

Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Behavioral subtyping

Charlie Garrod

Michael Hilton

Administrivia

- Homework 1 due tonight 11:59 p.m.
 - Everyone must read and sign our collaboration policy
- Reading due Tuesday: Effective Java, Items 15 and 39
- Homework 2 due next Thursday at 11:59 p.m.

Key concepts from Tuesday

Key concepts from Tuesday

- Java has a bipartite type system: primitives and objects
- Power of OO programming comes from dynamic dispatch
- Introduction to testing!
 - Test early, test often

Today

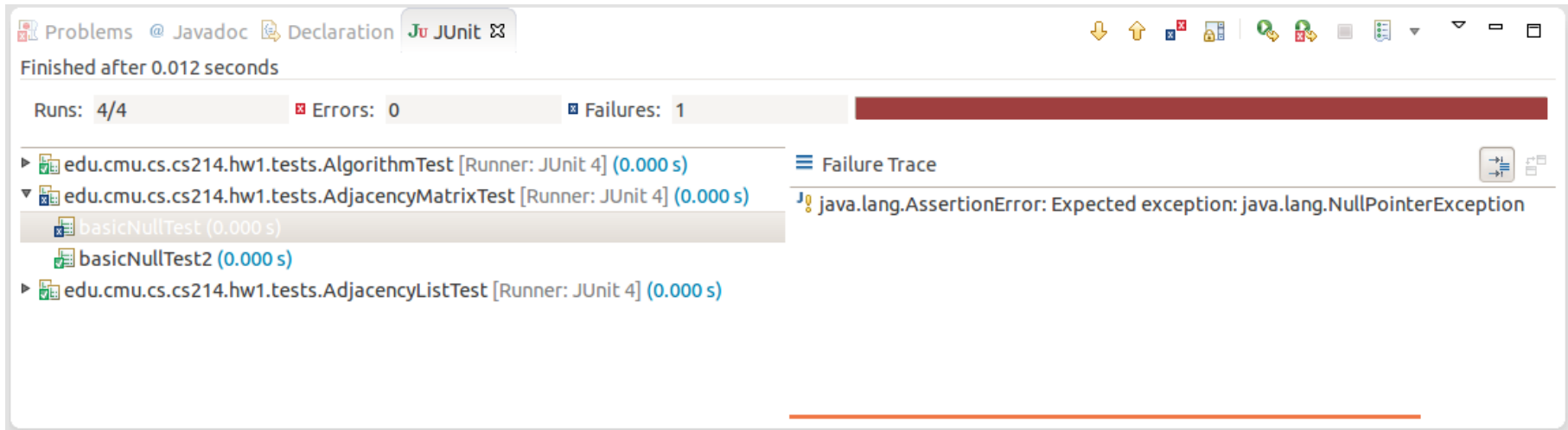
- Functional correctness, continued
- Behavioral subtyping
 - Liskov Substitution Principle
 - The `java.lang.Object` contracts

Unit testing

- Tests for small units: methods, classes, subsystems
 - Smallest testable part of a system
 - Test parts before assembling them
 - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment

JUnit

- A popular, easy-to-use, unit-testing framework for Java



A JUnit example

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest(){
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test...

    private int helperMethod...
}
```

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, ...

A testing example

```
/**  
 * computes the sum of the first len values of the array  
 *  
 * @param array array of integers of at least length len  
 * @param len number of elements to sum up  
 * @return sum of the first len array values  
 * @throws NullPointerException if array is null  
 * @throws ArrayIndexOutOfBoundsException if len > array.Length  
 * @throws IllegalArgumentException if len < 0  
 */  
int partialSum(int array[], int len);
```

A testing example

```
/**  
 * computes the sum of the first len values of the array  
 *  
 * @param array array of integers of at least length len  
 * @param len number of elements to sum up  
 * @return sum of the first len array values  
 * @throws NullPointerException if array is null  
 * @throws ArrayIndexOutOfBoundsException if len > array.length  
 * @throws IllegalArgumentException if len < 0  
 */  
int partialSum(int array[], int len);
```

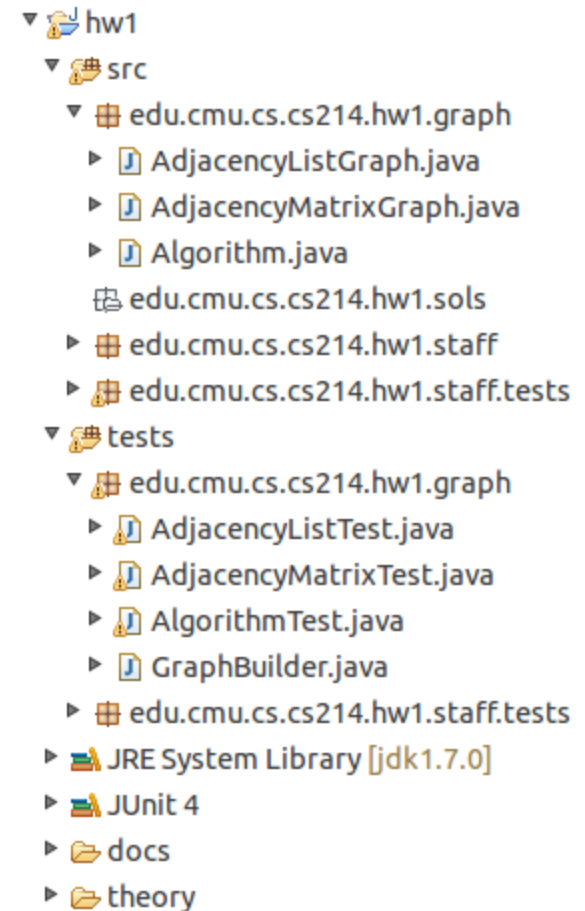
- Test null array
- Test length > array.length
- Test negative length
- Test small arrays of length 0, 1, 2
- Test long array
- Test length == array.length
- Stress test with randomly-generated arrays and lengths

A testing exercise

```
/**
 * Copies the specified array, truncating or padding with zeros
 * so the copy has the specified length. For all indices that are
 * valid in both the original array and the copy, the two arrays will
 * contain identical values. For any indices that are valid in the
 * copy but not the original, the copy will contain 0.
 * Such indices will exist if and only if the specified length
 * is greater than that of the original array.
 *
 * @param original the array to be copied
 * @param newLength the length of the copy to be returned
 * @return a copy of the original array, truncated or padded with
 *         zeros to obtain the specified length
 * @throws NegativeArraySizeException if newLength is negative
 * @throws NullPointerException if original is null
 */
int [] copyOf(int[] original, int newLength);
```

Test organization conventions

- Have a test class FooTest for each public class Foo
- Separate source and test directories
 - FooTest and Foo in the same package



Testable code

- Think about testing when writing code
 - Modularity and testability go hand in hand
- Same test can be used on all implementations of an interface!
- Test-driven development
 - Writing tests before you write the code
 - Tests can expose API weaknesses

Writing testable code

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        }
                    } else {
                        if () {
                            for () {
                                if () {
                                } else {
                                }
                            }
                            if () {
                                } else {
                                    if () {
                                    }
                                }
                            }
                        }
                    }
                }
            }
            if () {
                if () {
                    if () {
                        for () {
                        }
                    }
                }
            }
        }
    } else {
    }
}
```

Unit testing as a design mechanism:

- Code with low complexity
- Clear interfaces and specifications

Source:
<http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx>

Run tests frequently

- Run tests before every commit
 - Do not commit code that fails a test
- If entire test suite becomes too large and slow:
 - Run local package-level tests ("smoke tests") frequently
 - Run all tests nightly
 - Medium sized projects easily have 1000s of test cases
- Continuous integration servers scale testing

Continuous integration: Travis CI

The screenshot displays the Travis CI web interface for the repository `wyvernlang / wyvern`. The build status is `build passing`. The left sidebar shows the repository `wyvernlang/wyvern` with build `# 17`, a duration of 16 seconds, and a completion time of 3 days ago. The main content area shows the details for `Build #17`, including the commit `fd7be1c` by `potanin`. A yellow banner indicates that the job ran on legacy infrastructure. The build log is visible at the bottom, showing the setup of the Java environment and the execution of the `ant test` command.

Travis CI Build #17 - wyvernlang

<https://travis-ci.org/wyvernlang/wyvern/builds/79099642>

Travis CI Blog Status Help Jonathan Aldrich

Search all repositories

My Repositories +

- ✓ wyvernlang/wyvern # 17
 - ⌚ Duration: 16 sec
 - 📅 Finished: 3 days ago

wyvernlang / wyvern build passing

Current Branches Build History Pull Requests > Build #17 Settings

✓ SimpleWyvern-devel Asserting false (works on Linux, so its OK). # 17 passed

- 🐙 Commit fd7be1c
- 🐙 Compare 0e2af1f..fd7b
- ⌚ ran for 16 sec
- 📅 3 days ago

🕒 potanin authored and committed

📌 This job ran on our legacy infrastructure. Please read [our docs on how to upgrade](#)

Remove Log Download Log

```
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2
2
3 Build system information system_info
67
68 $ git clone --depth=50 --branch=SimpleWyvern-devel git.checkout 0.81s
69 $ jdk_switcher use oraclejdk8
70 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
71 $ java -Xmx32m -version
72 java version "1.8.0_31"
73 Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
74 Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
75 $ javac -J-Xmx32m -version
76 javac 1.8.0_31
77 $ cd tools 0.01s
78
79 The command "cd tools" exited with 0.
80 $ ant test 11.81s
81 Buildfile: /home/travis/build/wyvernlang/wyvern/tools/build.xml
82
83 copper-compose-compile:
```

Continuous integration: Travis CI build history

The screenshot shows the Travis CI web interface for the repository `wyvernlang / wyvern`. The page displays a list of build history entries, each with a status icon (green checkmark for passed, red X for failed), a commit message, the commit author, the build number, the result (e.g., 17 passed, 4 failed), the duration, and the time since the build.

My Repositories

- ✓ `wyvernlang/wyvern` #17
 - Duration: 16 sec
 - Finished: 3 days ago

Build History

Status	Commit Message	Author	Build #	Result	Duration	Time
✓	SimpleWyvern-devel Asserting false (works on L	potanin committed	# 17	passed	16 sec	3 days ago
✓	SimpleWyvern-devel Debugging mac bug.	potanin committed	# 16	passed	22 sec	3 days ago
✓	SimpleWyvern-devel Zooming in on Mac's IRBui	potanin committed	# 14	passed	15 sec	4 days ago
✓	SimpleWyvern-devel Zooming in on Mac LLVM b	potanin committed	# 13	passed	16 sec	4 days ago
✓	SimpleWyvern-devel Removed outdated tests	Jonathan Aldrich committed	# 7	passed	15 sec	11 days ago
✓	newlexer Merge branch 'master' of https://githu	Jonathan Aldrich committed	# 6	passed	14 sec	11 days ago
✓	master Build with JDK 8	Jonathan Aldrich committed	# 5	passed	13 sec	11 days ago
✗	master fixed Travis build script syntax error	Jonathan Aldrich committed	# 4	failed	5 sec	11 days ago
✗	master moved the VML file into the right place					

When should you stop writing tests?

When should you stop writing tests?

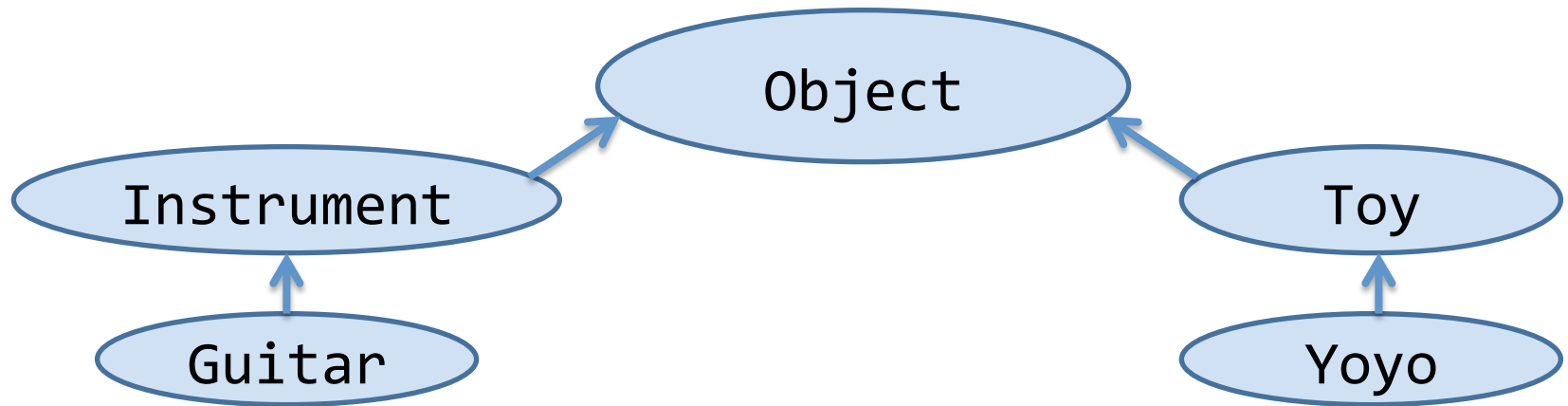
- When you run out of money...
- When your homework is due...
- When you can't think of any new test cases...
- The *coverage* of a test suite
 - Trying to test all parts of the implementation
 - Statement coverage: percentage of program statements executed
 - Compare to: method coverage, branch coverage, path coverage

Today

- Functional correctness, continued
- Behavioral subtyping
 - Liskov Substitution Principle
 - The `java.lang.Object` contracts

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;
```

```
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)  
    abstract void brake();
```

```
    //@ 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

- Functional correctness, continued
- Behavioral subtyping
 - Liskov Substitution Principle
 - The `java.lang.Object` contracts

Methods common to all Objects

- `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 for unequal objects
- `toString`: returns a printable string representation

The built-in `java.lang.Object` implementations

- Provide identity semantics:
 - `equals(Object o)`: returns true if `o` refers to this object
 - `hashCode()`: returns a near-random `int` that never changes
 - `toString()`: returns a string consisting of the type and hash code
 - For example: `java.lang.Object@659e0bfd`

The toString() specification

- Returns a concise, but informative textual representation
- Advice: Always override toString(), e.g.:

```
final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
    ...  
    @Override public String toString() {  
        return String.format("(%03d) %03d-%04d",  
                               areaCode, prefix, lineNumber);  
    }  
}
```

```
Number jenny = ...;  
System.out.println(jenny);  
Prints: (707) 867-5309
```

The equals(Object) specification

- Must define an equivalence relation:
 - Reflexive: For every object `x`, `x.equals(x)` is always true
 - Symmetric: If `x.equals(y)`, then `y.equals(x)`
 - Transitive: If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- Consistent: Equal objects stay equal, unless mutated
- "Non-null": `x.equals(null)` is always false

An equals(Object) example

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    @Override  
    public boolean equals(Object o) {  
        if (!(o instanceof PhoneNumber)) // Does null check  
            return false;  
        PhoneNumber pn = (PhoneNumber) o;  
        return pn.lineNumber == lineNumber  
            && pn.prefix == prefix  
            && pn.areaCode == areaCode;  
    }  
  
    ...  
}
```

The hashCode() specification

- Equal objects must have equal hash codes
 - If you override equals you must override hashCode
- Unequal objects should usually have different hash codes
 - Take all value fields into account when constructing it
- Hash code must not change unless object is mutated

A hashCode() 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;
    }

    ...
}
```

Summary

- Please complete the course reading assignments
- Test early, test often!
- Subtypes must fulfill behavioral contracts
- Always override hashCode if you override equals
- Always use `@Override` if you intend to override a method
 - Or let your IDE generate these methods for you...