

# Principles of Software Construction: Objects, Design, and Concurrency

## Introduction to Java

**Josh Bloch**

**Charlie Garrod**



# 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. A brief introduction to collections

# The “simplest” Java Program

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

# 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!");  
    }  
}
```

# 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!");  
    }  
}
```



# Complication 5: main must declare command line arguments even if it doesn't use them

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

# Complication 6: println uses the static field `System.out`

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

# Execution is a bit complicated, too

- 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 (JVM) 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. A brief introduction to collections

# Java has a *bipartite* (2-part) type system

---

## Primitives

int, long, byte, short,  
char, float, double, boolean

No identity except their value

Immutable

On stack, exist only when in use

Can't achieve unity of  
expression

Dirt cheap

---

## Object Reference Types

Classes, interfaces, arrays,  
enums, annotations

Have identity distinct from value

Some mutable, some immutable

On heap, garbage collected

Unity of expression with generics

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;
    }
}
```

# 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` – typically use `int` instead!
  - `byte` is broken – should have been unsigned
- `float` – typically use `double` instead!
  - Provides too little precision
  - Few compelling use cases, e.g., large arrays in resource-constrained environments

# 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...

## *idiomatic Java for loop*

```
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, as expected
```

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

# This fix is better still!

## *for-each loop*

```
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, as expected
```

- 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
- Prefer for-each loops to regular for-loops
- Use common idioms
- Watch out for *bad smells in code*
  - Such as index variable declared outside loop



# Objects

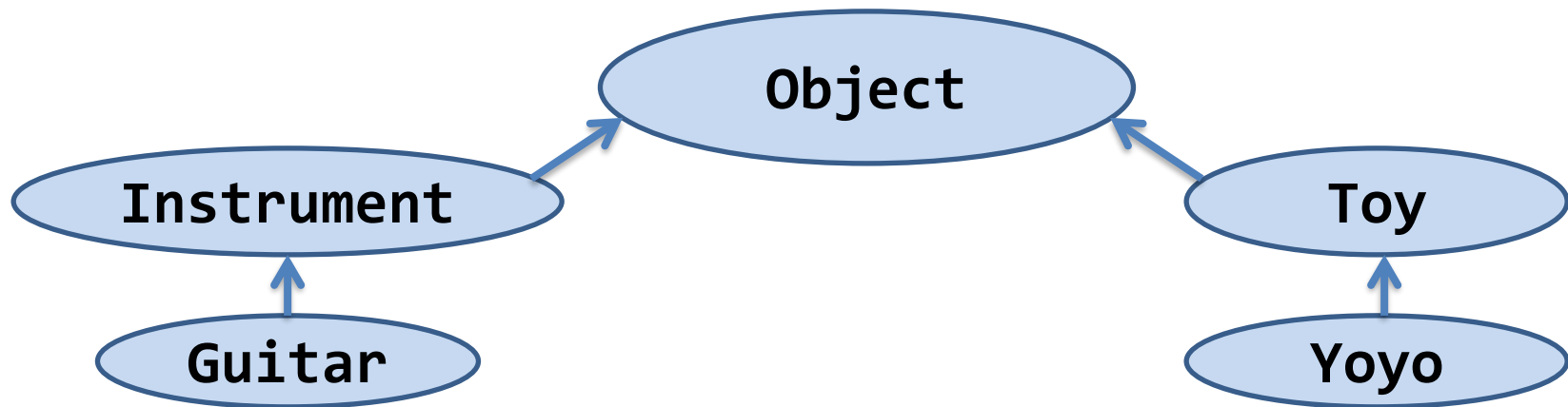
- All non-primitives are represented by objects.
- An **object** is a bundle of state and behavior
- State – the data contained in the object
  - In Java, these are called its instance **fields**
- Behavior – the actions supported by the object
  - In Java, these are called its **methods**
  - Method is just OO-speak for function
  - “Invoke a method” is OO-speak for “call a function”

# Classes

- Every object has a class
  - A class defines methods and fields
  - Methods and fields collectively known as **members**
- Class defines both type and implementation
  - Type  $\approx$  **what** object does (hence **where** it can be used)
  - Implementation  $\approx$  **how** the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  - Defines how users interact with its instances

# The Java 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

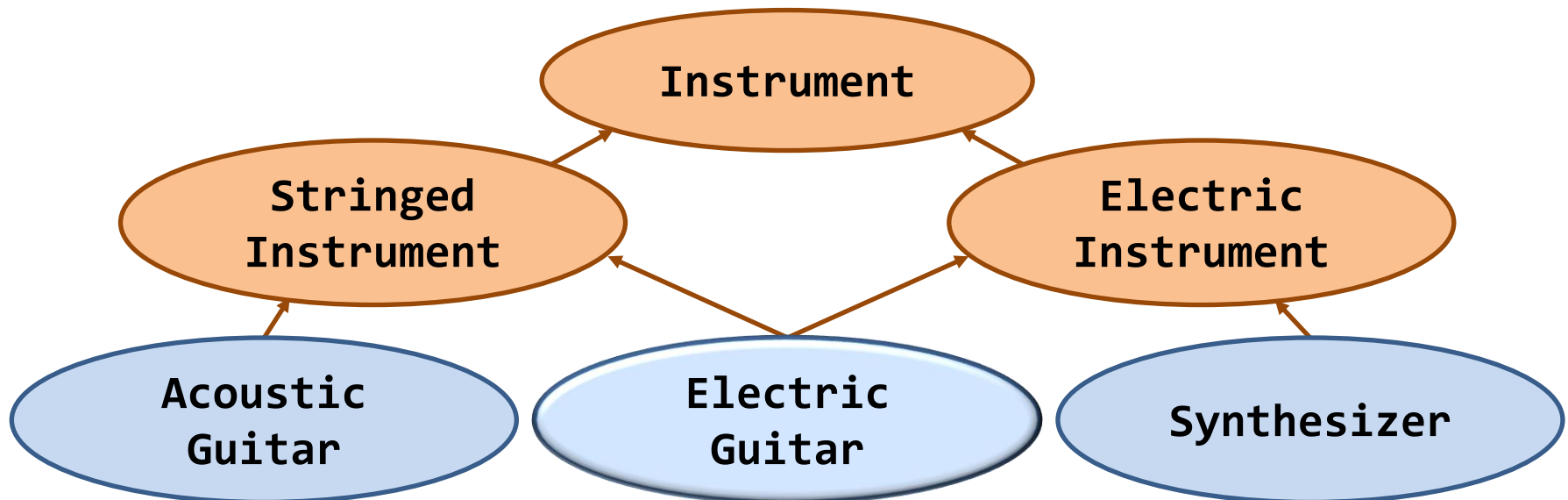


# Implementation inheritance

- A class:
  - Inherits visible fields and methods from its superclasses
  - Can *override* methods to change their behavior
- Overriding method implementation must obey the contract(s) of its superclass(es)
  - Ensures subclass can be used anywhere superclass can
  - *Liskov Substitution Principle* (LSP)
  - We will talk more about this in a later class

# 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



# 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
- Let 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!

# Comparing values

`x == y` compares `x` and `y` “directly”:

**primitive values:** returns true if `x` and `y` **have the same value**

**objects refs:** returns true if `x` and `y` **refer to same object**

`x.equals(y)` compares the ***values of the objects*** referred to by `x` and `y`\*

\* Assuming it makes sense to do so for the objects in question



# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);  
-----
```

# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);
```

-----

true

i 5

j 5

# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);  
-----
```

true

i 5

j 5

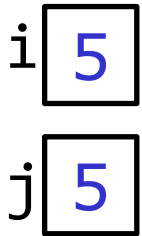
```
String s = "foo";  
String t = s;  
System.out.println(s == t);  
-----
```

# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);
```

-----

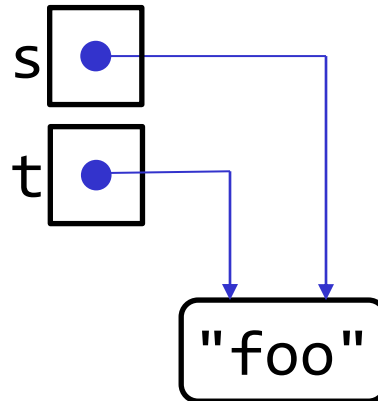
true



```
String s = "foo";  
String t = s;  
System.out.println(s == t);
```

-----

true

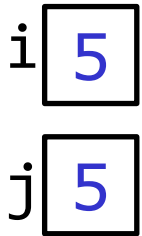


# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);
```

-----

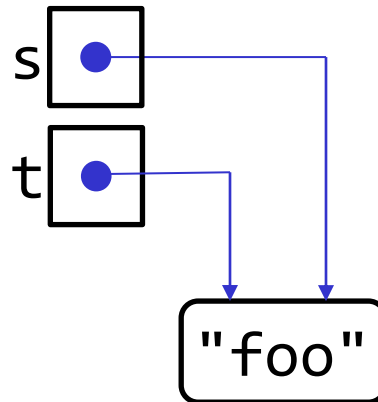
true



```
String s = "foo";  
String t = s;  
System.out.println(s == t);
```

-----

true



```
String u = "iPhone";  
String v = u.toLowerCase();  
String w = "iphone";  
System.out.println(v == w);
```

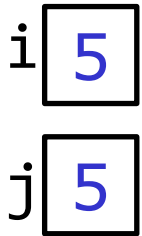
-----

# True or false?

```
int i = 5;  
int j = 5;  
System.out.println(i == j);
```

-----

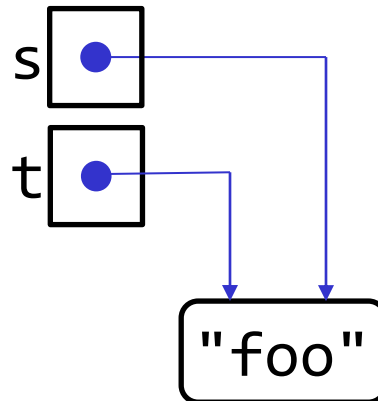
true



```
String s = "foo";  
String t = s;  
System.out.println(s == t);
```

-----

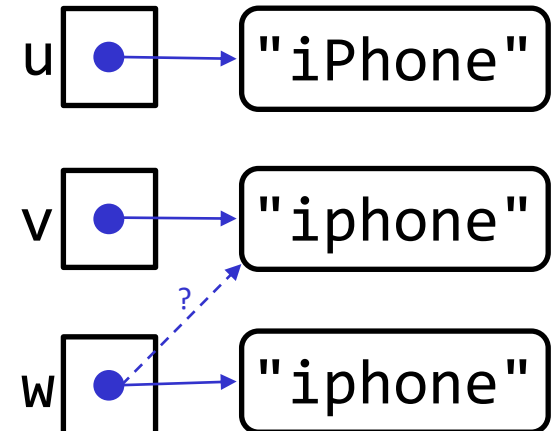
true



```
String u = "iPhone";  
String v = u.toLowerCase();  
String w = "iphone";  
System.out.println(v == w);
```

-----

**Undefined! (false in practice)**



# The moral

- **Always use `.equals` to compare object refs!**
  - (Except for enums, which are special)
  - The `==` operator can fail silently and unpredictably when applied to object references
  - Same goes for `!=`

# Outline

- I. “Hello World!” explained
- II. The Java type system
- III. Quick ‘n’ dirty I/O
- IV. A brief introduction to collections



# 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 – very similar to C

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

# Command line input example

*Echos all its 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 the GCD of its 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);  
    }  
}
```

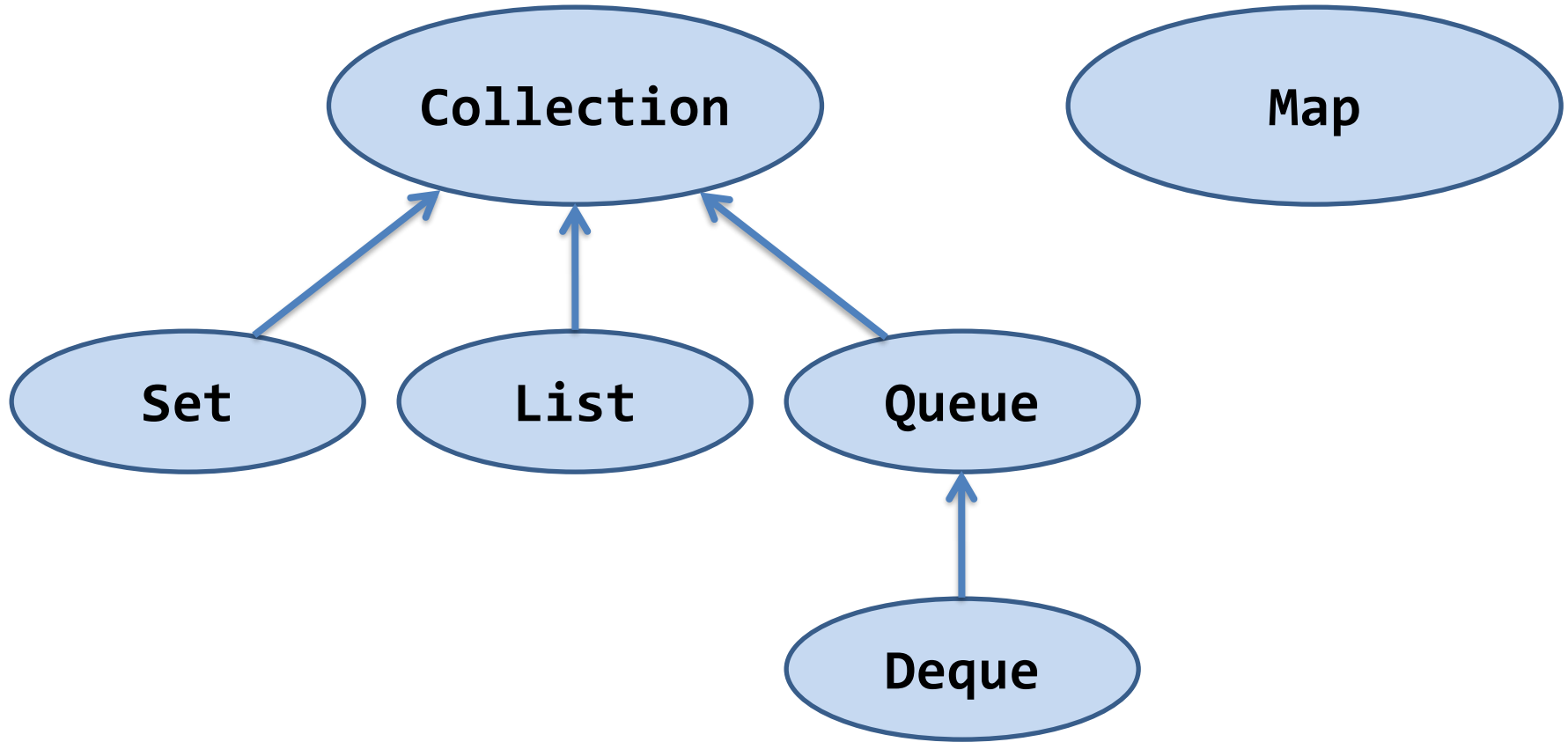
```
$ java Wc < Wc.java
```

```
32
```

# Outline

- I. “Hello World!” explained
- II. The type system
- III. Quick ‘n’ dirty I/O
- IV. A brief introduction to collections

# Primary collection interfaces



# “Primary” collection implementations

Interface	Implementation
Set	HashSet
List	ArrayList
Queue	ArrayDeque
Deque	ArrayDeque
(stack)	ArrayDeque
Map	HashMap

# Other noteworthy collection implementations

Interface	Implementation(s)
Set	LinkedHashSet TreeSet EnumSet
Queue	PriorityQueue
Map	LinkedHashMap TreeMap EnumMap



# Collections usage example 1

*Squeezes 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 unique words in alphabetical 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

*Prints the index of the 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 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:

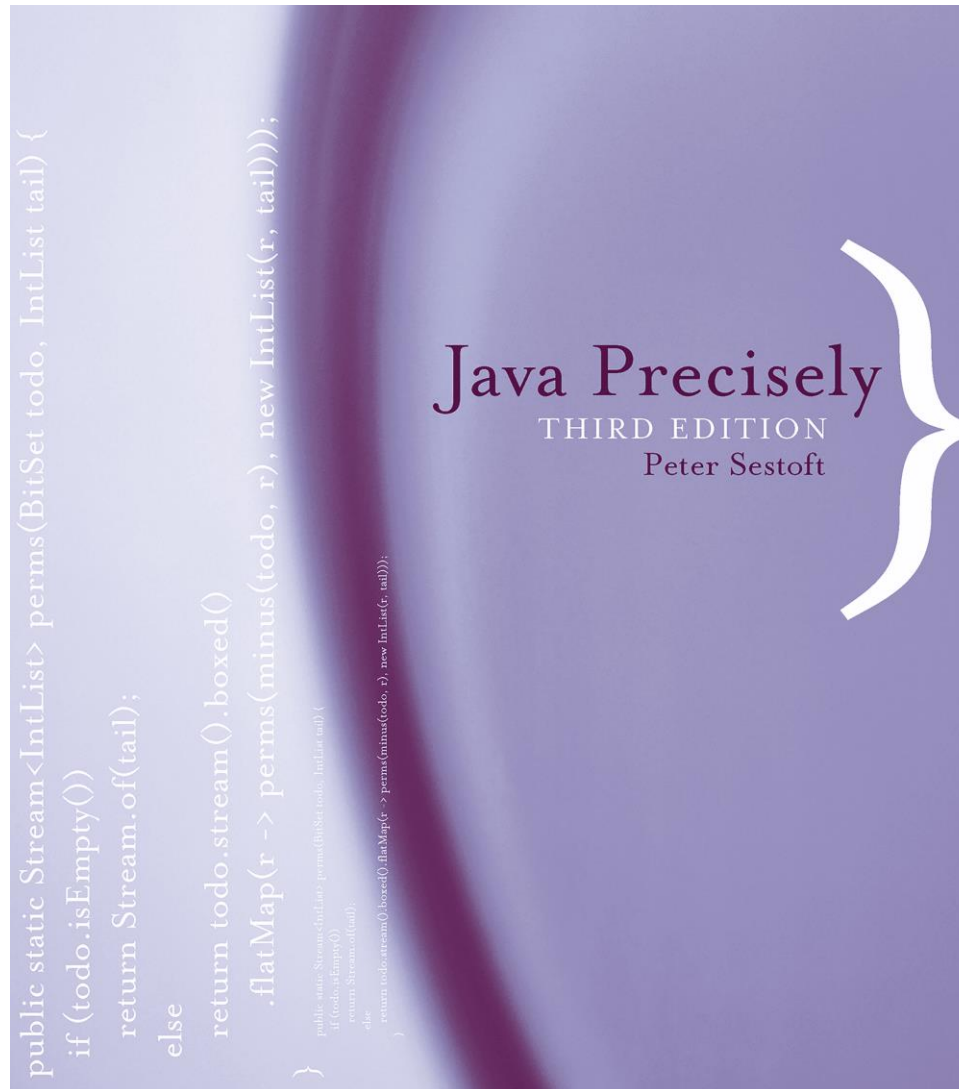
<https://docs.oracle.com/javase/11/docs/technotes/guides/collections/reference.html>

- For more info on *any* library class, see javadoc
  - Search web for <fully qualified class name> 11
  - e.g., `java.util.scanner` 11

# What about arrays?

- Arrays aren't a part of the collections framework
- But there is an adapter: `Arrays.asList`
- Arrays and collections don't mix well
- If you try to mix them and get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
  - But arrays of primitives (e.g., `int[]`) are preferable to lists of boxed primitives (e.g., `List<Integer>`)
- See *Effective Java* Item 28 for details

# To learn Java quickly



# 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 go a long way
- Collections framework is powerful & easy to use