Outline

- **Methods**
  - Constructors & Destructor
  - Overloading

- **Control Flow**
  - If else, while, do while, for

- **I/O**
  - Java I/O System
  - InputStreams
  - FilterOutputStreams
  - Character I/O Streams
  - Modifying Stream Behavior
  - Sources & Sinks of Data
  - Modifying Stream Behavior
Constructor & Destructor

- Java guarantees proper initialization with constructors, helps cleanup with garbage collector
Guaranteed Initialization with the Constructor

class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}

class SimpleConstructor {
    public static void main(String args[]) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
}
Method Overloading

• One word, many meanings: *overloaded*

```java
class Tree {
    int height;
    Tree() {
        System.out.println("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        System.out.println("Creating new Tree that is "+ i + " feet tall");
        height = i;
    }
    void info() {
        System.out.println("Tree is " + height+ " feet tall");
    }
    void info(String s) {
        System.out.println(s + ": Tree is "+ height + " feet tall");
    }
}
```
Method Overloading

```java
import java.util.Random;

class Overloading {
    public static void main(String[] args) {
        int i = 0;
        while(i != 9) {
            Tree t = new Tree(i = new Random().nextInt(10));
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
}
```
Default Constructor: Takes no Arguments

- Compiler creates one for you if you write no constructors

```java
class Bird {
    int i;
}
public class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // Default!
    }
}
```
Constructor Initialization

• Order of initialization
  – Order that variables/objects are defined in class
• Static data initialization

```java
class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);
    // ...
}
```

*b4* only created on *first* access or when first object of class *Cupboard* is created
public class Leaf {
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
}
this: Specifying a Member

- If you get lazy when creating identifiers
- Probably not a good practice, but I do it myself sometimes…

```java
class Flower {
    String name;
    Flower(String name) {
        // Without "this" it would assign
        // the argument to itself:
        this.name = name;
    }
}
```
Destructor

- garbage collection
  - Garbage collection is not destruction
  - Your objects may not get garbage collected
  - Garbage collection is only about memory

- finalize( )
  - In theory: releasing memory that the GC wouldn’t
  - It’s never been reliable: promises to be called on system exit; (causes bug in Java file closing)

- You must perform cleanup
  - Must write specific cleanup method
Member Initialization

- primitives are given default values if you don’t specify values

```java
void f() {
    int i; // No initialization
    i++;
}

class Data {
    int i = 999;
    long l; // Defaults to zero
    // ...
}
```
Explicit static Initialization

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}
class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() { System.out.println("Cups()""); } 
}
Array Initialization

- Creates a reference, not the array. Can’t size it. To create an array of primitives:

```java
int[] a1 = { 1, 2, 3, 4, 5 };
```

- An array of class objects:

```java
Integer[] a = new Integer[20];
System.out.println("length of a = " + a.length);
for(int i = 0; i < a.length; i++) {
    a[i] = new Integer(i);
    System.out.println("a[" + i + "] = " + a[i]);
}
```
Array Initialization

• Can also use bracketed list (The size is then fixed at compile-time)

```java
Integer[] a = {
    new Integer(1),
    new Integer(2),
    new Integer(3),
};
```

• If you do anything wrong either the compiler will catch it or an exception will be thrown
Control Flow

• the keywords
  – if-else, while, do-while, for, and a selection statement called switch.
• Java does not support the much-maligned goto (which can still be the most expedient way to solve certain types of problems).
• You can still do a goto-like jump, but it is much more constrained than a typical goto.
Control Flow

• If else
  – The conditional must produce a **boolean** result.
  – **Form**
    
    ```
    if(Boolean-expression)
    Statement
    or
    if(Boolean-expression)
    statement
    else
    statement
    ```
Control Flow Cont’d

- **Iteration**
  - *while, do-while* and *for* control looping and are sometimes classified as *iteration statements*. A *statement* repeats until the controlling *Boolean-expression* evaluates to false.

  - The form for a *while* loop
    ```java
    while(Boolean-expression)
    Statement
    ```
  - The form for do-while is
    ```java
    do
    statement
    while(Boolean-expression);
    ```
Control Flow Cont’d

• Iteration cont’d
  – The form of the for loop is:
    for(initialization; Boolean-expression; step)
    statement

```java
public class WhileTest {
    public static void main(String[] args) {
        double r = 0;
        while(r < 0.99d) {
            r = Math.random();
            System.out.println(r);
        }
    }
}
///:~
```
The Java I/O System

• **Goal**
  – to provide abstractions of all aspects of I/O
    • Directory structure, File, Memory, Network, etc.

• **Expressing all possible configurations**
  – Character, binary, buffered, reading lines, transparent data transfer, etc.
The File class

- **Deceiving**
  - refers to one or more file names, not a handle to a file itself
    - *Composite* design pattern: to represent tree structured hierarchy (node and leaf)

- **Set of file names**
  - `list()` gives an array of **String**

- For a subset of file names, you hand `list()` an object that implements **FilenameFilter**
Example: Limiting the Number of Files Returned by the list() Method

• Use of String[] list(FileNameFilter FFObj);
  – \texttt{FFObj} is an object of a class that implements the \texttt{FileNameFilter} interface
  • Defining only a single method, \texttt{boolean accept(File directory, String filename)};
    – Returning true for files in the directory that should be included in the list

• \texttt{OnlyExt} class implementing \texttt{FileNameFilter}
  – Restricting the visibility of the filenames returned by \texttt{list()} to files with names that end in the file extension specified when the object is constructed
Example: Limiting the Number of Files Returned by the list() Method Cont’d

• OnlyExt class

```java
import java.io.*;
public class OnlyExt implements FilenameFilter {
    String ext;
    public OnlyExt(String ext) {
        this.ext = "." + ext;
    }
    public boolean accept(File dir, String name) {
        return name.endsWith(ext);
    }
}
```
Example: Limiting the Number of Files Returned by the list() Method Cont’d

- Displaying files that use the .html extension

```java
// Directory of .HTML files.
import java.io.*;
class DirListOnly {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = f1.list(only);
        for (int i=0; i < s.length; i++) {
            System.out.println(s[i]);
        }
    }
}
```
I/O Fundamentals

• Different kinds of I/O
  – Files, the console, blocks of memory, network connections

• Different kinds of operations
  – Sequential, random-access, binary, character, by lines, by words, etc.
Binary Input and Output

• **InputStream**
  – All have `read()` methods you won’t usually use
  – Sometimes tricky to tell when you’re at the end

• **OutputStream**
  – All have `write()` methods you won’t usually use

• Wrapping classes in “decorators” to add functionality. More work while coding.
Adding Attributes & Useful Interfaces

• Two issues with I/O streams:
  – What you’re talking to
  – The way you talk to it

• One approach
  – Making a class for every possible combination

• Alternative
  – Java’s “filter” streams (decorators)

• Dynamically creating the functionality you need
  – Input: `FilterInputStream`
  – Output: `FilterOutputStream`
Filter Input Streams

- **DataInputStream**
  - Full interface for reading primitive and builtin types

- **BufferedInputStream**
  - Adding buffering to the stream (usually do this)

- **LineNumberInputStream**
  - Adding line numbering functionality (nothing else; you’ll probably add another filter)

- **PushbackInputStream**
  - Implementing a one-character push back, for scanners. You probably won’t use this
Filter Output Streams

- **DataOutputStream**
  - Full interface for writing primitive and built-in types; complementing **DataInputStream** for portable reading & writing of data

- **PrintStream**
  - Allowing primitive formatting for data display

- **BufferedOutputStream**
  - Adding a buffer to the output stream (usually do this)
Character I/O Streams

- Added in Java 1.1
- Can appear that they are intended to replace `InputStream` and `OutputStream` and `Reader` and `Writer` classes
  - Internationalization: uses 16-bit `char` (capable of holding Unicodes) instead of 8-bit `byte`
  - Also designed to improve speed
- Classes with no Character Versions
  - `DataOutputStream`
  - `File`
  - `RandomAccessFile`
  - `SequenceInputStream`
## Sources & Sinks of Data

### Binary
- InputStream
- OutputStream
- FileInputStream
- FileOutputStream
- StringBufferInputStream
- (no corresponding class)
- ByteArrayInputStream
- ByteArrayOutputStream
- PipedInputStream
- PipedOutputStream

### Character
- Reader
  - converter: InputStreamReader
- Writer
  - converter: OutputStreamWriter
- FileReader
- FileWriter
- StringReader
- StringWriter
- CharArrayReader
- CharArrayWriter
- PipedReader
- PipedWriter
Modifying Stream Behavior

**Binary**
- FilterInputStream
- FilterOutputStream
- BufferedInputStream
- BufferedOutputStream
- DataInputStream
- PrintStream
- LineNumberInputStream
- StringTokenizer
- PushbackInputStream

**Character**
- FilterReader
- FilterWriter (abstract class with no subclasses)
- BufferedReader (also has readLine() )
- BufferedWriter
- Use DataInputStream (except when you must use readLine( ) , then use a BufferedReader)
- PrintWriter
- LineNumberReader
- StringTokenizer (Use constructor that takes a Reader instead)
- PushbackReader
File I/O Examples

- **FileInputStream**
  
  - Getting bytes from a file

```java
import java.io.*;
public class Read {
    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream("in.txt");
            int b;
            while ((b = f.read()) != -1)
                System.out.print((char) b);
        } catch (FileNotFoundException fnfe) {
            // System.out.println(fnfe);
            fnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        System.out.flush();
    }
}
```
File I/O Examples (Cont’d)

• FileOutputStream

  – Writing bytes to a file

```java
import java.io.*;
public class Write {
    public static void main(String[] args) {
        try {
            byte ova[] = {'o', 'u', 't', '\n'};
            FileOutputStream f = new FileOutputStream(args[0]);
            f.write(ova);
            f.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```