Principles of Software Construction: Objects, Design, and Concurrency

Introduction to Java

Josh Bloch Charlie Garrod

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due next Thursday 11:59 p.m. EST
  - 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

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…

- Java 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* (VM) has many advantages
  - Safe, flexible, obviates need for manual memory mgt.
- 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

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

Primitives	**Object Reference Types**
`int, long, byte, short, char, float, double, boolean`	Everything else: classes, interfaces, arrays, enums, annotations
No identity except their value	Have identity distinct from value
Immutable	Some mutable, some immutable
On heap (local variables) or stack (object or class fields)	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(numTrailingZerosInFactorial(i));
    }

    static int numTrailingZerosInFactorial(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
    - large arrays in resource-constrained environments
    - Machine learning (but JVM doesn’t necessarily give you sufficient control for this: requires *vector instructions*)

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

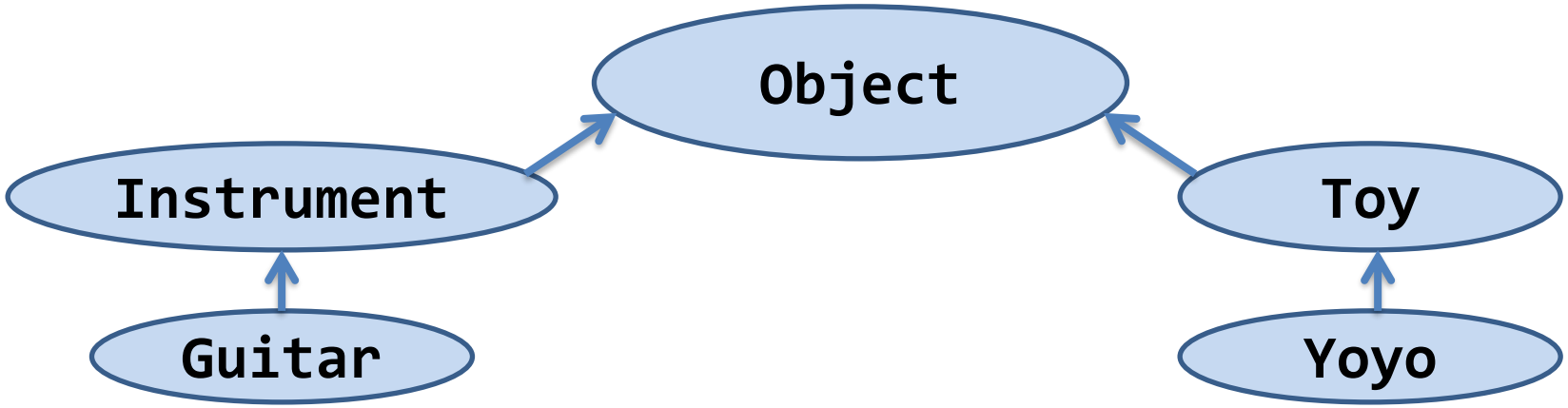
- Every object has a *class*
  - A class defines *methods* and *fields*
  - Methods and fields collectively known as *members*
- Class defines both type and implementation
  - Type ≈ what object does (hence where it can be used)
  - Implementation ≈ how the object does things
- Loosely speaking, the methods of a class define 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* (AKA *direct superclass*)
  - 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*

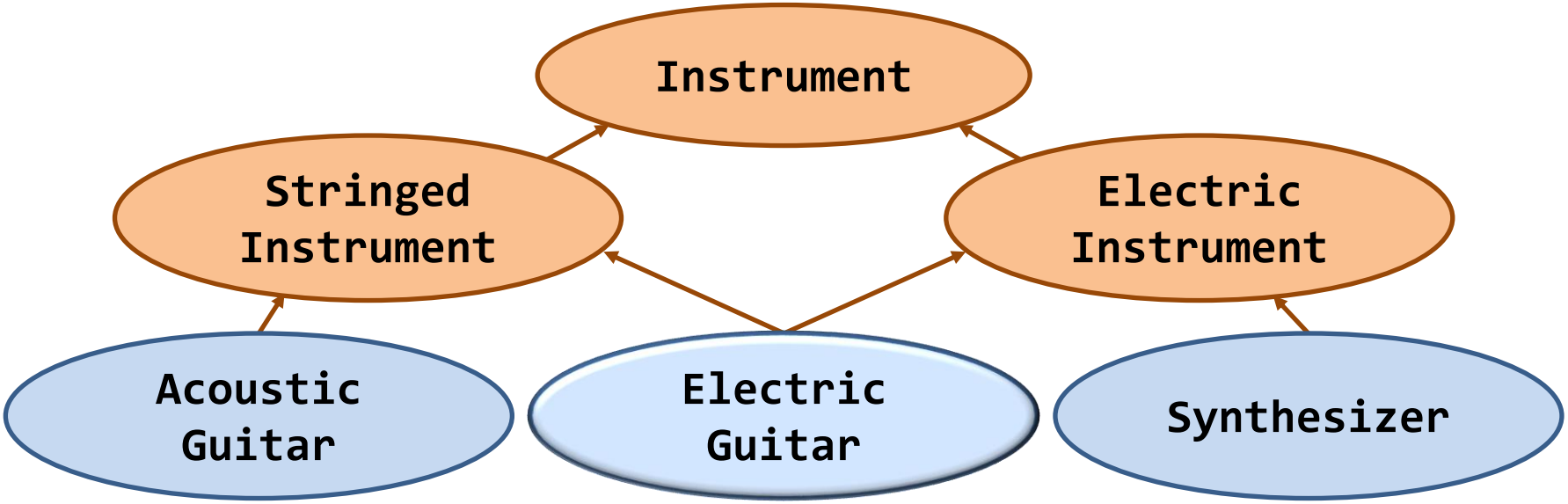


Implementation inheritance

- A class:
  - *Inherits* visible fields and methods from its superclasses
  - Can *override* methods to change their behavior
    - i.e., subtype can provide new implementations of superclass methods
    - Known as *subtype polymorphism*
- 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 access all constants of an enum (e.g. `Planet.values()`)

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
  - e.g., `Map<String, Integer>`
- **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 references: returns true if x and y **refer to same object**

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

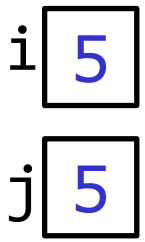
* Asuming it makes sense to do so for the objects in question

True or false?

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

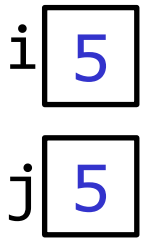
True or false?

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



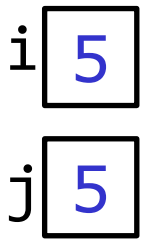
True or false?

```
int i = 5;                      String s = "foo";
int j = 5;                      String t = s;
System.out.println(i == j);     System.out.println(s == t);
---------------------------     ---------------------------
true
```

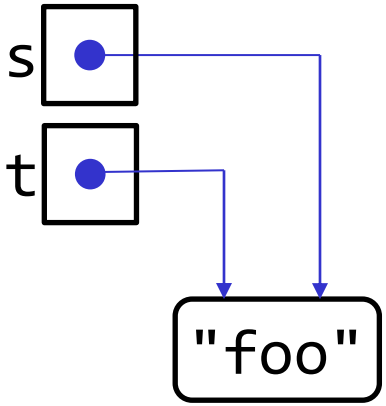


True or false?

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

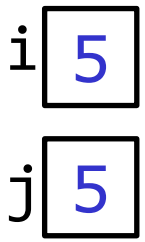



```
String s = "foo";
String t = s;
System.out.println(s == t);
---------------------------
true
```

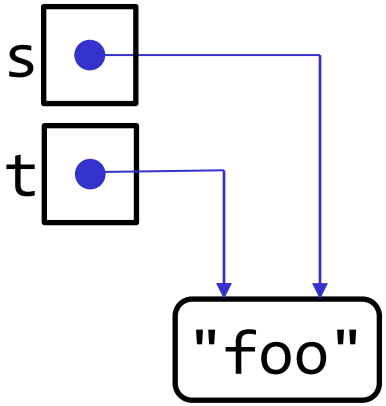


True or false?

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



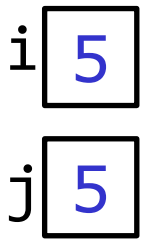

```
String s = "foo";
String t = s;
System.out.println(s == t);
---------------------------
true
```



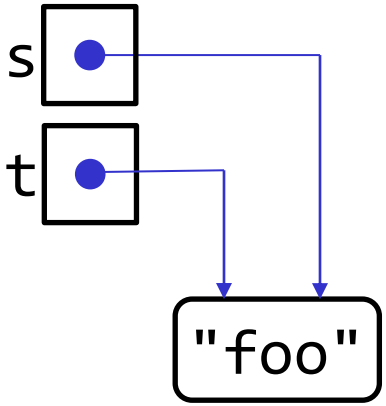

```
String u = "iPhone";
String v = u.toLowerCase();
String w = "iphone";
System.out.println(v == w);
---------------------------
```

True or false?

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

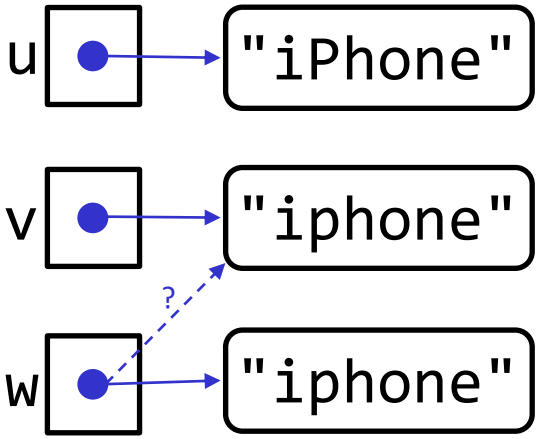



```
String s = "foo";
String t = s;
System.out.println(s == t);
---------------------------
true
```




```
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 the != operator

Outline

I. “Hello World!” explained
II. The Java type system
III. Quick ‘n’ dirty I/O

Console output

- Unformatted

```
System.out.println("Hello World");
System.out.println("Radius: " + r);     // String concatenation
System.out.println(r * Math.cos(theta));// Param type is double
System.out.println();   // No param, prints a blank line
System.out.print("*");  // Doesn't print line separator
```

- Formatted – very similar to C

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

Command line input (& console output)

Echos all its command line arguments

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

```
$ 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
```

Console input with Scanner

Counts the words on standard input

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

$ java Wc < Wc.java
32
```

Summary

- Java is well suited to large programs; for small ones, it may seem a bit verbose
- Bipartite type system – primitives and object refs
- Single implementation inheritance
- Multiple interface inheritance
- A few simple I/O techniques go a long way