Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for change (class level)

Introduction to Java +
Design for change: Information hiding

Michael Hilton **Bogdan Vasilescu**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Website: http://www.cs.cmu.edu/~mhilton/classes/17-214/s19/
- No smoking…
- 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
- Office hours start

Key concepts from Tuesday

- Introduction to this course
  - Object-oriented programming (via Java)
  - Design
  - Design
  - Design
  - Concurrency
  - Real-world tools, real-world skills
- Course infrastructure
  - Git, GitHub, Gradle, Travis-CI

Key to design: Evaluation of alternatives

Version A:

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

Version B':

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[j]);
  …
}
```

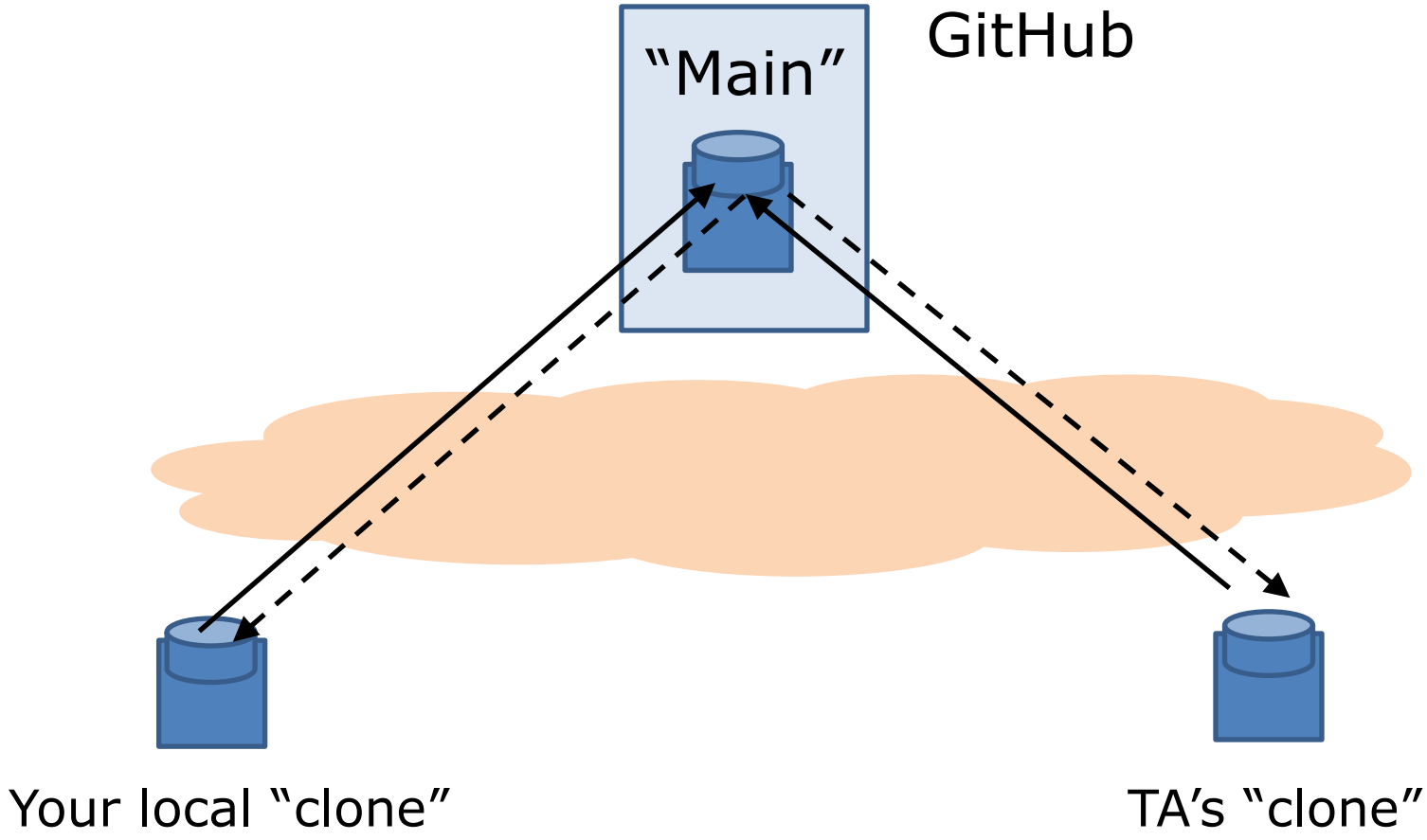
Metrics of software quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency / functional correctness
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous event … Recovers from all anomalous events
- Flexibility
  - Must be replaced entirely if spec changes … Easily adaptable to changes
- Reusability
  - Cannot be used in another application … Usable without modification
- Efficiency
  - Fails to satisfy speed or storage requirement … satisfies requirements
- Scalability
  - Cannot be used as the basis of a larger version … is basis for much larger version…
- Security
  - Security not accounted for at all … No manner of breaching security is known

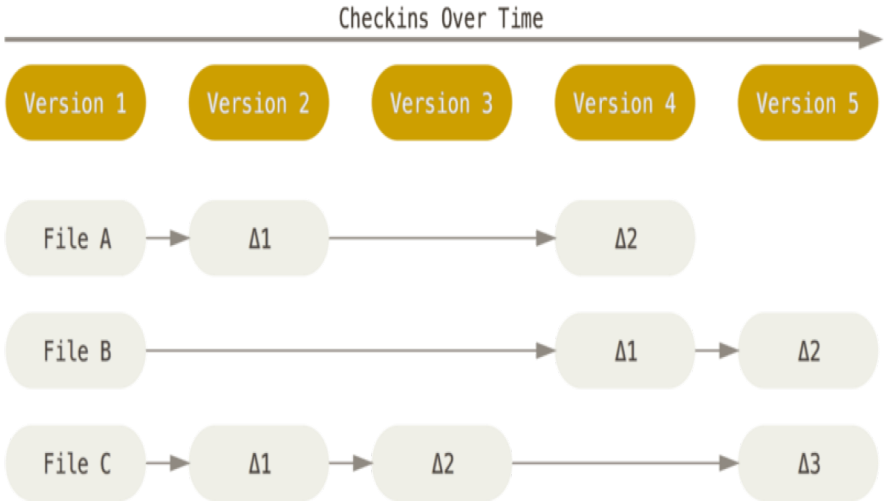
Design challenges/goals

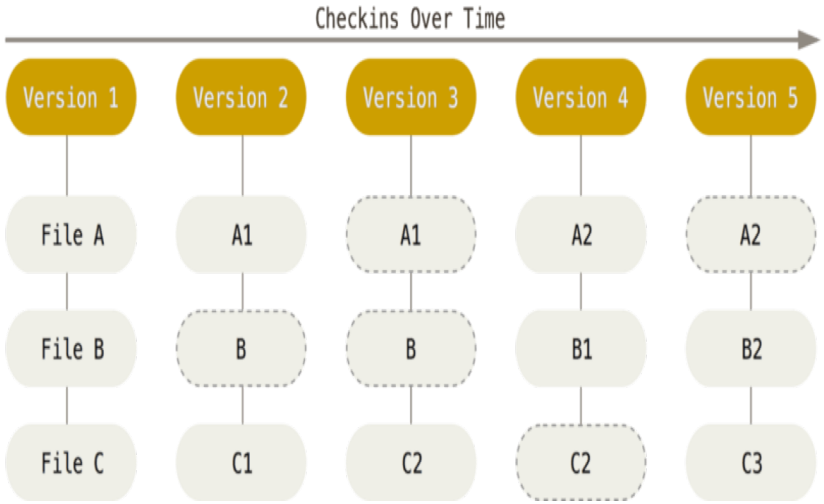
Version control (git): 214 workflow



You *push* homework solutions; *pull* recitations, homework assignments, grades. TAs vice versa

SVN (left) vs. Git (right)



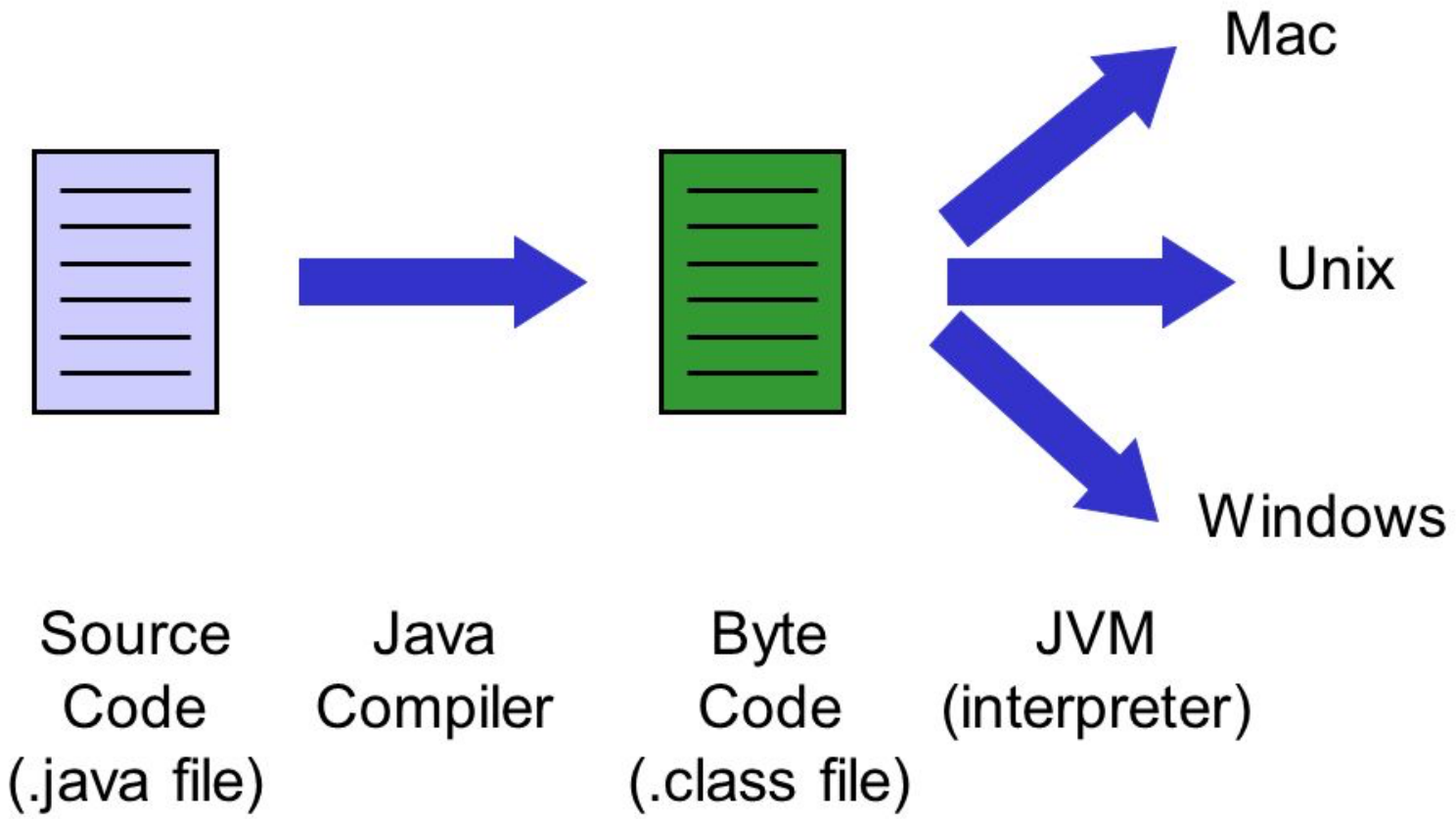


- SVN stores changes to a base version of each file
- Version numbers (1, 2, 3, …) are increased by one after each commit

- Git stores each version as a snapshot
- If files have not changed, only a link to the previous file is stored
- Each version is referred by the SHA-1 hash of the contents

https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Java virtual machine



http://images.slideplayer.com/21/6322821/slides/slide_9.jpg

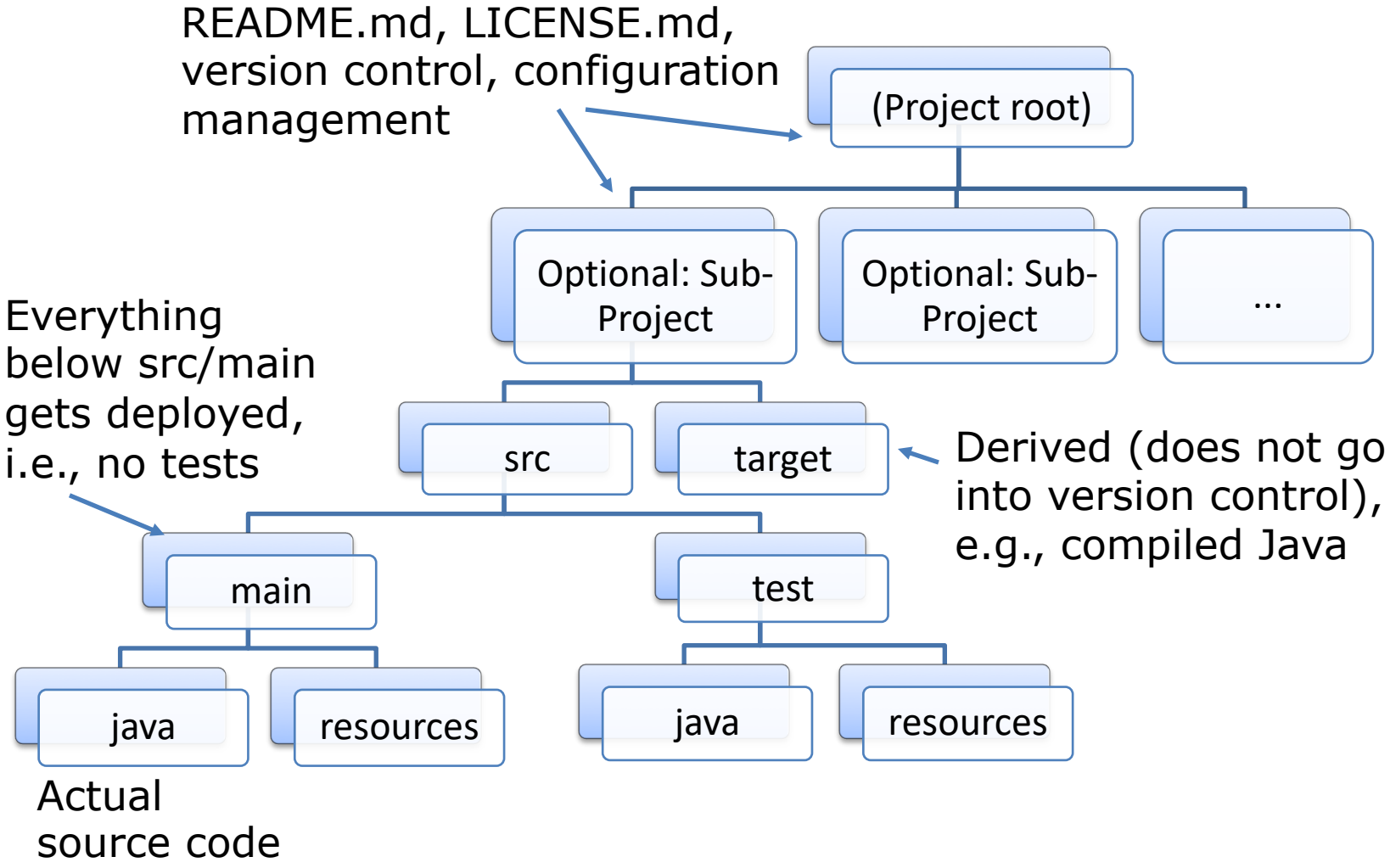
Build Manager

- Tool for scripting the automated steps required to produce a software artifact, e.g.:
  - Compile Java source files into class files
  - Compile Java test files
  - Run JUnit tests
  - If all tests pass, package compiled classes into .jar file.

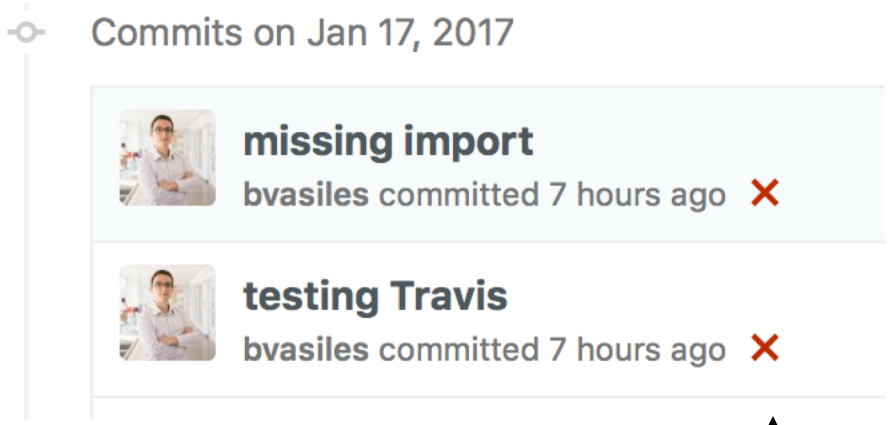
Types of Build Managers

- IDE project managers (limited functionality)
- Dependency-Based Managers
  - Make (1977)
- Task-Based Managers
  - Ant (2000)
  - Maven (2002)
  - Ivy (2004)
  - **Gradle** (2012)

Organizing a Java Project



Travis CI



- Cloud-based CI service; GitHub integration
  - Listens to *push* events and *pull request* events and starts "build" automatically
  - Runs in virtual machine / Docker container
  - Notifies submitter of outcome; sets GitHub flag
- Setup: project top-level folder .travis.yml
  - Specifies which environments to test in (e.g., jdk versions)

You will need for homework 1

- Java (+Eclipse/IntelliJ)

- Version control: Git
- Hosting: GitHub

- Build manager: Gradle
- Continuous integration service: Travis-CI

Today

- Introduction to Java
- Information hiding: Design for change, design for reuse
  - Encapsulation: Visibility modifiers in Java
  - Interface types vs. class types

Introduction to Java - Outline

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

The “simplest” Java Program

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

The “simplest” Java Program

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

- Complication: you must use a class even if you aren’t doing OO programming

The “simplest” Java Program

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

- Every application must contain a `main` method
- Entry point to the program
- Always “`public static void main`”

The “simplest” Java Program

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

Who can “see” (call) the method. More later.

Whether it’s shared by whole class or it’s different for each class instance (object). More later.

Return type.

The “simplest” Java Program

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

- Complication: `main` must declare command line args even if unused

The “simplest” Java Program

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

- Uses the `System` class from the core library to print the "Hello world!" message to standard output (console).

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
VI. Exceptions

Java type system

- **Primitive** types (no identity except their value):
  - `int`, `long`, `byte`, `short`, `char`, `float`, `double`, `boolean`
- **Object Reference** types (identity distinct from value; all non-primitives are objects):
  - Classes, interfaces, arrays, enums, annotations

- "Using" primitives in contexts requiring objects (canonical example is **collections**) :
  - `Boolean`, `Integer`, `Short`, `Long`, `Character`, `Float`, `Double`
  - **Don't use unless you have to!**

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`

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
```

Simpler still …

```
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
- 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
VI. Exceptions

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
```

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
```

Aside: “%n” vs “\n”?

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 The quick brown fox jumps over the lazy
  dog
The quick brown fox jumps over the lazy dog
```

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 (default delimiter: whitespace)

```
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
```

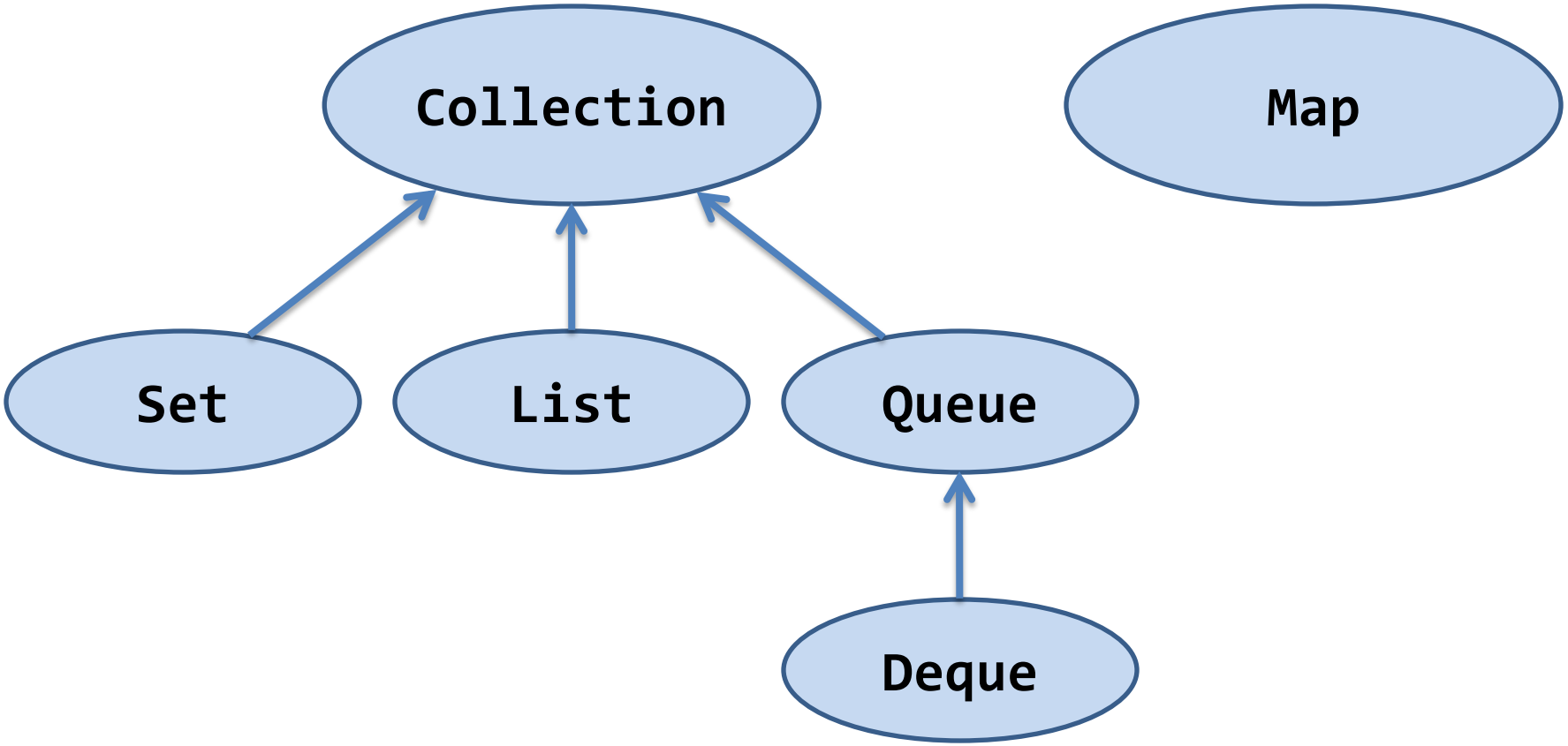
Outline

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

Java Collections

- A collection (**container**) groups multiple elements into a single unit.
- **Java Collections Framework**:
  - Coupled set of classes and interfaces that implement common collection **data structures**.
  - Includes **algorithms** (e.g., searching, sorting).
    - algorithms are *polymorphic*: can be used on many different implementations of collection interfaces.

Primary collection interfaces



Traversing collections

- Using **iterators**

```
Iterator<E> it = collection.iterator();
while (it.hasNext()){
    System.out.println(it.next());
}
```

`next()` returns current element (initially first element); then steps to next element and makes it the current element.

- Using **for-each** (compiles to iterator)

```
for (Object o : collection)
    System.out.println(o);
```

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`

Aside: Java's built-in class library

- `java.lang`: Many basic tools, library features
- `java.util`: Data structures and algorithms, other utilities
- `java.io`: Input/output
- `java.net`: Networking
- …

Outline

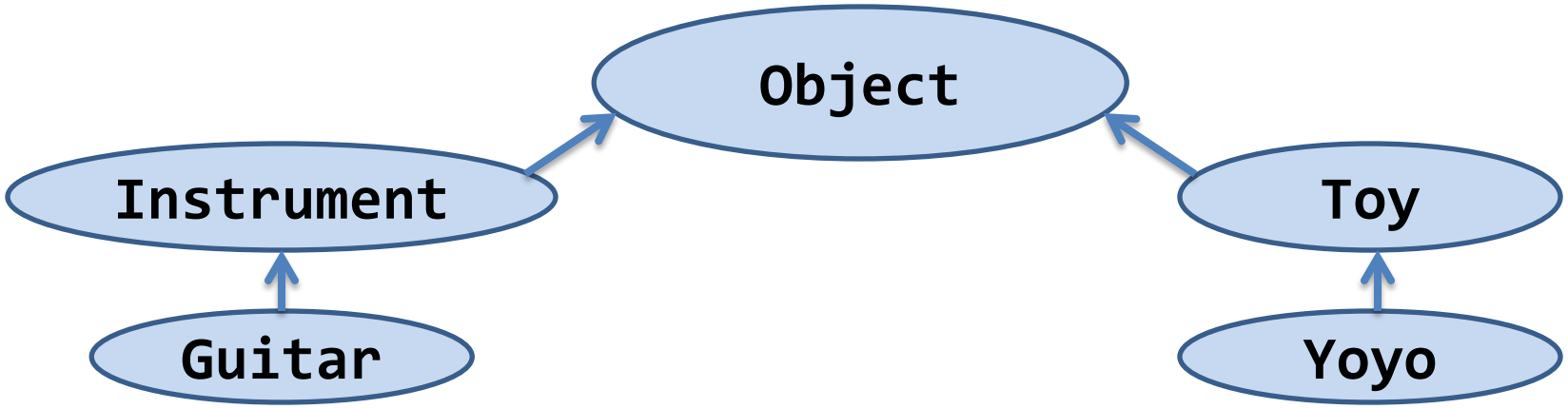
I. “Hello World!” explained
II. The type system
III. Quick ‘n’ dirty I/O
IV. Collections
V. Methods common to all Objects
VI. Exceptions

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



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
  - It's easy to get it wrong
- But often you don't want identity semantics, but *equality*
- 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`

Outline

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

What does this code do?

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
  switch (errno) {
  case _ENOFILE:
     System.err.println("File not found: " + …);
     return -1;
  default:
     System.err.println("Something else bad happened: " + …);
     return -1;
  }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
  System.err.println("Unknown internal error.");
  return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
  System.err.println("Error reading binary data from file");
  return -1;
}  // The Slide lacks space to close the file.  Oh well.
return i;
```

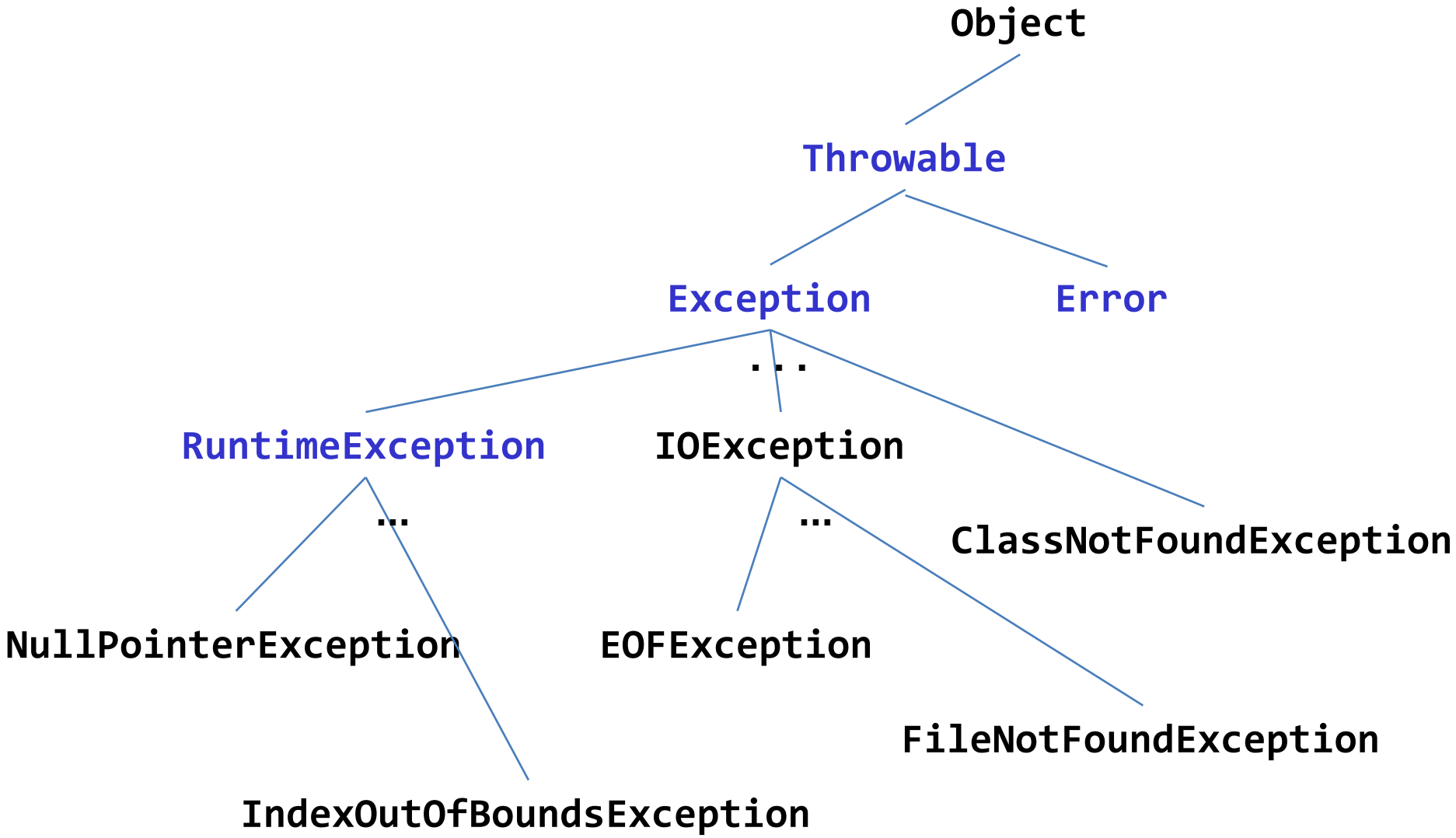
Compare to:

```
FileInputStream fileInput = null;
try {
    fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + fileName);
} catch (IOException e) {
    System.out.println("Couldn't read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```

Exceptions

- Notify the caller of an exceptional condition by automatic transfer of control
- Semantics:
  - Propagates up stack until `main` method is reached (terminates program), or exception is caught

The exception hierarchy in Java



Checked vs. unchecked exceptions

- Checked exception
  - Must be caught or propagated, or program won't compile
- Unchecked exception
  - No action is required for program to compile
  - But uncaught exception will cause program to fail!

Control-flow of exceptions

```
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds");
    }
}
```

Control-flow of exceptions

```
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds");
    }
}
```

Handle errors at a level you choose, not necessarily in the low-level methods where they originally occur.

Creating and throwing your own exceptions

```
public class SpanishInquisitionException extends
   RuntimeException {
     public SpanishInquisitionException() {
     }
}

public class HolyGrail {
     public void seek() {
          ...
          if (heresyByWord() || heresyByDeed())
               throw new SpanishInquisitionException();
          ...
     }
}
```

Benefits of exceptions

- You can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Provide high-level summary of error, and stack trace
  - Compare: core dump in C
- Improve code structure
  - Separate normal code path from exceptional
  - Ease task of recovering from failure
- Ease task of writing robust, maintainable code

Introduction to Java Summary

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

Today

- Introduction to Java
- Information hiding: Design for change, design for reuse
  - Encapsulation: Visibility modifiers in Java
  - Interface types vs. class types

Visibility modifiers in Java ("encapsulation")

- `private`: Accessible only from declaring class
- "package private": Accessible from any class in package
  - a.k.a. default access, no visibility modifier
- `protected`: Accessible from package and also from subclasses
- `public`: Accessible anywhere

Visibility modifier example

- Consider:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x;
        y += p.y;
    }
}
```

Visibility modifier example

- Consider:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public void translateBy(Point p) {
        x += p.x; // This is OK.  p.x and p.y are
        y += p.y; // accessible from the Point class!
    }
    public double getX() { return x; }
    public double getY() { return y; }
}
```

More next week

Summary

- Java's bipartite type system: primitives and object references
- Collections framework is powerful and easy to use
- Information hiding is a key design principle for reuse, change
  - Encapsulation via limiting visibility of methods and fields
  - Interfaces define expectations, support reuse and change