Principles of Software Construction: Objects, Design, and Concurrency

Object-Oriented Programming in Java

Josh Bloch

Charlie Garrod
Administrivia

• Homework 1 due Thursday 11:59 p.m.
  – Everyone must read and sign our collaboration policy
• First reading assignment due Tuesday
  – Effective Java Items 15 and 16
Key concepts from Thursday

• Bipartite type system – primitives & object refs
  – Single implementation inheritance
  – Multiple interface inheritance
• Easiest output – println, printf
• Easiest input – Command line args, Scanner
• Collections framework is powerful & easy to use
Outline

I. Object-oriented programming basics
II. Information hiding
III. Exceptions
Objects

• An **object** is a bundle of state and behavior
• State – the data contained in the object
  – In Java, these are the **fields** of the object
• Behavior – the actions supported by the object
  – In Java, these are called **methods**
  – Method is just OO-speak for function
  – Invoke a method = call a function
Classes

• Every object has a class
  – A class defines methods and fields
  – Methods and fields collectively known as **members**
• Class defines both type and implementation
  – Type ≈ where the object can be used
  – Implementation ≈ how the object does things
• Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  – Defines how users interact with instances
class Complex {
    private final double re; // Real Part
    private final double im; // Imaginary Part

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex subtract(Complex c) { ... }
    public Complex multiply(Complex c) { ... }
    public Complex divide(Complex c) { ... }
}
Class usage example

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

-1.0 + 1.0i
-0.0 + -1.0i
Interfaces and implementations

• Multiple implementations of API can coexist
  – Multiple classes can implement the same API
  – They can differ in performance and behavior

• In Java, an API is specified by *interface* or *class*
  – Interface provides only an API
  – Class provides an API and an implementation
  – A class can implement multiple interfaces
An interface to go with our class

public interface Complex {
    // No constructors, fields, or implementations!
    double realPart();
    double imaginaryPart();
    double r();
    double theta();
    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}

An interface defines but does not implement API
Modifying class to use interface

class OrdinaryComplex implements Complex {
    final double re; // Real Part
    final double im; // Imaginary Part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
    }
    public Complex subtract(Complex c) { ... }
    public Complex multiply(Complex c) { ... }
    public Complex divide(Complex c) { ... }
}
Modifying client to use interface

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                          + e.imaginaryPart() + "i");

        e = c.times(d);
        System.out.println(e.realPart() + " + "
                          + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

-1.0 + 1.0i
-0.0 + -1.0i
Interface permits multiple implementations

class PolarComplex implements Complex {
    final double r;
    final double theta;

    public PolarComplex(double r, double theta) {
        this.r = r;
        this.theta = theta;
    }

    public double realPart() { return r * Math.cos(theta); }
    public double imaginaryPart() { return r * Math.sin(theta); }
    public double r() { return r; }
    public double theta() { return theta; }

    public Complex plus(Complex c) { ... } // Completely different imple
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) { ... }
    public Complex dividedBy(Complex c) { ... }
}
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(1, Math.PI); // -1
        Complex d = new PolarComplex(1, Math.PI/2); // i

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + " + e.imaginaryPart() + "i");
    }
}

When you run this program, it **still** prints

-1.0 + 1.0i
-0.0 + -1.0i
Why multiple implementations?

• Different **performance**
  – Choose implementation that works best for your use

• Different **behavior**
  – Choose implementation that does what you want
  – Behavior *must* comply with interface spec (“contract”)

• Often **performance and behavior both vary**
  – Provides a functionality – performance tradeoff
  – Example: HashSet, LinkedHashSet, TreeSet
Interfaces and classes – the big picture

• Interfaces define *types*
  – Specify **what** functionality is provided by instances
  – These are the expectations implementations must meet

• Classes define implementations (and types)
  – Describe **how** instances meet expectations
Prefer interfaces to classes as types

...but don’t overdo it

• Use interface types for parameters and variables unless a single implementation will suffice
  – Supports change of implementation
  – Prevents dependence on implementation details

• But sometimes a single implementation will suffice

Set<Criminal> senate = new HashSet<>();          // Do this...

HashSet<Criminal> senate = new HashSet<>();      // Not this
interface Animal {
    void vocalize();
}
class Dog implements Animal {
    public void vocalize() {
        System.out.println("Woof!");
    }
}
class Cow implements Animal {
    public void vocalize() {
        moo();
    }
    public void moo() {
        System.out.println("Moo!");
    }
}

What Happens?
1. Animal a = new Animal();   a.vocalize();
2. Dog b = new Dog();         b.vocalize();
3. Animal c = new Cow();      c.vocalize();
4. Animal d = new Cow();      d.moo();
Historical note: simulation and the origins of OO programming

• Simula 67 was the first object-oriented language
• Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center
• Developed to support *discrete-event simulation*
  – Application: operations research, e.g. traffic analysis
  – Extensibility was a key quality attribute for them
  – Code reuse was another
Outline

I. Object-oriented programming basics
II. Information hiding
III. Exceptions
Information hiding

- Single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides internal data and other implementation details from other modules
- Well-designed code hides *all* implementation details
  - Cleanly separates API from implementation
  - Modules communicate *only* through APIs
  - They are oblivious to each others’ inner workings
- Known as *information hiding* or *encapsulation*
- Fundamental tenet of software design [Parnas, ‘72]
Benefits of information hiding

• **Decouples** the classes that comprise a system
  – Allows them to be developed, tested, optimized, used, understood, and modified in isolation

• **Speeds up system development**
  – Classes can be developed in parallel

• **Eases burden of maintenance**
  – Classes can be understood more quickly and debugged with little fear of harming other modules

• **Enables effective performance tuning**
  – “Hot” classes can be optimized in isolation

• **Increases software reuse**
  – Loosely-coupled classes often prove useful in other contexts
Information hiding with interfaces

• Declare variables using interface types
• Client can use only interface methods
• Fields not accessible from client code
• But this only takes us so far
  – Client can access non-interface members directly
  – In essence, it’s voluntary information hiding
Mandatory Information hiding

*visibility modifiers* for members

• **private** – Accessible *only* from declaring class

• **package-private** – Accessible from any class in the package where it is declared
  – Technically known as default access
  – You get this if no access modifier is specified

• **protected** – Accessible from subclasses of declaring class *(and within package)*

• **public** – Accessible from anywhere
Hiding interior state in OrdinaryComplex

class OrdinaryComplex implements Complex {
    private double re;  // Real Part
    private double im;  // Imaginary Part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
    }
    public Complex subtract(Complex c) { ... }
    public Complex multiply(Complex c) { ... }
    public Complex divide(Complex c) { ... }
}
Best practices for information hiding

• Carefully design your API
• Provide *only* functionality required by clients
  – *All* other members should be private
• You can always make a private member public later without breaking clients
  – But not vice-versa!
Outline

I. Object-oriented programming basics
II. Information hiding
III. Exceptions
What does this code do?

```java
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
    switch (errno) {
    case _ENOFILE:
        System.err.println("File not found: " + ...);
        return -1;
    default:
        System.err.println("Something else bad happened: " + ...);
        return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1; // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
} // The Slide lacks space to close the file. Oh well.
return i;
```
What does this code do?

```java
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
    switch (errno) {
    case _ENOFILE:
        System.err.println("File not found: " + ...);
        return -1;
    default:
        System.err.println("Something else bad happened: " + ...);
        return -1;
    }
}
DataInputStream dataInput = new DataInputStream(fIn);
if (dataInput == null)
    System.err.println("Unknown internal error.");
    return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
}  // The Slide lacks space to close the file. Oh well.
return i;
```
There’s a better way: *exceptions*

FileInputStream fileInput = null;

try {
    fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read file: " + e);
    return DEFAULT_VALUE;
}
Exceptions

• Inform caller of problem by transfer of control

• Semantics
  – Propagates up stack until main method is reached (terminates program), or exception is caught

• Sources
  – Program can throw explicitly
  – Underlying virtual machine (JVM) can generate
Control-flow of exceptions

```java
public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Caught index out of bounds");
    }
}

public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42; // Index is too high; throws exception
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}
```
Benefits of exceptions

• You can’t forget to handle common failure modes
  – Compare: using a flag or special return value

• Provide high-level summary of error, and stack trace
  – Compare: core dump in C

• Improve code structure
  – Separate normal code path from exceptional
  – Ease task of recovering from failure

• Ease task of writing robust, maintainable code
Checked vs. unchecked exceptions

• Checked exception
  – Must be caught or propagated, or program won’t compile
  – Exceptional condition that programmer must deal with

• Unchecked exception
  – No action is required for program to compile...
    • But uncaught exception will cause failure at runtime
  – Usually indicates a programming error

• Error
  – Special unchecked exception thrown by JVM*
  – Recovery is impossible*
Java’s exception hierarchy

- Throwable
  - Exception
    - RuntimeException
    - IOException
    - EOFException
    - Checked Exceptions:
      - FileNotFoundException
      - NullPointerException
      - IndexOutOfBoundsException
      - UnknownCheckedException
    - Error
      - StackOverflowError
  - Error
    - StackOverflowError

Diagram:

- Object
  - Throwable
    - Exception
      - RuntimeException
      - IOException
      - EOFException
      - Checked Exceptions
        - FileNotFoundException
        - NullPointerException
        - IndexOutOfBoundsException
        - UnknownCheckedException
      - Error
        - StackOverflowError

Design choice: checked exceptions, unchecked exceptions, and error codes

• Unchecked exception
  – Programming error, other unrecoverable failure

• Checked exception
  – An error that every caller should be aware of and handle

• Special return value (e.g., null from Map.get)
  – Common but atypical result

• **Do not use error codes** – too easy to ignore

• **Do not return null to indicate zero-length result**
  – Use a zero-length list or array instead
Using your own exception types

class SpanishInquisitionException extends RuntimeException {
    SpanishInquisitionException(String detail) {
        super(detail);
    }
}

class HolyGrail {
    public void seek() {
        ...
        if (heresyByWord() || heresyByDeed())
            throw new SpanishInquisitionException("heresy");
        ...
    }
}
Guidelines for using exceptions (1)

• Avoid unnecessary checked exceptions (EJ Item 71)
• Favor standard exceptions (EJ Item 72)
  – IllegalArgumentException – invalid parameter value
  – IllegalStateException – invalid object state
  – NullPointerException – null param where prohibited
  – IndexOutOfBoundsException – invalid index param
• Throw exceptions appropriate to abstraction (EJ Item 73)
Guidelines for using exceptions (2)

• Document all exceptions thrown by each method
  – Checked and unchecked (EJ Item 74)
  – But don’t *declare* unchecked exceptions!

• Include failure-capture info in detail message (Item 75)
  – throw new IllegalArgumentException(
    "Modulus must be prime: " + modulus);

• Don’t ignore exceptions (EJ Item 77)
  // Empty catch block IGNORES exception – Bad smell in code!
  try {
    ...
  } catch (SomeException e) { }
Remember this slide?

There's one part we didn't show you: cleanup

FileInputStream fileInput = null;

try {
    fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read file: " + e);
    return DEFAULT_VALUE;
} finally { // Close file if it's open
    if (fileInput != null) {
        try {
            fileInput.close();
        } catch (IOException ignored) {
            // No recovery necessary (or possible)
        }
    }
}
Manual resource termination is ugly and error prone, esp. for multiple resources

• Even good programmers usually get it wrong
  – Sun’s Guide to Persistent Connections got it wrong in code that claimed to be exemplary
  – Solution on page 88 of Bloch and Gafter’s Java Puzzlers is badly broken; no one noticed for years

• 70% of the uses of the close method in the JDK itself were wrong in 2008(!)

• Even “correct” idioms for manual resource management are deficient
The solution: try-with-resources

*Automatically closes resources!*

```java
try (DataInput dataInput =
    new DataInputStream(new FileInputStream(fileName))) {
    return dataInput.readInt();
} catch (IOException e) {
    System.err.println("Could not read file: " + e);
    return DEFAULT_VALUE;
}
```
File copy without manual termination

```java
class FileCopy {  
    static void copy(String src, String dest) throws IOException {  
        InputStream in = new FileInputStream(src);  
        try {  
            OutputStream out = new FileOutputStream(dest);  
            try {  
                byte[] buf = new byte[8 * 1024];  
                int n;  
                while ((n = in.read(buf)) >= 0) {  
                    out.write(buf, 0, n);  
                }  
            } finally {  
                out.close();  
            }  
        } finally {  
            in.close();  
        }  
    }  
}
```
File copy with try-with-resources

static void copy(String src, String dest) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest)) {
        byte[] buf = new byte[8 * 1024];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
Summary

- Interface-based designs handle change well
- Information hiding is crucial to good design
- Exceptions are far better than error codes
- The need for checked exceptions is rare
- `try-with-resources` is a big win; always use it