Principles of Software Construction: Objects, Design, and Concurrency

Object-Oriented Programming in Java

Josh Bloch Charlie Garrod

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due Thursday 11:59 p.m., EDT
  – Everyone must read and sign our collaboration policy
- First reading assignment due Today
  – 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. More object-oriented programming
II. Information hiding (AKA encapsulation)
III. Enums

Objects – review

- An **object** is a bundle of **state** and **behavior**
- State – the data contained in the object
  - Stored in the **fields** of the object
- Behavior – the actions supported by the object
  - Provided by **methods**
    - Method is just OO-speak for function
    - Invoke a method = call a function

Classes – review

- 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 ≈ **what** the object is and **where** it 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

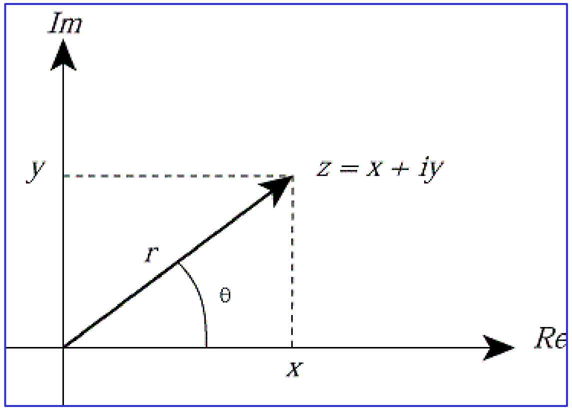
Class example – complex numbers

```
class Complex {
    final double re;  // Real Part
    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

```
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.printf("Sum:     %d + %di%n",
                          e.realPart(), e.imaginaryPart());
        e = c.times(d);
        System.out.printf("Product: %d + %di%n",
                          e.realPart(), e.imaginaryPart());
    }
}
```

When you run this program, it prints

```
Sum:     -1.0 + 1.0i
Product: -0.0 + -1.0i
```

Interfaces and implementations

- Multiple implementations of an API can coexist
  - Multiple classes can implement the same API
- In Java, an API is specified by *class* or *interface*
  - Class provides an API and an implementation
  - Interface provides *only* an API
  - A class can implement multiple interfaces
    - Remember diagram: `ElectricGuitar implements StringedInstrument, ElectricInstrument`

An interface to go with 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 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

```
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.printf("Sum:     %d + %di%n",
                          e.realPart(), e.imaginaryPart());
        e = c.times(d);
        System.out.printf("Product: %d + %di%n",
                          e.realPart(), e.imaginaryPart());
    }
}
```

When you run this program, it *still* prints

```
Sum:     -1.0 + 1.0i
Product: -0.0 + -1.0i
```

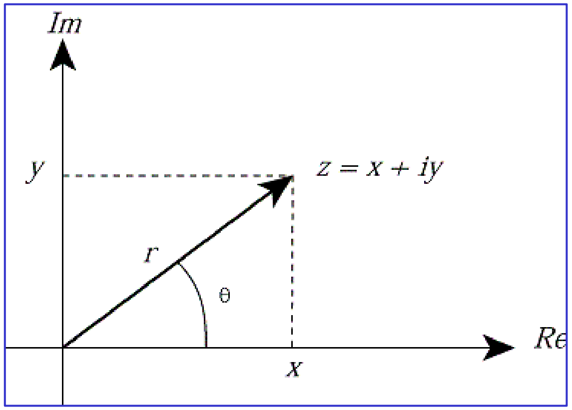
Interface enables multiple implementations

```java
class PolarComplex implements Complex {
    final double r;    // Different representation!
    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)  { ... } // Different implementation!
    public Complex minus(Complex c) { ... }
    public Complex times(Complex c) {
        return new PolarComplex(r * c.r(), theta + c.theta());
    }
    public Complex dividedBy(Complex c) { ... }
}
```



Interface decouples client from implementation

```
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.printf("Sum:     %d + %di%n",
                           e.realPart(), e.imaginaryPart());
        e = c.times(d);
        System.out.printf("Product: %d + %di%n",
                           e.realPart(), e.imaginaryPart());
    }
}
```

When you run this program, it *still* prints

```
Sum:     -1.0 + 1.0i
Product: -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`

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
  - In which cases write a class and be done with it

```
Set<Criminal> senate = new HashSet<>();          // Do this…
HashSet<Criminal> senate = new HashSet<>();      // Not this
```

Check your understanding

```
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();
```

Outline

I. More object-oriented programming
II. Information hiding (AKA encapsulation)
III. Enums

Information hiding (AKA encapsulation)

- 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
- Fundamental tenet of software design

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 and implementation-specific methods not accessible from client code
- But this takes us only so far
  - Client can access non-interface members directly
  - In essence, it’s **voluntary information hiding**

Mandatory Information hiding

Vsibility 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 any class

Hiding internal 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
- Use the most restrictive access modifier possible
- You can always make a private member public later without breaking clients but not vice-versa!

Outline

I. More object-oriented programming
II. Information hiding (AKA encapsulation)
III. Enums

Enums – review

- Java has object-oriented enums
- In simple form, they look just like C enums:

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

- But they have many advantages
  - Compile-time type safety
  - Multiple enum types can share value names
  - Can add or reorder without breaking constants
  - High-quality `Object` methods
  - Screaming fast collections (`EnumSet`, `EnumMap`)
  - Can easily iterate over all constants of an enum

You can add data to enums

```
public enum Planet {
    MERCURY(3.302e+23, 2.439e6), VENUS (4.869e+24, 6.052e6),
    EARTH(5.975e+24, 6.378e6), MARS(6.419e+23, 3.393e6);

    private final double mass;   // In kg.
    private final double radius; // In m.

    private static final double G = 6.67300e-11;  // N m²/kg²

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double mass()   { return mass; }
    public double radius() { return radius; }
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```

You can add behavior too

```
public enum Planet {
   ... // As on previous slide

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity; // F = ma
    }
}
```

Watch it go!

```
public static void main(String[] args) {
   double earthWeight = Double.parseDouble(args[0]);
   double mass = earthWeight / EARTH.surfaceGravity();

   for (Planet p : Planet.values()) {
      System.out.printf("Your weight on %s is %f%n",
                        p, p.surfaceWeight(mass));
   }
}
```

```
$ java WeightOnPlanet 180
Your weight on MERCURY is 68.023205
Your weight on VENUS is 162.909181
Your weight on EARTH is 180.000000
Your weight on MARS is 68.328719
```

You can even add value-specific behavior

```
public enum Operation {
    PLUS  ("+", (x, y) -> x + y),
    MINUS ("-", (x, y) -> x - y),
    TIMES ("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override public String toString() { return symbol; }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

Watch it go!

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                x, op, y, op.apply(x, y));
}

$ java TestOperation 4 2
4.000000 + 2.000000 = 6.000000
4.000000 - 2.000000 = 2.000000
4.000000 * 2.000000 = 8.000000
4.000000 / 2.000000 = 2.000000
```

Enums are your friend

- Use them whenever you have a type with a fixed number of values known at compile time
- You may find them useful on Homework 2
- See Effective Java Items 34, 42

Summary

- Interface-based provides flexibility
- Information hiding is crucial to good design
- Enums are your friend