INTERFACES
EXCEPTION HANDLING

18TH LECTURE

엄현상 (Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University
Outline

- **Interfaces**
  - An Instrument interface
  - “Multiple Inheritance” in Java
  - Java “Multiple Inheritance”

- **Error Handling with Exceptions**
  - The problem
  - What’s an exception?
  - Basic Exception / Catching an Exception
  - The Exception Specification
  - Creating your own exceptions
  - Catching any Exception
  - Rethrowing an Exception
  - RuntimeException

  - One more factor: finally
  - What’s “finally” For?
  - Exceptions in Constructors
  - Exception Matching
  - Catching Base-Class Constructor Exceptions
  - “Inheritance” of Exceptions
  - Overhead
  - Guidelines
  - Summary
Interfaces

- Can’t have any fields or method definitions
An Instrument interface

- No “concrete” elements in interface
- You don’t extend, you implement

```java
import java.util.*;

interface Instrument {
    // Compile-time constant:
    int i = 5; // static & final
    // Cannot have method definitions:
    void play(); // Automatically public
    String what();
    void adjust();
}

class Wind implements Instrument {
    public void play() {
        System.out.println("Wind.play()");
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}
```
“Multiple Inheritance” in Java

- New class has combined interfaces of all types
  - But using only one physical implementation: that of the concrete base class
Java “Multiple Inheritance”

• To add extra interfaces
  – *Not* to combine implementations (using composition for that)
• Using it if you need to upcast to more than one base type
• Guideline
  – Using interfaces when possible, avoiding abstract classes
  – You never know when you’ll need to combine interfaces; any sort of concreteness prevents it
Error Handling with Exceptions

• Java
  – “Badly-formed code will not be run”
• Not all errors can be caught at compile time
• Run-time error handling integrated into the core of the language, enforced by the compiler
• Can’t get too far learning the language without it
The problem

• Coping with errors during program execution
• Errors can be caused by
  – Program logic
    • I.e., exceeding array bounds
    • Can be prevented by the programmer
  – Status of the environment
    • I.e., network goes down
      – Cannot be prevented by the programmer
What’s an exception?

• Exception
  – A type of object that signals an error condition and provides information about the error
• Once an exception is generated, control is passed up the call stack to a specific handler
  – You can have as many handlers as you want, for different exceptions and/or at different levels
• Java exceptions cannot be ignored
Basic Exception

• *Exceptional Condition*
  – not enough info in the current context to continue processing

• **throw** an exception:
  ```java
  if(t == null)
  throw new NullPointerException();
  ```

• Exception arguments
  ```java
  if(t == null)
  throw new NullPointerException("t=null");
  ```
  – Like any other constructor
  – Info can be extracted later
Catching an Exception

- **try** block
  - A guarded region
  
  ```java
  try {
    // Code that may generate exceptions
  } catch(Type1 id1) {
    // Handle exceptions of Type1
  } catch(Type2 id2) {
    // Handle exceptions of Type2
  } catch(Type3 id3) {
    // Handle exceptions of Type3
  }
  // etc...
  ```
The Exception Specification

```java
void f() throws TooBigException { //...
```

- If you say `void f() {}`
- It means that no exceptions (except for those derived from the special class `RuntimeException`) may be thrown
- Compiler verifies exception specifications!
- This guarantees that all (checked) exceptions will get caught somewhere
Creating your own exceptions

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

public class FullConstructors {
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (MyException e) {
            e.printStackTrace(System.err);
        }
        try {
            g();
        } catch (MyException e) {
            e.printStackTrace(System.err);
        }
    }
}
Catching any Exception

• All the exceptions you need to worry about
• Being derived from `Exception`
  ```java
  catch(Exception e) {
    System.out.println("Caught exception");
  }
  ```
• Special system errors are derived from `Error`
• Program bugs: `RuntimeException`
  – These are thrown automatically for run-time programming errors
Rethrowing an Exception

catch(Exception e) {
    System.out.println("Exception was thrown");
    throw e;
}

• Performing anything you can locally, then letting a global handler perform more appropriate activities
What’s in a name?

- Name of the exception is typically the most important thing about it
- Names tend to be long and descriptive
- Code for the exception class itself is usually minimal
- Once you catch the exception you are usually done with it
• Name is confusing, since every exception is thrown at runtime

• Base class for all errors generated by programming mistakes that appear at runtime
  – NullPointerException,
  – ArrayIndexOutOfBoundsException,
  – IllegalArgumentException, etc.

• Do not need to include RuntimeException classes in the exception specification
### One more factor: finally

- At least one catch or finally clause must be present

```java
try {
    // The guarded region: Dangerous activities
    // that might throw A, B, or C
} catch(A a1) {
    // Handler for situation A
} catch(B b1) {
    // Handler for situation B
} catch(C c1) {
    // Handler for situation C
} finally {
    // Activities that happen every time
}
```

- **Try block (mandatory)**
- **Catch clauses**
- **Finally clause**
What’s “finally” For?

- Always getting called, regardless of what happens with the exception and where it’s caught
- To set something other than memory back to its original state (GC handles memory) (close files, network connections, etc.)

```java
class Switch {
    boolean state = false;
    boolean read() { return state; }
    void on() { state = true; }
    void off() { state = false; }
}

public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            OnOffSwitch.f();
        } catch(OnOffException1 e) {
            System.err.println("OnOffException1");
        } catch(OnOffException2 e) {
            System.err.println("OnOffException2");
        } finally {
            sw.off();
        }
    }
}
```
class FourException extends Exception {}

public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println("Entering first try block");
        try {
            System.out.println("Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println("finally in 2nd try block");
            }
        } catch (FourException e) {
            System.err.println("Caught FourException in 1st try block");
        } finally {
            System.err.println("finally in 1st try block");
        }
    }
} ///:~
import java.io.*;

class InputFile {
    private BufferedReader in;
    InputFile(String fname) throws Exception {
        try {
            in =
                new BufferedReader(
                    new FileReader(fname));
            // Other code that might throw exceptions
        } catch(FileNotFoundException e) {
            System.err.println(
                "Could not open " + fname);
            // Wasn't open, so don't close it
            throw e;
        } catch(Exception e) {
            // All other exceptions must close it
            try {
                in.close();
            } catch(IOException e2) {
                System.err.println(
                    "in.close() unsuccessful");
            }
            throw e; // Rethrow
        } finally {
            // Don't close it here!!!
        }
    }
}
Exception Matching

- Base-class handler will catch
- Derived-class object

```java
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

class Human {
    public static void main(String[] args) {
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.err.println("Caught Sneeze");
        } catch(Annoyance a) {
            System.err.println("Caught Annoyance");
        }
    }
}
```
Catching Base-Class Constructor Exceptions

- Cannot have *anything* before base-class constructor call, not even a `try` block
- Thus cannot catch base-class constructor exceptions in the derived-class constructor
- Must show exception in derived-class constructor exception specification
“Inheritance” of Exceptions

• Base-class method throws an exception
  – Derived-class method may throw that exception or one derived from it

• Derived-class method
  – Throwing an exception that isn’t a type/subtype of an exception thrown by the base-class method
Overhead

• Exceptions are free as long as they don’t get thrown
• If they are thrown, very expensive
• Not using exceptions for normal flow of control
• Only using exceptions to indicate abnormal conditions
Guidelines

• Handling an exception
  – Only if you have enough information in the current context to correct the error (partially or totally)
  – Otherwise, just letting the exception propagate up

• Separating error handling code (which almost never runs) from code that represents the normal path of execution
  – Making code more readable
Guidelines Cont’d

• Handling tasks, not statements
  – Not encompassing every single statement in a try block
  – Instead, putting tasks inside of a try block, then handling each exception that can occur

• Using loops to retry
  – Like C++, no *resumption* in Java
  – If you need to retry, putting the exception handling inside a *do...while* loop
Guidelines Cont’d

• Using exceptions in constructors
  – People assume construction succeeds
• If you catch an exception, doing something with it
  – Not “stubbing it out” by having an empty Handler
  – This discards the exception; not robust coding
• Cleaning up using finally
Summary

• You have no choice in Java
  – You must catch exceptions
  – You must use exception specifications
  – The compiler enforces exception use

• A clean, straightforward error-handling model
  – You don’t have to decide how to handle errors
  – You don’t have to figure out how someone else handles errors
  – You don’t worry about whether errors get handled

• Seemingly more work at first
  – Only because you’ve been ignoring errors!