/ipij/glossary.html).

**declarative programming language**

A **programming language** based on **declarations** and **assertions**, i.e., **statements**, generally containing **variables**, about what must be true. A declarative programming language has rules of **execution** that calculate **values** for which these statements hold. All computation in a declarative programming language is **implicit**. Contrast **functional**, **imperative**, and **object oriented** programming languages.
default value

The value associated with a name that has been declared but not assigned an initial value. See the sidebar on Default Initialization in the chapter on Things, Types, and Names.

default visibility

Also called package visibility. The visibility level of an unmodified interface, class, method, field, or constructor. Visible to only within the package.

definition

A statement that both declares and initializes a name. See the chapter on Things, Types, and Names.

delegation

See event delegation.

design

The process of figuring out what your program should do and how it should accomplish it.

design for modifiability

The design principle that says that you should build software that is easy to maintain and adapt. See software lifecycle.

designer

A software engineer while working on the design of a program.

dial name

A name capable of referring to something of a primitive type, whose value is encoded directly in the memory reserved by the name. The types named by dial names are formally called value types. See the chapter on Things, Types, and Names.

disk server

A computer that provides (access to) disk storage for other computers on a network.

dispatch

A control-flow management technique in which you decide how to respond by considering the value that you have been asked to respond to (as opposed, e.g., to other environmental factors).

dispatch on case

A situation in which the decision of what action to take depends on which of a set of known values matches the value of a particular expression. A special case of dispatch. In Java, often implemented with a switch statement.

disjunction

The logical operator || (or).

dot

See period.

double precision floating point

A representation for rational numbers (and an approximation for real numbers) that uses 64 bits of storage. In Java, implemented by the primitive type double. See floating point.

down cast

A cast from superclass to subclass. May be invalid; should be guarded.
The property of being in an **environment** (or system) and interacting with it.

**encapsulation**
Packaging up of specific details into a single manipulable unit, often one that hides these (implementation) details from the **user**.

**entity**
A member of the community. A conceptual unit consisting of an **object** or set of objects that is (implicitly or explicitly) **persistent** and that **interacts** with other entities.

**environment**
Where an **entity** is **embedded**. What the entity interacts with.

**error checking**
Code (often a conditional statement) designed to catch illegal values or other potential problems, and to correct them, before or as they arise. A way to avoid **bugs** in your **program**. An important part of design.

**evaluate**
To compute the **value** of an **expression**.

**event**
1. Something that happens.

2. A special kind of **object** used in **event-driven programming** to record the occurrence of a particular event (in the conventional sense). See the chapters on Event-Driven Programming and Event Delegation.

**event-driven programming**
A style of programming in which an implicit (often, system-provided) control loop activates **event handler methods** when a relevant **event** occurs. See the chapters on Event-Driven Programming and Event Delegation.

**event delegation**
The system by which a separate listener object provides **event handlers** for another (GUI Component) object. Used in Java AWT versions 1.1 and later. See the chapter on Event Delegation.

**event handler**
In **event-driven programming**, a method that is called when a relevant **event** occurs. See the chapters on Event-Driven Programming and Event Delegation.

**exclamation point**
! Used in **boolean negation**.

**exception**
A special kind of Java **object** used to indicate an exceptional circumstance. Typically used in conjunction with **throw** and **catch** statements. See the chapter on Exceptions.

**execute**
To follow the **instructions** corresponding to a ** statement** or ** program**.

**exit condition**
The condition under which the repeated execution of a **loop** stops. Formally called the termination condition for the loop.

**explicit**
What I'm telling you. Contrast **implicit**.
expression

A piece of Java code with a type and a value, capable of being evaluated. Contrast statement. See the chapter on Expressions.

extend

To reuse the implementation supplied by a superclass (or, for interfaces, a parent interface) through inheritance.

F

factory

A class, metaphorically, for its instances.

feature

1. A deliberately designed and generally beneficial aspect of a program.
2. post hoc. A bug, when discovered by a user after it's too late to fix it.

field

A data member of a class, i.e., a name associated with each instance of a class (if not static) or with the class object itself (if static). See the chapter on Classes and Objects.

field access

An expression requiring an object and a field name. Its type is the declared type of the field and whose value is the value currently associated with that field.

final

A modifier indicating
1. that the value associated with a name, once assigned, cannot be changed, or
2. that a method cannot be overridden in a subclass, or
3. that a class cannot be extended.

file

A collection of information stored as a single unit on a persistent storage medium such as a computer disk.

file server

A computer that provides (access to) a set of files for other computers on a network.

floating point

A representation for rational numbers (and an approximation for real numbers) that uses 32 bits of storage. In Java, implemented by the primitive type float. Contrast double precision floating point.

flow of control

The sequence of instructions executed. Certain statements (such as conditionals and loops) modify the flow of control. Also called control flow.

footprint

See method footprint.
function
A method, especially one with a return value.

functional programming language
A programming language based on expressions rather than statements, i.e., in which (almost) everything has a value. Functional programming languages minimize the role of assignment. Contrast object oriented, functional, and imperative programming languages.

functional recursion
A form of recursion in which a function—or Java method—is defined recursively. See recursive function. Sometimes called procedural recursion, especially when the recursive method has no return value. Contrast structural recursion.

G
getter, getter method
A method that exists solely to provide read access to a field. Formally called a selector.

global variable
A term with no meaning in Java.

grandparent
A parent’s parent. Who to call when the parent falls through.

graphical user interface
A user interface that makes use of windows, icons, mouse, etc., and is typically implemented in an event-driven style. Sometimes abbreviated GUI.

guarantees
See promises.

guard expression
A test that prevents execution of a potentially dangerous statement.

GUI
An acronym for graphical user interface.

H
hyphen
- Used as the unary and binary subtraction operator and to indicate negative numbers.

I
identifier
The formal term for a name.

idiot proofing
A not very tactful name for error checking, especially as concerns interaction with the user.

if, if/else
Java’s conditional statement.

imperative programming language
A programming language based on statements rather than on objects. The major units of imperative programming languages are procedures, typically without return values. (These are roughly void)
methods, but without encapsulating objects.) Contrast object oriented, imperative, and declarative programming languages.

implement

What an implementor does. More specifically, what a class does with an interface. Also what a programmer does.

implementor

The 1. person or 2. entity that provides the implementation for an interface or contract. Contrast user.

implementation

Executable code. Also "how to". Contrast use.

implicit

What I'm not telling you, but is so anyway. Contrast explicit.

incremental program design

The design-build-test-design cycle in which every attempt is made to keep the program working at all times and to make only minor modifications between tests.

index

An expression, typically with an integer value, used to select a member from a (generally uniform) set.

inherit

What a child does with a parent. Specifically, what a subclass does with a superclass.

inheritance

The process by which one class shares the definition and implementation provided by another. Also the process by which one interface extends another. Uses the Java keyword extends. See the chapter on Inheritance.

initialization

The assignment of an initial value to a name or, by extension, to an object's fields.

input

Information that is read by a program or entity; or, the stream or other resource from which input is read.

instance

An object created from a class, whose type is that class. See the chapter on Classes and Objects.

instanciate

To create an instance from a class, typically through the use of a constructor (and new).

instruction follower

The thing that executes statements. In Java, a Thread.

instructions

Code, generally statements, explaining how to do something. Followed step by step by an instruction follower.

integer type
In Java, one of `byte`, `short`, `int`, `long`, `char`, or `boolean`. Expressions of these (and only these) types may be used as the test expression of a `switch` statement.

**interface**

1. The common region of contact between two or more `entities`.

2. (Java) A formal statement of method signatures and constants defining a `type` and constraining the behavior of objects implementing that interface.

See the chapter on Interfaces.

**interface body**

The portion of an interface `definition` containing the interface's members. The portion of an interface definition enclosed by `{ }`.

**interaction**

Literally, *action between*. How two (or more) things get along.

**invocation**

To call a `method`, i.e., execute its `body`, passing `arguments` to be associated with the method's parameters.

**J**

**Java console**

A place in every Java environment from which `standard input` is read and to which `standard output` is written. I/O to the Java console is provided by `cs101.util.Console`, `java.lang.System.in`, and `java.lang.System.out`.

**jelly**

An exceedingly sticky concoction made from the juice of a fruit, often a grape, ideally purple. See also peanut butter.

**just going around in circles**

What happens in a `recursion` without a `base case`.

**K**

**keyword**

A word with special meaning in Java. All Java keywords are `reserved`, i.e., cannot be used as Java names.

**L**

**label**

A `no-op` marker `statement` that simply indicates its location in `code` for later reference, e.g., by a `break` or `continue` statement. Not related to the more common `label name`.

**label name**

A `name` capable of referring to something of an `object type`, i.e., anything not of a `primitive type`. See the chapter on Things, Types, and Names.

**layered**

Especially layered service. A `service` that is accomplished through reliance on another service.
One who reads.

left-hand side

In an assignment, the expression representing the shoebox or label to which the value is assigned.

legacy

Old, often out-of date. Typical in "legacy software", a piece of software that an organization continues to use (and that a software engineer must therefore adapt, integrate, or interface with) long beyond its desirable lifetime.

literal

A Java expression to be read literally, i.e., at face value. Only the primitive types plus strings have corresponding literal expressions. See the sidebar on Java Primitive Types in the chapter on Things, Types, and Names.

local

Another term for a variable. Short for local variable.

local variable

The formal term for a variable.

logical operator

An operator that computes an arithmetic function such as conjunction or disjunction. See the chapter on Expressions.

loop

A construct by which a sequence of statements is executed repeatedly, typically until some exit condition is met. Contrast sequence, conditional.

lossy

Losing information. For example, a narrowing coercion may be lossy because the thing being coerced may be too big to fit into the new type.

M

magic number

A literal number appearing without explanation or obvious meaning in your code. It is generally better style to use a constant, i.e., a final name.

mail server

A computer that provides (electronic) mail service for other computers on a network.

maintain

To continually test, debug, and modify a program so as to fix bugs and otherwise ensure that it continues to work reliably (e.g., in the face of changes to its environment, to its requirements, or to the underlying system).

member

A constructor, field, or method of a class. Alternately, a (static) field or (abstract) method of an interface. Also member (inner) classes or interfaces. See the chapter on Classes and Objects.

method

An executable class member. Consists of a signature plus a body (unless abstract). When a method is invoked on an argument list, the body is executed with each of the method's parameter names bound to its corresponding argument.
method body
The portion of a method that contains executable statements. When a method is invoked (on a list of arguments), its body is executed within the scope of the parameter bindings, i.e., with the parameter names bound to the corresponding arguments.

method footprint
The name plus the ordered list of parameter types of a method. An object may have at most one method with any particular footprint. Contrast method signature. See the chapter on Interfaces.

method invocation
See invocation.

method overriding
When a subclass redefines a method or field that would otherwise be inherited from its superclass.

method overloading
When one object has two or more methods with the same name (but different footprints), typically performing different functions.

method signature
The specification of a method's name, ordered list of parameter types, return type, and exceptions, possibly including modifiers. Contrast method footprint. See the chapter on Interfaces.

model
(In a GUI.) An object implementing how the mechanism works, i.e., what it does within your program. Contrast view.

modifier
A formal Java term such as abstract, final, public, static, synchronized, etc., which is used in the definition of a class, interface, or member. See the Java Charts for details.

mutator
The formal name for a setter method.

N

name
A Java expression that refers to a particular object or value. Examples include variables, parameters, fields, class names, and interface names. Every name has an associated type (fixed when the name is declared). Within its scope, the name is generally bound to a value (of the appropriate type). See the chapter on Things, Types, and Names.

name binding
The association of a name with a value, typically through assignment or through parameter binding during method invocation. The details of this association depend on whether the name is a shoebox name or a label name, i.e., of primitive or object type.

narrowing, narrowing coercion
Treating a thing of one type as though it were of another, smaller, less precise type. Includes coercion to a smaller primitive type (e.g., long to int) as well as coercion to a superclass (or super-interface) type.

natural language
What humans speak (before they become geeks).

negation
Not. Of a boolean, the other one.

network
A set of interconnections, often between computers.
no-args  Taking no arguments or, more properly, having no parameters.

no-op  Having no effect, like talking to a wall or shouting into the wind.

null  A Java keyword. The non-value with which an unbound label name is associated.

null character  The character with unicode number 0. Not to be confused with the non-value null.

O

object  A non-primitive, non-null Java Thing. An instance of (a subclass of) java.lang.Object.

object oriented programming language  A programming language based on objects. Contrast functional, imperative, and declarative programming languages.

object type  In Java, any type other than one of the eight primitive types. All object types are named by label names.

operand  One sub-expression of an operator expression. See the chapter on Expressions.

operator  The part of an operator expression that determines the particular relationship of the operands to the expression's value. See the chapter on Expressions.

operator expression  An expression involving an operator (e.g., +) and one or more operands. Typically, the value of the expression is a particular function of the operands, with the operator specifying what function. See the chapter on Expressions.

overriding  See method overriding.

overloading  See method overloading.

output  The information that is written by a program or entity; or, the stream or other resource to which it is written.

P

package  1. A named group of Java interface and class definitions.

  2. The default visibility level of an unmodified interface, class, method, field, or constructor. Visible only within the package(1).
An archaic but amazingly persistent storage medium made of wood pulp. Reported continually over the last half-century to be destined for imminent obsolescence with the incipient advent of the paperless office. Sometimes used with a typewriter.

**parameter**

A name whose scope is a single invocation of the method to which it belongs. Declared in the method signature. When the method is invoked on a list of arguments, each parameter is bound to the corresponding argument prior to (and with scope over) the execution of the method body.

**parameter binding**

The form of name binding that occurs when a method is invoked on a list of arguments. Each of the method's parameters is bound to the corresponding argument, i.e., the first parameter to the first argument, etc.

**parent, parent type**

A generic term encompassing superclass, interface implemented, or interface extended. Also, the common enemy uniting child and grandparent.

**peanut butter**

A gooey brown paste made by grinding up a certain legume, often consumed with jelly between two slices of very bland white bread.

**period**

. Sometimes also called dot. Used in method invocation and field access expressions, package naming, and as a decimal point.

**persistent**

Existing even when not currently the subject of the coder's, computer's, or instruction-follower's attention.

**pipe**

See vertical bar.

**pointer**

A term with no meaning in Java.

**polymorphism**

Behaving differently with different types of things.

**port**

To translate a piece of software from one programming language to another or from one kind of computer system to another.

**postcondition**

What is true after something has happened. Typically indicates something that has changed. Contrast precondition.

**postfix**

Coming after.

**precondition**

What must be true before something can happen. Contrast postcondition.

**predicate**

An expression or method whose (return) value is of type boolean.

**prefix**

Prior to.
primitive type
In Java, one of `byte`, `short`, `int`, `long`, `float`, `double`, `char`, or `boolean`. All primitive types are named by shoebox names. See the sidebar on Java Primitive Types in the chapter on Things, Types, and Names.

private
A Java keyword. A class or interface member declared private is visible only within the body of its defining class or interface.

procedural abstraction
Combining a group of instructions into a single named unit (a procedure; in Java, a method) so that it can be reused.

procedural recursion
See functional recursion.

procedure
Something that is done. In Java, a method, especially (but not exclusively) one without a return value.

program
n. A collection of executable code. The how-to instructions that a computer follows.
v. To compose a program. See also incremental program design, debug.

programmer
A person who develops (designs, writes, debugs, modifies) a program.

programming language
A language in which one writes a program. For the purposes of this book, Java.

programming environment
A set of on-line tools in which a programmer develops (designs, writes, debugs, modifies) a program. Typically includes (at least) an editor, compiler, runtime environment, and debugger.

promises
Commitments a program (or its designer) makes about that program's behavior. Some promises are embodied in the program's interface.

protected
A Java keyword. A class or interface member declared protected is visible within its package and within any class (or interface) that extends (or implements) its containing class (or interface).

prototype
A simple, often hastily thrown together version of a program (or other product) intended to help in the design process. A prototype is not intended for serious use and may lack the features or complexity of the final, production version.

public
A Java keyword. An interface, class, method, field, or constructor declared public is visible everywhere.

Q
query
A specialized request to a program or other entity, in which some information is provided and other, matching information is to be returned by the program in response.

**R**

read, read access
Interacting with a name by obtaining its associated value, or with an object by reading the value(s) of one or more of its fields, or with an input stream or other resource by obtaining the next value from it.

real time
When things happen. On a human time scale (or faster). Contrast batch.

recipe
The instructions for how to do something. A class is a recipe for the behavior of its instances. A constructor is the recipe for how to make an instance of its class.

recovery
Also recovering from error. What a program ought to do after something has gone wrong; patch things up as well as possible and move on. If things are disastrous enough (e.g., after a catastrophic failure), this can be a significant task. It is facilitated by design that anticipates the need for eventual recovery.

recursion
The use of recursive definitions to accomplish real things. Contrast just going around in circles.

recursive call
The recursive case of a recursive function.

recursive case
In a recursive definition, the part that relies on (a simpler form of) the thing being defined. Contrast base case.

recursive definition
A definition in terms of itself. That is, the definition uses the thing being defined. Consists of one or more base cases and at least one recursive case. See recursive function, recursive structure.

recursive function
A method that is defined recursively. That is, the implementation of the method contains an invocation of the same method with simpler argument(s). See recursive definition.

recursive structure
In Java, a class whose instances contain members of the same type as the instances themselves. That is, the class defines one or more non-static fields whose type is the same as the type of the class. See recursive definition.

reference type
The formal term for the types named by a label name.

requirements
Expectations that a program must meet. Often identified by a designer using a technique such as use cases and embodied in a program's promises or guarantees.

reserved word
A word that cannot be used as an identifier in Java, typically because it is a keyword.

resource library
A class that exists to hold methods that don't logically belong to any particular object, or other (typically system-wide) resources. Typically not an instantiable class. See the chapter on Designing with Objects.
return

A **statement** whose **execution** causes normal termination of the execution of a **method body**. If the return statement contains an **expression**, its **type** must match the **return type** of the **method**. In this case, the expression is **evaluated** prior to exiting the method body and the **value** of this expression is the **return value** of the **method invocation**.

return path

See **call path**.

return type

The **type** of the **value** returned by a **method invocation**. The first item in a **method declaration**.

return value

The **value** returned by a **method invocation**.

rule

A **proto-method**. Consists of a **specification** and a **body**. See the chapter on **Statements and Rules**.

rule body

The set of **statements** detailing how a **rule** is to be accomplished. A **proto-method body**. See the chapter on **Statements and Rules**.

rule specification

The information needed and provided by a **rule**. A **proto-signature**. See the chapter on **Statements and Rules**.

run time

The time at which a **program** is **executed**. Run time information refers to information that is not known until the program is executed, i.e., cannot be determined from the **source code** alone. Contrast **compile time**.

Runnable

**Executable**, especially (Runnable) by a **Thread**.

S

scope

The expanse of code within which a name has meaning, i.e., is a valid expression. See the note on **Scoping** in the chapter on **Expressions**. Not quite.

scribe

One who **writes**.

selector

The formal name for a **getter** method.

self-animating object

An **object** or **entity** with its own **animacy**, i.e., one that runs **concurrently** and **persistently**. See the chapter on **Self-Animating Objects**.

semantics

The rules defining what **expressions** and **statements** in a **language** mean (or what they do). Contrast **syntax**.

semicolon

**;** Used to end a simple statement.

sequence

Two or more **statements** to be executed one after the other, in order. Contrast **conditional, loop**.

sequential

One after another; one at a time; in order. Contrast **concurrent**.
server
With respect to some service, the (computational) entity that provides that service. Contrast client.

server push
A communication pattern in which the server initiates the service. Contrast client pull.

setter, setter method
A method that exists solely to provide write access to a field, i.e., to change its value. Formally called a mutator.

shared reference
A situation in which two label names refer to the same object.

shoebox name
Also dial name. A name capable of referring to something of a primitive type, whose value is encoded directly in the memory reserved by the name. The types named by shoebox names are formally called value types. See the chapter on Things, Types, and Names.

side effect
A change to something that occurs as a consequence of evaluating an expression. For example, an assignment.

signature
See method signature.

slash
/ Used to delineate comments and as the division operator.

software
Another term for computer program.

software engineer
A person who designs, builds, tests, debugs, and maintains computer programs.

software lifecycle
The stages of a computer program's life: design, build, test, and then maintain/adapt/modify for a very long time. Note that the vast majority of the software lifecycle is spent in the final phase(s).

soundness
A promise made by a system that all information that it supplies is true (or relevant). Trivially (and not very interestingly) accomplished by providing no information at all. Contrast completeness.

source code
See code.

standard input
The stream which reads from the Java console. Bound to java.lang.System.in.

standard output
The stream which writes to the Java console. Bound to java.lang.System.out.

state
What is true of a program or entity at a specific time. Especially the current set of associations of values with names.

statement
A piece of executable Java code. Has neither type nor value. Contrast expression. See the chapter on Statements and Rules.
**static**

A **modifier** indicating a member of a **class** (rather than of its **instances**).

**stream**

A **persistent** Java **object** which permits the **reading** or **writing** of multiple sequential values. Represents a connection to another (potentially non-Java) entity. Used for **input** or **output**.

**string**

A sequence of characters. In Java, represented by the **object type** **String**. Although there is no **primitive type** representation of strings in Java, they are described in the sidebar on **Java Primitive Types** in the chapter on **Things, Types, and Names**. Also, what's strung taut between two **tin cans**, carrying the sound by vibrating, in a **tin can telephone**.

**structural recursion**

A form of **recursion** in which a **class** is **defined** recursively. See **recursive structure**. Contrast **functional recursion**.

**style**

What we all wish we had.

**subclass**

A **class** that **inherits** from another, i.e., **extends** that other. Contrast **superclass**. See the chapter on **Inheritance**.

**superclass**

A **class** that is **inherited** from by another, i.e., the other **extends** the superclass. Contrast **subclass**. See the chapter on **Inheritance**.

**symbolic constant**

A **name**, associated with an unchanging but meaningless **value**. Used when the uniqueness and consistency of the value are important, but the particular value is not. See the chapter on **Dispatch**.

**syntax**

The rules defining what is legal in a certain **language**. Where to put the **semicolons**. Contrast **semantics**.

**T**

**target**

In a **method invocation expression**, the **object** whose **method** it is.

**termination condition**

The formal name for an **exit condition**.

**test**

1. A crucial part of **program** development in which program behavior is exercised in an attempt to find failures, or **bugs**.

2. In a **conditional** statement, another name for the boolean expression known as the **condition**.

**Thing**

The nouns of Java, including Things of **primitive type** and **objects**. See the chapter on **Things, Types, and Names**.

**this**

A Java (label) **name** that is **bound** to the current **instance**. Because it refers to an instance, **static members** are outside of its **scope**.
Thread
A Java instruction follower.

throw statement
A particular kind of Java statement, typically used with exceptions, that causes an object to be thrown and thereby circumvents the typical return trajectory. See the chapter on Exceptions.

throws clause
The part of a method signature which specifies any exceptions thrown by that method. See the chapter on Exceptions.

tin can
What one person talks into, and another listens on, in a tin can telephone. Corresponds to a socket.

Also a container for food, though neither peanut butter nor jelly. Said food must be eliminated before construction of the telephone.

tin can telephone
A device consisting of two tin cans, empty of food and with one end of each removed completely, and a string strung taut between a hole punched in the intact ends of each of the cans. Communication is accomplished when one person speaks into one tin can and another person listens at the other.

top down, top-down design
An approach to design that starts by identifying the highest level, most abstract, or largest things in your system and proceeds by decomposing them. Contrast bottom up.

top level
Immediately inside the containing structure. Top level within a class means inside the class body but not inside any other structure.

trinary operator
An operator that takes three operands. (Also ternary operator.) See the chapter on Expressions.

type
A partial specification of the Thing. In Java, a type is either a primitive type or an object type. See the chapter on Things, Types, and Names.

type-of-thing name-of-thing rule
The rule that says: to declare a name, first state its type, then state its name.

typewriter
An archaic device vaguely resembling a keyboard attached directly to a printer with no intervening memory. Requires paper.

U

unary operator
An operator that takes one operand. See the chapter on Expressions.

unbound
The state of a label name when it is not associated with an object, i.e., has no object referent. In this case, the label name is associated with the non-value null.

Unicode
The representation used by Java for characters.
up cast
   A cast or coercion from subclass to superclass. Always valid.

use
   n. The incorporation of a resource into a program. Contrast implementation.

use case
   A description of a single interaction between a user and an entity. A technique used by a designer to identify requirements on the entity. Includes preconditions and postconditions. @@

user
   1. A human being, with respect to a computer program.
   2. A piece of code, with respect to another piece of code, especially an interface. Contrast implementor.

user interface
   The portion of a program with which a (human) user interacts. See also graphical user interface.

V

value
   Either a primitive value or an object.

class
   A Java name that has scope only from its declaration to the end of the enclosing block. Variables are formally called local variables; sometimes, this is abbreviated to locals.

vertical bar
   | Also called pipe. Used in disjunction.

view
   (In a GUI.) An object implementing how the mechanism looks. In java.awt, this typically includes its basic on-screen behavior. Contrast model.

virtual field
   A piece of state within an object that is not stored directly as a field, but is instead calculated using the values of other fields of the object. Must be accessed using a getter method as there is no field to read directly.

virtual machine
   The utility that actually runs your (compiled) Java program.

visibility
   Whether a class, field, method, or constructor can be used by a particular piece of code. Visibility levels include private, protected, default (or package), and public.

void
   The return type of a method whose invocation does not return anything. Contrast null.

W

web server
   A computer that provides (access to) web pages for other computers on a network.
white bread
A substance resembling styrofoam, but with less taste and texture. Generally available in uniform white squares with pale brown edges, called crusts, that must be removed before serving to small children. Useful mostly to keep the peanut butter and jelly from getting on your fingers.

white space
Tabs, spaces, carriage returns, and other characters that are meant to be seen as empty space.

widening, widening coercion
Treating a thing of one type as though it were of another, larger, more precise type. Includes coercion to a larger primitive type (e.g., short to int) as well as coercion to a subclass (or super-interface) type.

write, write access
Interacting with a name by changing its associated value, or with an object by changing the value of one or more of its fields, or with an output stream or other resource by providing a value to it.

© 2003 Lynn Andrea Stein

This chapter is excerpted from a draft of Introduction to Interactive Programming In Java, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
webmaster@cs101.org