Principles of Software Construction: Objects, Design, and Concurrency

Keepin' It Real:
Achieving and Maintaining Correctness in the Face of Change

Josh Bloch Charlie Garrod

School of Computer Science

Pop quiz - What does this print?

```
int[] a = new int[]  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++)
    sum1 += a[i];

int j;
int sum2 = 0;
for (j = 0; i < a.length; j++)
    sum2 += a[j];

System.out.println(sum1 - sum2);
```

Maybe not what you expect!

```
int[] a = new int[]  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++)
    sum1 += a[i];

int j;
int sum2 = 0;
for (j = 0; i < a.length; j++) //  Copy/paste error!
    sum2 += a[j];

System.out.println(sum1 - sum2);
```

You might expect it to print 0, but it prints 55

You could fix it like this…

```
int[] a = new int[]  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++)
    sum1 += a[i];

int j;
int sum2 = 0;
for (j = 0; j < a.length; j++)
    sum2 += a[j];

System.out.println(sum1 - sum2);  // Now prints 0, as expected
```

But this fix is far better…

```
int sum1 = 0;
for (int i = 0; i < a.length; i++)
   sum1 += a[i];

int sum2 = 0;
for (int i = 0; i < a.length; i++)
   sum2 += a[i];

System.out.println(sum1 - sum2);  // Prints 0
```

- Reduces scope of index variable to loop
- Shorter and less error prone

This fix is better still!

```
int sum1 = 0;
for (int x : a)
   sum1 += x;

int sum2 = 0;
for (int x : a)
   sum2 += x;

System.out.println(sum1 - sum2); // Prints 0
```

- Eliminates scope of index variable entirely!
- Even shorter and less error prone

Lessons from the quiz

- Minimize scope of local variables [EJ Item 45]
  - Declare variables at point of use
- Initialize variables in declaration
- Use common idioms
  - Design patterns in the small
- Watch out for bad smells in code
  - Such as index variable declared outside loop

Outline

I. More Java basics
   A. Type system
   B. Object hierarchy & methods
   C. Collections
   D. Enums

II. Unit testing

Java type system has two parts

Primitives	Object Reference Types
`int`, `long`, `byte`, `short`, `char`, `float`, `double`, `boolean`	Classes, interfaces, enums, arrays, annotations
No identity except their value	Have identity distinct from value
Immutable	Some mutable, some not
On stack, exist only when in use	On heap, garbage collected
Can't achieve unity of expression	Unity of expression with generics
Dirt cheap	More costly

Programming with primitives

A lot like C!

```
public class Junk {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        System.out.println(trailingZerosInFactorial(i));
    }

    static int trailingZerosInFactorial(int i) {
        int result = 0; // Conventional name for return value

        while (i >= 5) {
            i /= 5;      // equivalent to i = i / 5;
            result += i; // As in C, remainder is discarded
        }
        return result;
    }
}
```

Primitive type summary

• `int`	32-bit signed integer
• `long`	64-bit signed integer
• `byte`	8-bit signed integer
• `short`	16-bit signed integer
• `char`	16-bit unsigned character
• `float`	32-bit IEEE 754 floating point number
• `double`	64-bit IEEE 754 floating point number
• boolean	Boolean value: `true` or `false`

Deficient primitive types

- `byte`, `short` - use `int` instead!
  - `byte` is broken - should have been unsigned
- `float` - use `double` instead!
  - Provides too little precision
- Only compelling use case is large arrays in resource constrained environments

Boxed primitives

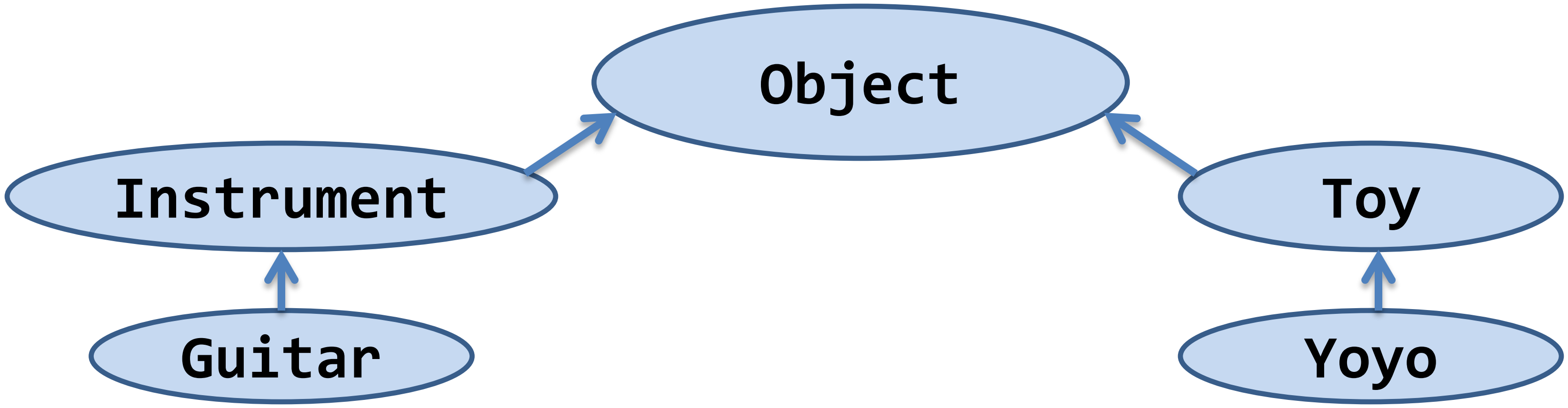
- Immutable containers for primitive types
- `Boolean`, `Integer`, `Short`, `Long`, `Character`, `Float`, `Double`
- Lets you "use" primitives in contexts requiring objects
- **Canonical use case is collections**
- **Don't use boxed primitives unless you have to!**
- Language does *autoboxing* and *auto-unboxing*
  - Blurs but does not eliminate distinction
  - There be dragons!

Outline

I. More Java basics
   A. Type system
   B. Object hierarchy & methods
   C. Collections
   D. Enums
II. Class Invariants
   A. Defensive copying
   B. Immutability
III. Unit testing

The class hierarchy

- The root is Object
- All classes except Object have one parent class
  - Specified with an extends clause
    `class Guitar extends Instrument`
  - If `extends` clause omitted, defaults to `Object`
- A class is an instance of all its superclasses



Implementation inheritance

- A class:
  - Inherits visible fields and methods from its superclasses
  - Can override methods to change their behavior
- Overriding method implementation must obey contract of its superclass(es)
  - Ensures subclass can be used anywhere superclass can
  - Liskov Substitution Principle (LSP)

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

`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 over the object lifetime
  - `toString()` - returns a nasty looking string consisting of the type and hash code
    - For example: `java.lang.Object@659e0bfd`

Overriding `Object` implementations

- No need to override `equals` and `hashCode` if you want identity semantics
  - When in doubt, don't override them
  - It's easy to get it wrong
- Nearly always override `toString`
  - `Println` invokes it automatically
  - Why settle for ugly?

Overriding `toString`

Overriding `toString` is easy and beneficial

```
public 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`

Overriding `equals`

- Overriding `equals` is hard - here's the contract

  The equals method implements an **equivalence relation**. It is:

  – **Reflexive**: For any non-null reference value x, x.equals(x) must return true.

  – **Symmetric**: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true.

  – **Transitive**: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.

  – **Consistent**: For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

  – For any non-null reference value x, x.equals(null) must return false.

Overriding hashCode

- Overriding hashCode is hard too - here's 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.

Why the contracts matter

- **No class is an island**
- If you put an object with a broken `equals` or `hashCode` into a collection, the collection breaks!
- Arbitrary behavior may result
  – System may generate incorrect results or crash
- To build a new *value type*, you *must* override `equals` and `hashCode`
  – Next lecture we'll show you how

Contracts (Review)

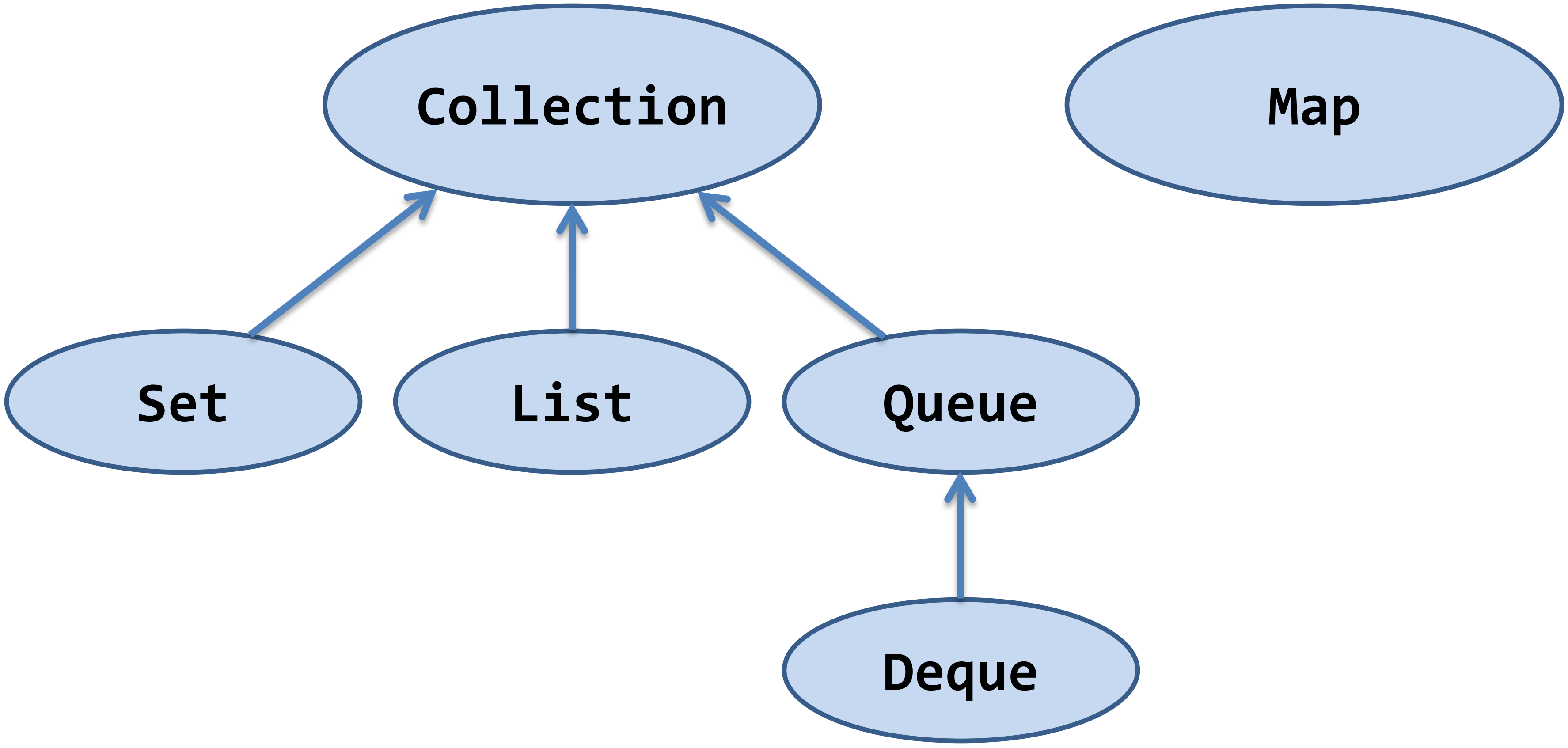
- Agreement between an object and its user
- Includes:
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Performance expectations

Outline

I. More Java basics
   A. Type system
   B. Object hierarchy & methods
   C. Collections
   D. Enums

II. Unit testing

Primary collection interfaces



Primary collection implementations

Interface	Implementation
`Set`	`HashSet`
`List`	`ArrayList`
`Queue`	`ArrayDeque`
`Deque`	`ArrayDeque`
`[stack]`	`ArrayDeque`
`Map`	`HashMap`

Other noteworthy collection impls

Interface	Implementation(s)

`Set`	`LinkedHashSet`
	`TreeSet`
	`EnumSet`

`Queue`	`PriorityQueue`
`Map`	`TreeMap` `EnumMap`

Collections usage example 1

Squeeze duplicate words out of command line

```
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

Prints a frequency table of words

```
public class Frequency {
  public static void main(String[] args) {
     Map<String, Integer> m = new TreeMap<>();
     for (String word : args) {
         Integer freq = m.get(word);
         m.put(word, (freq == null ? 1 : freq + 1));
     }
     System.out.println(m);
  }
}

$ java Frequency if it is to be it is up to me
{be=1, if=1, is=2, it=2, me=1, to=2, up=1}
```

What about arrays?

- Arrays aren't really a part of the collections framework
- But there is an adapter: `Arrays.asList`
- Arrays and collections don't mix
  - Arrays are covariant and reified
  - Generics are nonvariant and erased
- If you try to mix them and get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
- See *Effective Java* Item 25 for details

More information on collections

- For *much* more information on collections, see the annotated outline:

  https://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html

Outline

I. More Java basics
   A. Type system
   B. Object hierarchy & methods
   C. Collections
   D. Enums
II. Unit testing

Enums

- 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 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;

    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 Planet 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
```

Can add enum-constant specific behavior

- Each constant can provide its own overriding of a method
  - Don't do this unless you have to
  - Useful (but not required) for homework 3

```
 public enum BinaryOp implements BinaryOperation {
     PLUS   {double apply(double x, double y) { ... }},
     MINUS  {double apply(double x, double y) { ... }},
     TIMES  {double apply(double x, double y) { ... }},
     DIVIDE {double apply(double x, double y) { ... }};
}
```

Outline

I. More Java basics
   A. Type system
   B. Object hierarchy & methods
   C. Collections
   D. Enums
II. Unit testing

Context

- **Information Hiding** means modules are independent, communicate only via APIs
- **Contracts** describe behavior of the APIs (without revealing implementation details)
- **Testing** helps gain confidence that modules behave correctly

Formal verification

- Use mathematical methods to prove correctness with respect to the formal specification
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

"Testing shows the presence, not the absence of bugs"

Edsger W. Dijkstra 1969

Testing

- Executing the program with selected inputs in a controlled environment
- Goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald Knuth 1977

Unit tests

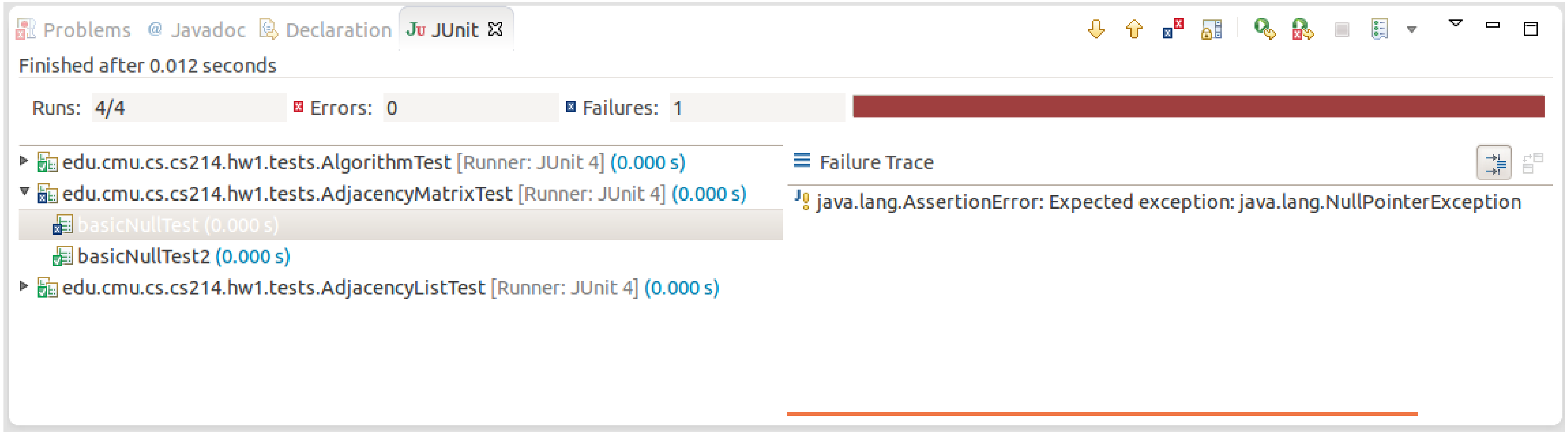
- Unit tests for small units: functions, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Little dependencies on other system parts or environment
- Insufficient but a good starting point,
  extra benefits:
  - Documentation (executable specification)
  - Design mechanism (design for testability)

Automate testing

- Execute a program with specific inputs, check output for expected values
- Easier to test small pieces than testing user interactions
- Set up testing infrastructure
- Execute tests regularly

JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



```
Finished after 0.012 seconds
Runs: 4/4    Errors: 0    Failures: 1

edu.cmu.cs.cs214.hw1.tests.AlgorithmTest [Runner: JUnit 4] (0.000 s)
edu.cmu.cs.cs214.hw1.tests.AdjacencyMatrixTest [Runner: JUnit 4] (0.000 s)
    basicNullTest (0.000 s)
    basicNullTest2 (0.000 s)
edu.cmu.cs.cs214.hw1.tests.AdjacencyListTest [Runner: JUnit 4] (0.000 s)

java.lang.AssertionError: Expected exception: java.lang.NullPointerException
```