Principles of Software Construction:
Objects, Design, and Concurrency

Part 1: Design for reuse

Behavioral subtyping

Josh Bloch **Charlie Garrod**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 graded soon
  - Please sign and return collaboration policy to Gradescope
- Reading due today: Effective Java, Items 17 and 50
  - Optional reading due Thursday
  - Required reading due next Tuesday
- Homework 2 due Sunday 11:59 p.m.

Required reading participation quiz

- https://bit.ly/32x0vsU

Design goals for your Homework 1 solution?

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011

One Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double similarityTo(Document d) {
        … ourText = download(url);
        … theirText = download(d.url);
        … ourFreq = computeFrequencies(ourText);
        … theirFreq = computeFrequencies(theirText);
        return cosine(ourFreq, theirFreq);
    }
    …
}
```

Compare to another Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double s
        … ourText =
        … theirText
        … ourFreq =
        … theirFreq
        return cosi
    }
    …
}
```

```
class Document {
    private final … frequencies;
    public Document(String url) {
        … ourText = download(url);
        frequencies = computeFrequencies(ourText);
    }

    public double similarityTo(Document d) {
        return cosine(frequencies,
                      d.frequencies);
    }
    …
}
```

Using the Document class

```
For each url:
    Construct a new Document

For each pair of Documents d1, d2:
    Compute d1.similarityTo(d2)
    …
```

- What is the running time of this, for n urls?

Latency Numbers Every Programmer Should Know

Jeff Dean, Senior Fellow, Google

PRIMITIVE	LATENCY: ns	us	ms
L1 cache reference	0.5		
Branch mispredict	5		
L2 cache reference	7		
Mutex lock/unlock	25		
Main memory reference	100		
Compress 1K bytes with Zippy	3,000	3	
Send 1K bytes over 1 Gbps network	10,000	10	
Read 4K randomly from SSD*	150,000	150	
Read 1 MB sequentially from memory	250,000	250	
Round trip within same datacenter	500,000	500	
Read 1 MB sequentially from SSD*	1,000,000	1,000	1
Disk seek	10,000,000	10,000	10
Read 1 MB sequentially from disk	20,000,000	20,000	20
Send packet CA->Netherlands->CA	150,000,000	150,000	150

The point

- Constants matter
- Design goals sometimes clearly suggest one alternative

Design goals for your Homework 2 solution?

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011

Key concepts from last Thursday

Key concepts from last Thursday

- Exceptions
- Specifying program behavior: contracts
- Testing:
  - Continuous integration, practical advice
  - Coverage metrics, statement coverage
- The `java.lang.Object` contracts

Selecting test cases

- Write tests based on the specification, for:
  - Representative cases
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
- Other tests: performance, security, system interactions, …

Methods common to all objects

- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on `Object`
  - **`toString`** - returns a printable string representation
  - **`equals`** - returns true if the two objects are "equal"
  - **`hashCode`** - returns an `int` that must be equal for equal objects, and is likely to differ on unequal objects

The hashCode contract

Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

The hashCode contract in English

- Equal objects **must** have equal hash codes
  - If you override `equals` you must override hashCode
- Unequal objects **should** have different hash codes
  - Take all value fields into account when calculating it
- Hash code must not change unless object mutated
  - Use a deterministic function of the field values

hashCode override example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17;  // Nonzero is good
        result = 31 * result + areaCode;   // Constant must be odd
        result = 31 * result + prefix;     //     "     "   "   "
        result = 31 * result + lineNumber; //     "     "   "   "
        return result;
    }

    ...
}
```

Alternative hashCode override

Less efficient, but otherwise equally good!

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        return Objects.hash(areaCode, prefix, lineNumber);
    }

    ...
}
```

A one liner. No excuse for failing to override hashCode!

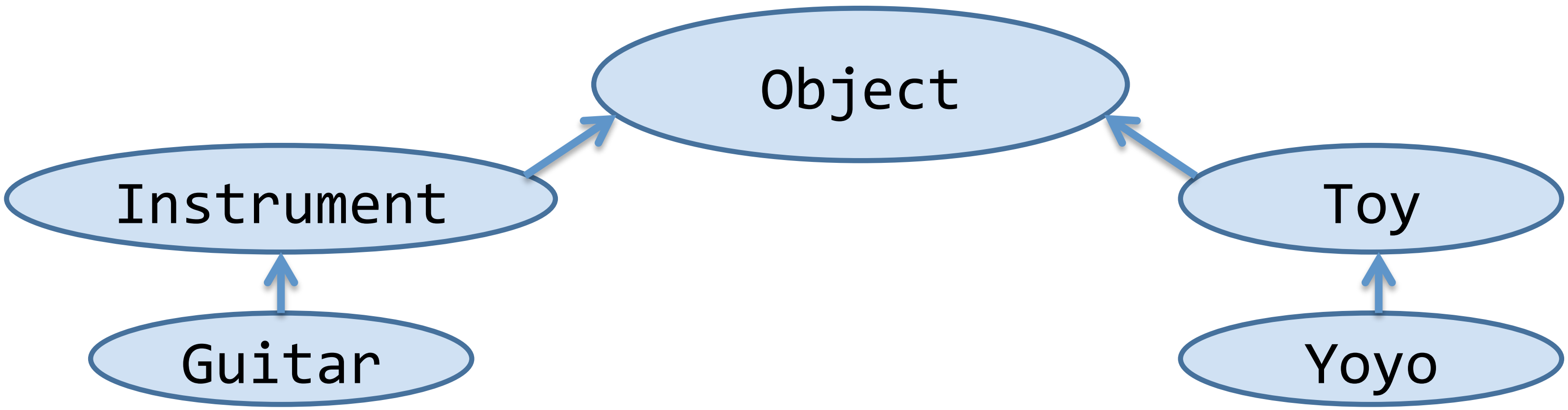
For more than you want to know about overriding object methods, see *Effective Java* Chapter 2

Today

- Behavioral subtyping
  - Liskov Substitution Principle
- Design for reuse: delegation and inheritance (Thursday)
  - Java-specific details for inheritance

Recall: The class hierarchy

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



Behavioral subtyping

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions

This is called the *Liskov Substitution Principle*.

Behavioral subtyping

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions
- Also applies to specified behavior. Subtypes must have:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

This is called the *Liskov Substitution Principle*.

LSP example: Car is a behavioral subtype of Vehicle

```
abstract class Vehicle {
  int speed, limit;

  //@ invariant speed < limit;

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  abstract void brake();
}
```

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0
        && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions

LSP example: Hybrid is a behavioral subtype of Car

```
class Car extends Vehicle {
  int fuel;
  boolean engineOn;
  //@ invariant speed < limit;
  //@ invariant fuel >= 0;

  //@ requires fuel > 0
        && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  void brake() { … }
}
```

```
class Hybrid extends Car {
  int charge;
  //@ invariant charge >= 0;
  //@ invariant …
  //@ requires (charge > 0
                  || fuel > 0)
              && !engineOn;
  //@ ensures engineOn;
  void start() { … }

  void accelerate() { … }

  //@ requires speed != 0;
  //@ ensures speed < \old(speed)
  //@ ensures charge > \old(charge)
  void brake() { … }
}
```

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method brake has stronger postcondition

institute for SOFTWARE RESEARCH

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

(Yes.)

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

Is this Square a behavioral subtype of Rectangle?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
     Square(int w) {
         super(w, w);
     }
}
```

```
class GraphicProgram {
    void scaleW(Rectangle r, int f) {
        r.setWidth(r.getWidth() * f);
    }
}
```

← **Invalidates stronger invariant (h==w) in subclass**

(Yes! But the Square is not a square…)

This Square is *not* a behavioral subtype of Rectangle

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
   //@ ensures w==neww
             && h==old.h;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
   //@ invariant h>0 && w>0;
   //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }

   //@ requires neww > 0;
   //@ ensures w==neww
             && h==neww;
   @Override
   void setWidth(int neww) {
      w=neww;
      h=neww;
   }
}
```

Today

- Behavioral subtyping
  - Liskov Substitution Principle
- Design for reuse: delegation and inheritance (Thursday)
  - Java-specific details for inheritance