Principles of Software Construction: Objects, Design, and Concurrency

Introduction to Java

Josh Bloch Charlie Garrod Darya Melicher

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due next Thursday 11:59 p.m.
  - Everyone must read and sign our collaboration policy
- First reading assignment due Tuesday
  - Effective Java Items 15 and 16

Outline

I. "Hello World!" explained
II. The type system
III. Quick 'n' dirty I/O
IV. Collections
V. Methods common to all Objects

The “simplest” Java Program

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```


institute for SOFTWARE RESEARCH

Complication 1: you must use a class even if you aren't doing OO programming

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Complication 2: `main` must be `public`

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```


institute for SOFTWARE RESEARCH

Complication 3: `main` must be `static`

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Complication 4: `main` must return `void`

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```


institute for SOFTWARE RESEARCH

Complication 5: `main` must declare command line arguments even if unused

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Complication 6: standard I/O requires use of `static` field of `System`

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Execution is a bit complicated

- First you **compile** the source file
  - javac `HelloWorld.java`
  - Produces class file `HelloWorld.class`
- Then you launch the program
  - `java HelloWorld`
  - Java Virtual Machine (JVM) executes `main` method

On the bright side…

- Has many good points to balance shortcomings
- Some verbosity is not a bad thing
  - Can reduce errors and increase readability
- Modern IDEs eliminate much of the pain
  - Type `psvm` instead of `public static void main`
- Managed runtime has many advantages
  - Safe, flexible, enables garbage collection
- It may not be best language for Hello World…
  - But Java is very good for large-scale programming!

Outline

I. “Hello World!” explained
II. The type system
III. Quick ‘n’ dirty I/O
IV. Collections
V. Methods common to all Objects

Java type system has two parts

Primitives	Object Reference Types
`int`, `long`, `byte`, `short`, `char`, `float`, `double`, `boolean`	Classes, interfaces, arrays, enums, 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 TrailingZeros {
    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;      // Same as i = i / 5; Remainder discarded
            result += i;
        }
        return result;
    }
}
```


institute for SOFTWARE RESEARCH

Primitive type summary

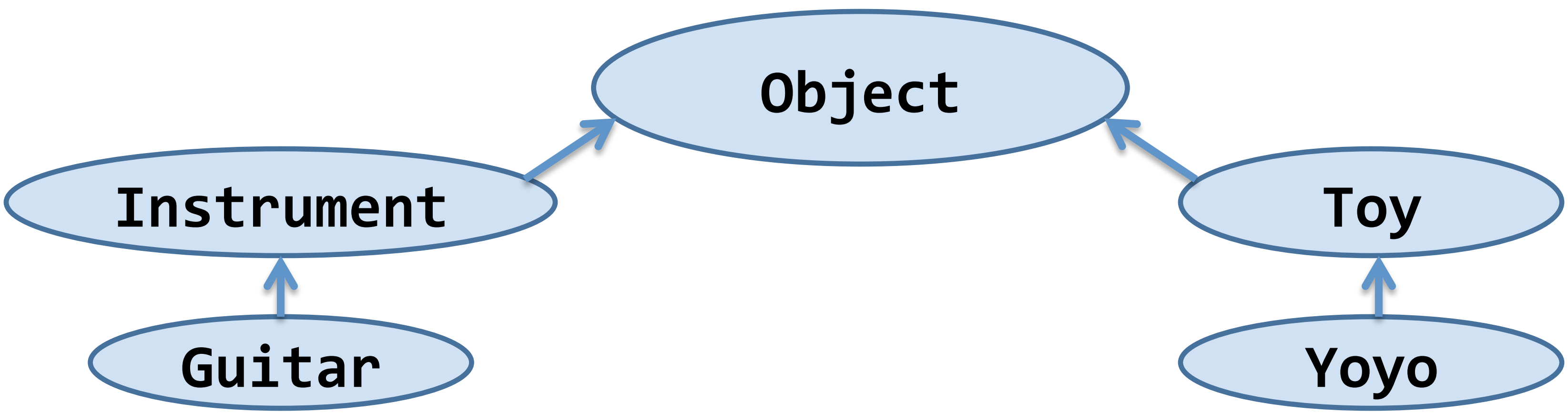
- `int` 32-bit signed integer
- `long` 64-bit signed integer
- `byte` 8-bit signed integer
- `short` 16-bit signed integer
- `char` 16-bit unsigned integer/character
- `float` 32-bit IEEE 754 floating point number
- `double` 64-bit IEEE 754 floating point number
- boolean Boolean value: `true` or `false`

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, especially in resource-constrained environments

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 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(s) of its superclass(es)
  - Ensures subclass can be used anywhere superclass can
  - Liskov Substitution Principle (LSP)

Interface types

- Defines a type without an implementation
- Much more flexible than class types
  - An interface can extend one or more others
  - A class can implement multiple interfaces

```
interface Comparable<T> {
    /**
     * Returns a negative number, 0, or a positive number as this
     * object is less than, equal to, or greater than other.
     */
    int compareTo(T other);
}
```

Enum types

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

```
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 existing uses
  - High-quality `Object` methods are provided
  - Screaming fast collections (`EnumSet`, `EnumMap`)
  - Can iterate over all constants of an enum

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!

Pop Quiz!

What does this fragment 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 57]
  - Declare variables at point of use
- Initialize variables in declaration
- Use common idioms
- Watch out for *bad smells in code*
  - Such as index variable declared outside loop

Outline

I. "Hello World!" explained
II. The type system
III. Quick 'n' dirty I/O
IV. Collections
V. Methods common to all Objects

Output

- Unformatted

```
System.out.println("Hello World");
System.out.println("Radius: " + r);
System.out.println(r * Math.cos(theta));
System.out.println();
System.out.print("*");
```

- Formatted

```
System.out.printf("%d * %d = %d%n", a, b, a * b); // Varargs
```

Command line input example

Echos all command line arguments

```
class Echo {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.print(arg + " ");
        }
    }
}

$ java Echo Woke up this morning, had them weary blues
Woke up this morning, had them weary blues
```

Command line input with parsing

Prints GCD of two command line arguments

```
class Gcd {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        System.out.println(gcd(i, j));
    }

    static int gcd(int i, int j) {
        return i == 0 ? j : gcd(j % i, i);
    }
}

$ java Gcd 11322 35298
666
```

Scanner input

Counts the words on standard input

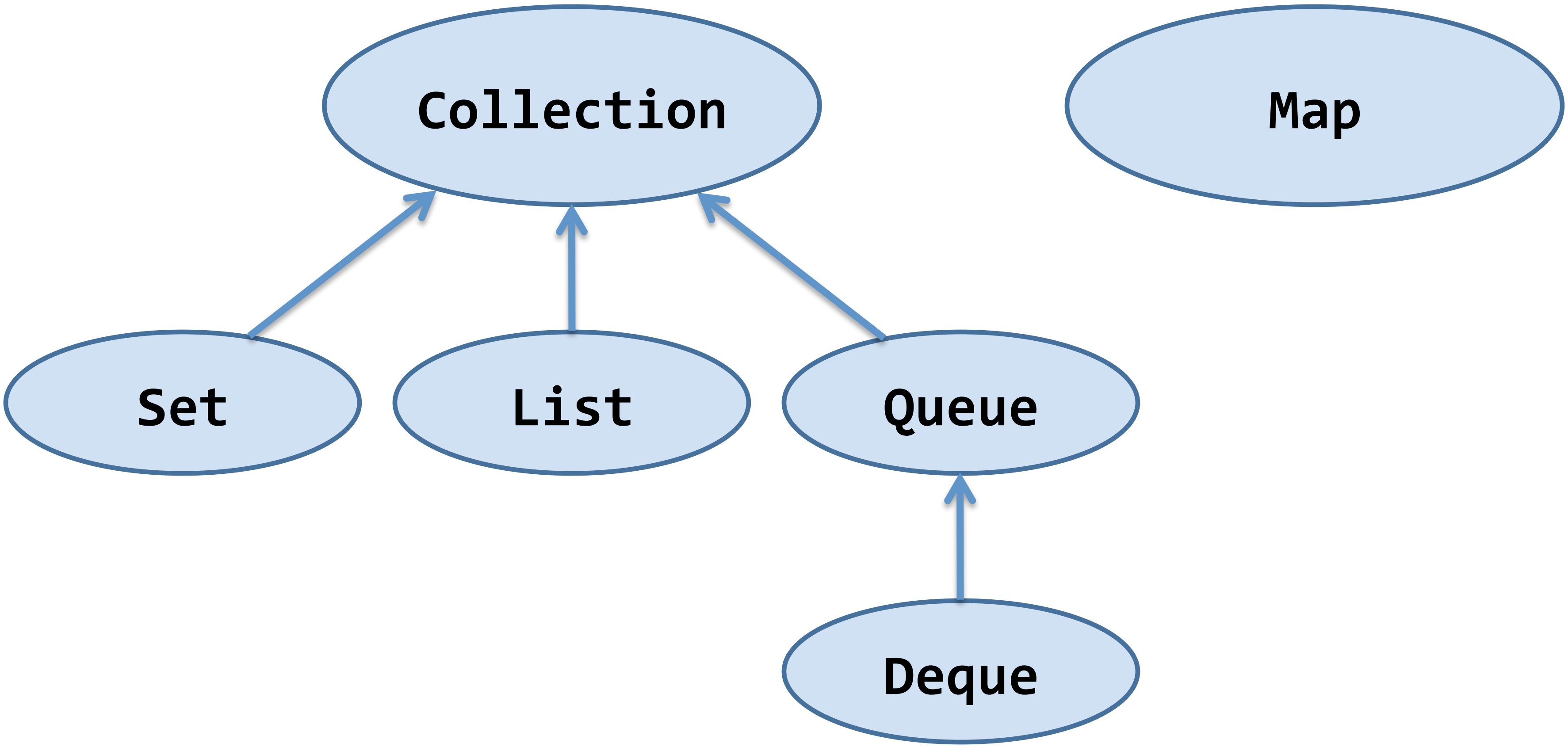
```
class Wc {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        long result = 0;
        while (sc.hasNext()) {
            sc.next(); // Swallow token
            result++;
        }
        System.out.println(result);
    }
}
```

```
$ java Wc < Wc.java
32
```

Outline

I. “Hello World!” explained
II. The type system
III. Quick ‘n’ dirty I/O
IV. Collections
V. Methods common to all Objects

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`	`LinkedHashMap`
	`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

Print unique words in lexicographic order

```
public class Lexicon {
   public static void main(String[] args) {
       Set<String> s = new TreeSet<>();
       for (String word : args)
           s.add(word);
       System.out.println(s);
   }
}

$ java Lexicon I came I saw I conquered
[I, came, conquered, saw]
```

Collections usage example 3

Print index of first occurrence of each word

```
class Index {
    public static void main(String[] args) {
        Map<String, Integer> index = new TreeMap<>();

        // Iterate backwards so first occurrence wins
        for (int i = args.length - 1; i >= 0; i--) {
            index.put(args[i], i);
        }
        System.out.println(index);
    }
}

$ java java Index if it is to be it is up to me to do it
{be=4, do=11, if=0, is=2, it=1, me=9, to=3, up=7}
```

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

- For more info on *any* library class, see javadoc
  - Search web for <fully qualified class name> 8
  - e.g., `java.util.scanner 8`

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 28 for details

Outline

I. “Hello World!” explained
II. The type system
III. Quick ‘n’ dirty I/O
IV. Collections
V. Methods common to all Objects

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

```
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 tricky – 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 also tricky – 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

Summary

- Java is well suited to large programs; small ones may seem a bit verbose
- Bipartite type system – primitives & object refs
  - Single implementation inheritance
  - Multiple interface inheritance
- A few simple I/O techniques will get you started
- Collections framework is powerful & easy to use