Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for change (class level)

Introduction to Java + Design for change: Information hiding

Charlie Garrod  Bogdan Vasilescu
Administrivia

• No smoking...
• Reading assignment due Tuesday: Effective Java Items 15 + 16
• Homework 1 due next Thursday 11:59 p.m.
  – Everyone must read and sign our collaboration policy
• Office hours start today
Key concepts from Tuesday

• Introduction to this course
  – Object-oriented programming (via Java)
  – Design
  – Design
  – Design
  – Concurrency
  – Real-world tools, real-world skills

• Course infrastructure
  – Git, GitHub, Gradle, Travis-CI
Key to design: Evaluation of alternatives

Version A:

```java
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
}
```

Version B':

```java
interface Comparator {
    boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
Metrics of software quality

- **Sufficiency / functional correctness**
  - Fails to implement the specifications ... Satisfies all of the specifications

- **Robustness**
  - Will crash on any anomalous event ... Recovers from all anomalous events

- **Flexibility**
  - Must be replaced entirely if spec changes ... Easily adaptable to changes

- **Reusability**
  - Cannot be used in another application ... Usable without modification

- **Efficiency**
  - Fails to satisfy speed or storage requirement ... satisfies requirements

- **Scalability**
  - Cannot be used as the basis of a larger version ... is basis for much larger version...

- **Security**
  - Security not accounted for at all ... No manner of breaching security is known

Source: Braude, Bernstein, Software Engineering. Wiley 2011
Today

• Introduction to Java
  – Java's bipartite type system: primitives and object references
  – Java collections framework: data structures and algorithms

• Information hiding: Design for change, design for reuse
  – Encapsulation: Visibility modifiers in Java
  – Interface types vs. class types
A simple Java program

class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
Java: A virtual machine architecture

• You first compile the source file:
  – `javac HelloWorld.java`
    • Produces `HelloWorld.class`

• Then run the class file with a Java Virtual Machine (JVM):
  – `java HelloWorld`
    • Executes the main method
Java type system

- **Primitive types**
  - `int`, `long`, `double`, `boolean`, `char`, `byte`, `short`, `float`
- **Object types**
  - Classes, interfaces, arrays, enums, annotations
  - Identity (==) is conceptually distinct from equality (.equals(...))
Java type system

- **Primitive types**
  - int, long, double, boolean, char, byte, short, float
- **Object types**
  - Classes, interfaces, arrays, enums, annotations
  - Identity (==) is conceptually distinct from equality (.equals(...))
- Java sometimes converts between primitive and object types
  - Integer, Long, Double, Boolean, Short, Char, Float, Byte
  - "Autoboxing" and "unboxing"
Java type system

• Primitive types
  – int, long, double, boolean, char, byte, short, float
• Object types
  – Classes, interfaces, arrays, enums, annotations
  – Identity (==) is conceptually distinct from equality (.equals(…))
• Java sometimes converts between primitive and object types
  – Integer, Long, Double, Boolean, Short, Char, Float, Byte
  – "Autoboxing" and "unboxing"
• Generic types (a.k.a. Parameterized types)
  – e.g. List<Integer>, HashMap<Bicycle,Double>
Some methods are present on all Objects

- `equals`: returns true if the two objects are conceptually equal
- `hashCode`: returns an int that must be equal for equal objects, and is likely to differ on unequal objects
- `toString`: returns a printable string representation
The class hierarchy

• The root is Object (all non-primitives are Objects)
• All classes except Object have one parent class
  – Specified with an extends clause:
    ```java
class Guitar extends Instrument {
  ...
}
```
  – If extends clause is omitted, defaults to Object
• A class is an instance of all its superclasses
Java interfaces

- Defines a type without an implementation
- More flexible than class types
  - An interface can extend multiple other interfaces
  - A class can implement multiple interfaces

```java
interface Comparator {
    boolean compare(int i, int j);
}

class AscendingComparator implements Comparator {
    public boolean compare(int i, int j) { return i < j; }
}
class DescendingComparator implements Comparator {
    public boolean compare(int i, int j) { return i > j; }
}
```
Java arrays

• Conceptually represented as an object
  – Provides .length, runtime bounds-checking

```java
String[] answers = new String[42];
if (answers.length == 42) {
    answers[42] = "no"; // ArrayIndexOutOfBoundsException
}
```
Java enums

- Like C enumerations, but represented as an object
  - Provides many object-oriented features, type safety, ...

```java
enum Planet { MERCURY, VENUS, EARTH, MARS,
              JUPITER, SATURN, URANUS, NEPTUNE; }

Planet location = ...;
if (location.equals(Planet.EARTH)) {
    System.out.println("Honey, I'm home!");
}
```
Java annotations

- Annotations mark code without any immediate functional effect

```java
class Bicycle {
    ...
    @Override
    public String toString() {
        return ...;
    }
}
```
Java's built-in class library

- `java.lang`: Many basic tools, library features
- `java.util`: Data structures and algorithms, other utilities
- `java.io`: Input/output
- `java.net`: Networking
- ...
Primary collection interfaces (in java.util)

- Collection
  - Set
  - List
  - Queue
  - Deque
- Map
Primary collection implementations

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet</td>
</tr>
<tr>
<td>List</td>
<td>ArrayList</td>
</tr>
<tr>
<td>Queue</td>
<td>ArrayDeque</td>
</tr>
<tr>
<td>Deque</td>
<td>ArrayDeque</td>
</tr>
<tr>
<td>[stack]</td>
<td>ArrayDeque</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
</tr>
</tbody>
</table>
### Other noteworthy collection implementations

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>LinkedHashSet, TreeSet, EnumSet</td>
</tr>
<tr>
<td>Queue</td>
<td>PriorityQueue</td>
</tr>
<tr>
<td>Map</td>
<td>LinkedHashMap, TreeMap, EnumMap</td>
</tr>
</tbody>
</table>
Collections usage example 1

Squeeze duplicate words out of command line

```java
public class Squeeze {
    public static void main(String[] args) {
        Set<String> s = new LinkedHashSet<>();
        for (String word : args)
            s.add(word);
        System.out.println(s);
    }
}
```

$ java Squeeze I came I saw I conquered
[I, came, saw, conquered]
Collections usage example 2

Print unique words in lexicographic order

```java
class Lexicon {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        for (String word : args)
            s.add(word);
        System.out.println(s);
    }
}
```

```bash
$ java Lexicon I came I saw I conquered
[I, came, conquered, saw]
```
Collections usage example 3

Print index of first occurrence of each word

class Index {
    public static void main(String[] args) {
        Map<String, Integer> index = new TreeMap<>();

        // Iterate backwards so first occurrence wins
        for (int i = args.length - 1; i >= 0; i--) {
            index.put(args[i], i);
        }

        System.out.println(index);
    }
}

$ java Index if it is to be it is up to me to do it
{be=4, do=11, if=0, is=2, it=1, me=9, to=3, up=7}
Java arrays are not Collections

- Arrays and collections don't mix
  - If you get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
  - See *Effective Java* Item 28 for details
More information on collections

• For much more information on collections, see: https://docs.oracle.com/javase/tutorial/collections/index.html
Today

• Introduction to Java
  – Java's bipartite type system: primitives and object references
  – Java collections framework: data structures and algorithms

• Information hiding: Design for change, design for reuse
  – Encapsulation: Visibility modifiers in Java
  – Interface types vs. class types
Visibility modifiers in Java ("encapsulation")

- **private**: Accessible only from declaring class
- "package private": Accessible from any class in package
  - a.k.a. default access, no visibility modifier
- **protected**: Accessible from package and also from subclasses
- **public**: Accessible anywhere
Visibility modifier example

- Consider:

```java
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x;
        y += p.y;
    }
}
```
Visibility modifier example

• Consider:

```java
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x; // This is OK. p.x and p.y are
        y += p.y; // accessible from the Point class!
    }
    public double getX() { return x; }
    public double getY() { return y; }
}
```
Information hiding is more general than visibility

- Use interfaces to separate expectations from implementation
  - Create interfaces to define your API
  - Declare variables, arguments, and return values as interface type
    - Write API in terms of other interfaces, not implementations
- Do not publicly document implementation details
A more complex example

public 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) { ... }
}
Using the Complex class

```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`
Extracting an interface from our class

```java
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 our earlier class to use the interface

```java
public class OrdinaryComplex implements Complex {
    private final double re; // Real part
    private 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 our earlier client to use the 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
Interfaces permit multiple implementations

```java
public class PolarComplex implements Complex {
    private final double r;        // Radius
    private final double theta;    // Angle

    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 new imple
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) { ... }
    public Complex dividedBy(Complex c) { ... }
}
```
Interface decouples client from implementation

```java
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI, 1); // -1
        Complex d = new PolarComplex(Math.PI/2, 1); // 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
Coming next Tuesday

- The information hiding punchline
- Specifications
- Introduction to testing