

Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Design for reuse: delegation and inheritance

Josh Bloch

Charlie Garrod



Administrivia

- Homework 1 graded soon
- Reading due today: Effective Java, Items 17 and 50
 - Optional reading due Thursday
 - Required reading due next Tuesday
- Homework 2 due Thursday 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 similarityTo(Document d) {  
    ... ourText = download(url);  
    ... theirText = download(d.url);  
    ... ourFreq = computeFrequencies(ourText);  
    ... theirFreq = computeFrequencies(theirText);  
    return cosine(ourFreq, theirFreq);  
}  
...  
}
```

```
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

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

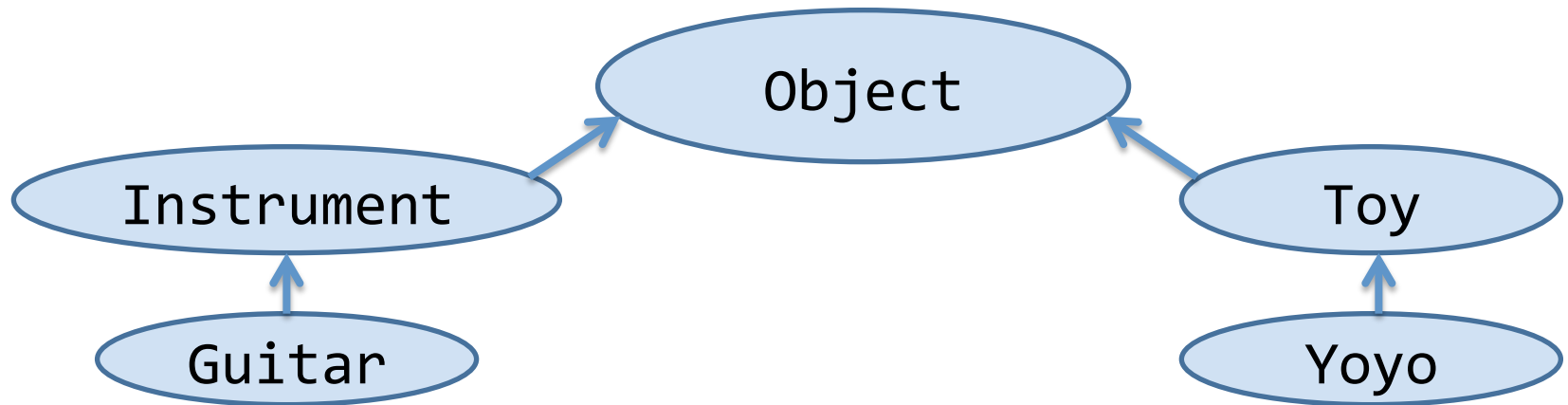
- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on `Object`
 - **`equals`** - returns `true` if the two objects are “equal”
 - **`hashCode`** - returns an `int` that must be equal for equal objects, and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

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 Objects)
- 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

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
 - Java-specific details for inheritance

Recall our earlier sorting example:

Version A:

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

Version B':

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
final Order ASCENDING = (i, j) -> i < j;  
final Order DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Order cmp) {  
    ...  
    boolean mustSwap =  
        cmp.lessThan(list[i], list[j]);  
    ...  
}
```

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - e.g. here, the Sorter is delegating functionality to some Order
- Judicious delegation enables code reuse

```
interface Order {
    boolean lessThan(int i, int j);
}
final Order ASCENDING = (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
    ...
    boolean mustSwap =
        cmp.lessThan(list[i], list[j]);
    ...
}
```

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - e.g. here, the Sorter is delegating functionality to some Order
- Judicious delegation enables code reuse
 - Sorter can be reused with arbitrary sort orders
 - Orders can be reused with arbitrary client code that needs to compare integers

```
interface Order {
    boolean lessThan(int i, int j);
}
final Order ASCENDING = (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
    ...
    boolean mustSwap =
        cmp.lessThan(list[i], list[j]);
    ...
}
```

Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

```
public interface List<E> {  
    public boolean add(E e);  
    public E      remove(int index);  
    public void   clear();  
    ...  
}
```

- Suppose we want a list that logs its operations to the console...

Using delegation to extend functionality

The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`

- One solution:

```
public class LoggingList<E> implements List<E> {  
    private final List<E> list;  
    public LoggingList<E>(List<E> list) { this.list = list; }  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return list.add(e);  
    }  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return list.remove(index);  
    }  
    ...  
}
```

Delegation and design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
 - E.g., the Order

Thursday: inheritance

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