(A Woefully Incomplete) Introduction To Java

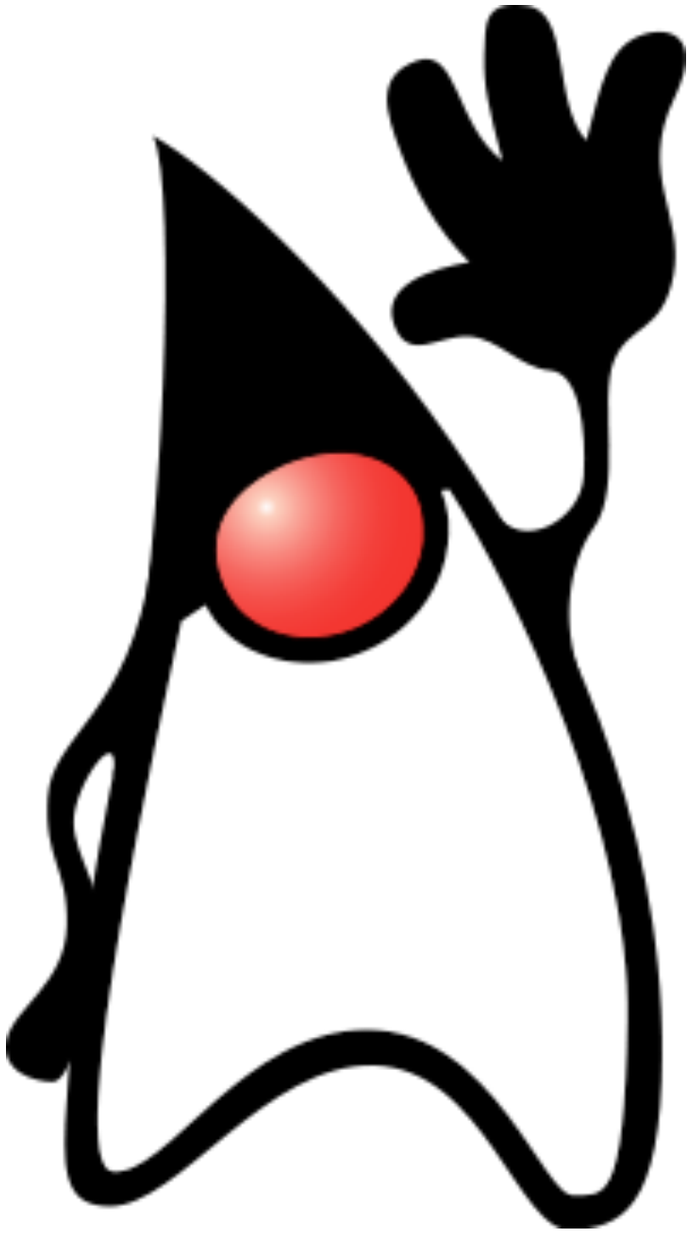
CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 11 — 10/02/2012

Goals of the Lecture

- Present an introduction to the Java programming language
- Coverage of the language will be INCOMPLETE
  - Some topics are selected due to their relationship to creating mobile applications using the Android framework
- Not covering
  - statements, conditional constructs, object instantiation, data types, how to create generics, etc.
- For those, see Head First Java or Thinking in Java

History

- Java got started as a project in 1990 to create a programming language that could function as an alternative to the C and C++ programming languages
  - It made its big splash in 1994 when an alpha release was created that allowed Duke (Java's Mascot) to be animated within a Web browser
- This was a smart move; Java's Applet framework was just a move to get developers to try out the language



History: Simpler and Safer

- In 1994, I attended several talks by one of Java's main designers, James Gosling, talking about the new language
    - I was in graduate school at the time; I liked what I saw and I was building research prototypes in Java by the end of 1994; switched from Ada (!)
- In his talks, Gosling emphasized that Java
    - was both **simpler** and **safer** than C++
- In addition, he talked about "write once, run anywhere"
    - Java code could run wherever a Java Virtual Machine (JVM) existed
        - (at least that was the theory)

Simpler

- Java was simpler than C++ because
  - it removed language features that added complexity or were easily misused
    - pointers and pointer arithmetic
    - the notion of “friend” classes
    - ability to define new operators
  - no explicit memory management due to garbage collector
- Java had no “history”; no need for backwards compatibility (back in 1994!)

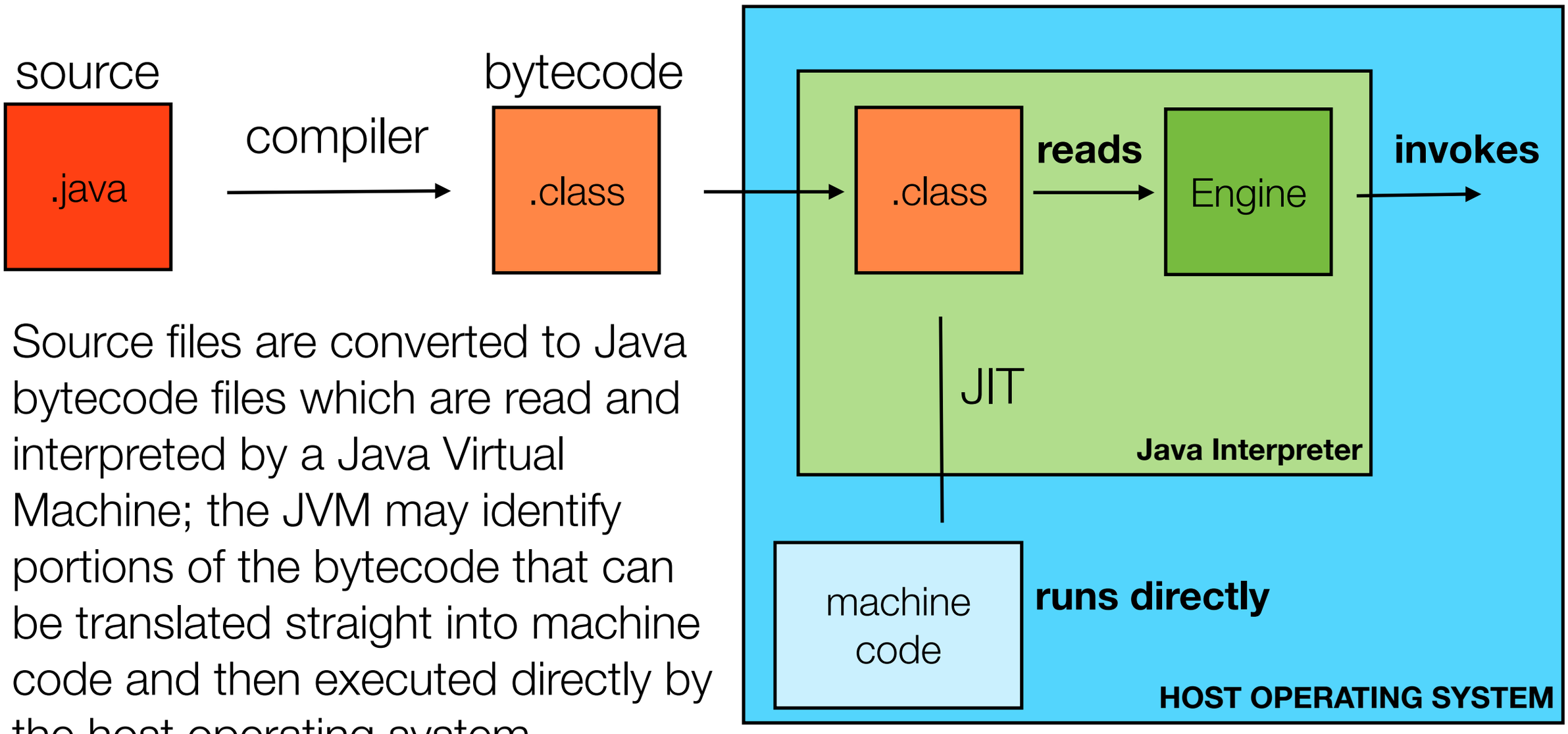
Safer

- Java was safer than C++ because
  - it was interpreted (runs in a protected virtual machine)
  - code downloaded from elsewhere was sandboxed
    - e.g. applet code could not access the host machine except in very clearly defined ways
  - built-in bounds checking and no pointers made it more difficult for malicious code to be written to “hack” the language’s run-time system

Support for Object Orientation

- More importantly for us
  - Java had a clean object model while still providing access to primitive types (ints, floats, etc.)
    - (Hybrid approach was adopted for performance reasons that are now largely obsolete)
  - Single inheritance object-oriented model plus interfaces
  - Extensive class library
    - lots of classes to create objects we can use in our own code

Java is interpreted



Source files are converted to Java bytecode files which are read and interpreted by a Java Virtual Machine; the JVM may identify portions of the bytecode that can be translated straight into machine code and then executed directly by the host operating system

Java Performance

- Java suffered performance problems for many years when compared with code in other languages that had been directly compiled for a particular OS/ machine
    - Now, extensive use of “just in time” compilation has largely eliminated these concerns
        - Java provides excellent performance for many frameworks across many domains
            - provides native code interface (access to C libraries) to gain additional speed if needed
- Minecraft, for instance, is Java + OpenGL

Fundamentals

- A Java program at its simplest is a collection of objects and a main program
    - The main program creates an object or two and sends messages to those objects to get the ball rolling
        - These objects communicate with more objects to achieve the objectives of the program
- You typically have a non-OO main routine that bootstraps objects;
    - you are then in OO land until the end of the program

Hello World

```
1  public class HelloWorld {
2
3    public static void main(String[] args) {
4      System.out.println("Hello, world!");
5    }
6
7  }
8
```

Anatomy (I)

Public class HelloWorld is contained in a file **HelloWorld.java**

```
1  public class HelloWorld {
2
3    public static void main(String[] args) {
4      System.out.println("Hello, world!");
5    }
6
7  }
8
```

Compiling this file produces a new file called **HelloWorld.class**

If Java's interpreter is passed the name HelloWorld, it looks for that name's associated class file and then looks for a static method called main that takes an array of strings; execution begins with the first line of that routine after any static init code

Anatomy (II)

```
1  public class HelloWorld {
2
3    public static void main(String[] args) {
4      System.out.println("Hello, world!");
5    }
6
7  }
8
```

main() is static because at the start of execution, no program-supplied objects have been created… there is no instance of HelloWorld to invoke methods on. Instead, the interpreter reads in the class, locates the statically available main() method, and invokes it

Anatomy (III)

```
1  public class HelloWorld {
2
3    public void sayItLikeYouMeanIt() {
4      System.out.println("HELLO, WORLD!");
5    }
6
7    public static void main(String[] args) {
8      System.out.println("Hello, world!");
9      // sayItLikeYouMeanIt();
10     HelloWorld hello = new HelloWorld();
11     hello.sayItLikeYouMeanIt();
12
13
14   }
15
16 }
17
```

This is a non-static method

You can't call a non-static method from a static method unless you have an object

Packages

- One mechanism for grouping classes is known as the **package**
- A class can declare itself to be part of a package with the package keyword followed by a dotted name, for example
  - package mypack.foo;
- The previous statement declares that there is a top-level package called “mypack” that contains a sub-package called “foo”. The class that uses this statement is a member of package “foo”
  - Top level packages “java” and “javax” are reserved;
  - java has some conventions around package names (see below)

Big Systems (I)

- Packages enable the creation of large-scale software systems written in Java;
- They prevent name clashes
  - import foo.Employee;
  - import bar.Employee;
- These two import statements refer to **two separate classes** both named Employee. They can both be used by the same program as long as you use the full class name

Big Systems (II)

- In addition, to preventing name clashes, packages allow multiple classes to be deployed as a group
- They do this via a combination of two things
  - a mapping between package names and the file system
  - the ability to store a snapshot of a file system in a single file (known as a .jar file, a Java ARchive)
- A jar file can be handed to a Java interpreter; it can read the file, reconstruct the snapshot and access/execute Java classes stored within the snapshot at runtime

Example

- Let’s return to the Student example featured in Lecture 2
- Demo
  - example_0: original program
  - example_1: program with all classes in packages
  - example_2: Test program not in package; accesses all other classes via a jar file
  - example_3: Let you’re IDE handle it!

Discussion of Example (I)

- Files that declare themselves to be in a package need to be physically located in a file system that mimics the package structure
    - Thus the class Student.java was located in a directory called “students” that was itself in a directory called “ken” because Student declared itself to be in the package “ken.students”
- To access files in a package, we need to use an import statement. A “*” in an import statement pulls in all classes contained within that package: import ken.students.*;

The import Statement

- The import statement brings names of classes into scope
  - import foo.bar.Baz;
- The above makes Baz visible; we can then say things like
  - Baz b = new Baz();
- We can skip the import statement but then we would say
  - foo.bar.Baz b = new foo.bar.Baz();
- Classes in the same package can “see” each other with no need for an import statement

Discussion of Example (II)

- The Java compiler creates .class files such that they mimic the structure of their packages
  - Student.class was placed in the “ken/students” directory
- We can have the Java compiler keep .java and .class files separate by passing the name of a directory to the -d command line argument (as we saw in example_2)
  - Conventions dictate the use of “src”, “build” and “lib” directories to store .java, .class and .jar files

Discussion of Example (III)

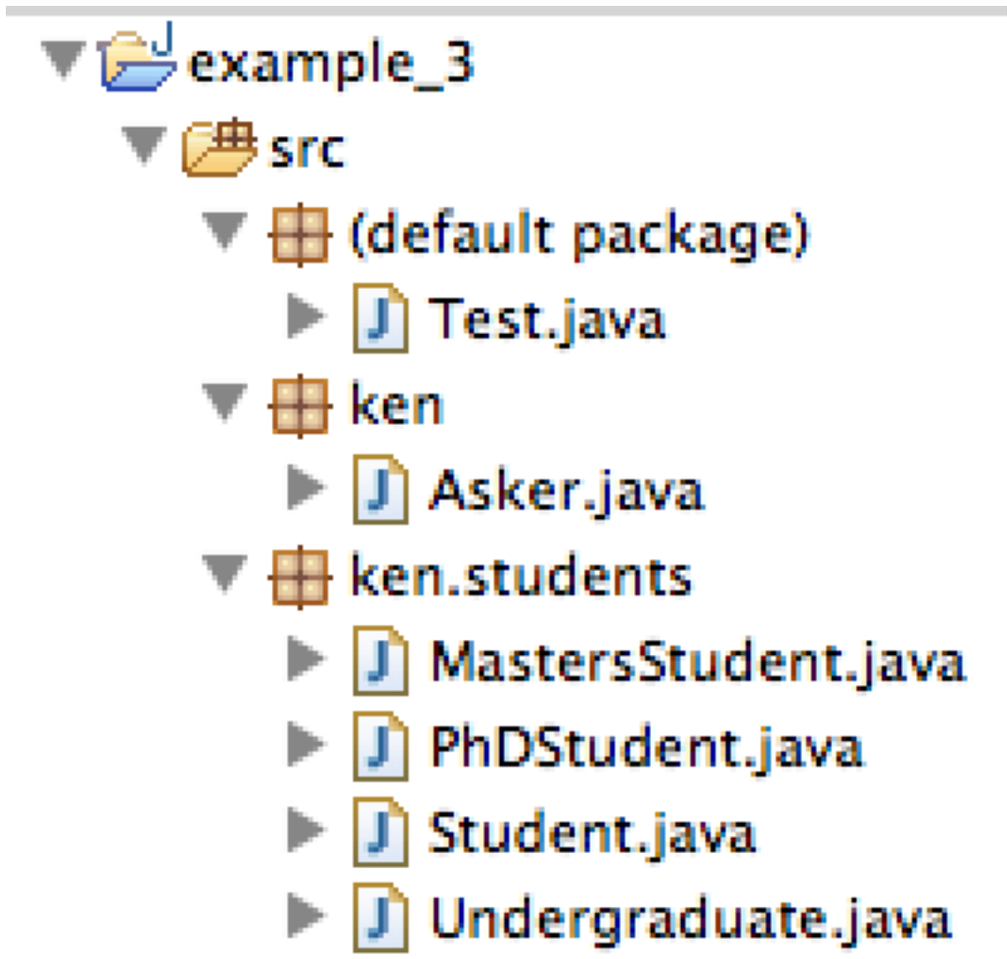
- The jar command creates .jar files using a syntax that is similar to the Unix "tar" command
  - jar cvf <file.jar> directory
    - create a .jar file containing the contents of directory
  - jar tvf <file.jar>
    - prints a "table of contents" for .jar file

Discussion of Example (IV)

- The -classpath or -cp option is used by
  - javac: to allow files to access the classes in a jar file during compilation
  - java: to locate classes in jar files that are needed during runtime
- When used with the “java” command we must be sure to include all directories that we need, including the “current directory”, in order to ensure that the interpreter can find everything it needs to execute the given program

Discussion of Example (V)

- Finally, you can opt to ignore all of these details and just let your IDE handle it
- Here's a screenshot of example_2 set-up in Eclipse
- (now named example_3)



Discussion of Example (VI)

- And, here's a snapshot of the file system that Eclipse automatically creates for this project
  - based on the package statements contained in our source files

```
engr2-1-197-173-dhcp:workspace $ tree example_3
example_3
├── bin
│   ├── Test.class
│   └── ken
│       ├── Asker.class
│       └── students
│           ├── MastersStudent.class
│           ├── PhDStudent.class
│           ├── Student.class
│           └── Undergraduate.class
└── src
    ├── Test.java
    └── ken
        ├── Asker.java
        └── students
            ├── MastersStudent.java
            ├── PhDStudent.java
            ├── Student.java
            └── Undergraduate.java
```

Returning to HelloWorld

```
1  public class HelloWorld {
2
3    public static void main(String[] args) {
4      System.out.println("Hello, world!");
5    }
6
7  }
8
```

How does this work? What is System? What is “out”? System looks like a package name; out looks like an object reference that responds to the method println()

Discussion

- System is, in fact, an object; Google “java.lang.System”
- out is also an object. It provides methods for printing to stdout
- It turns out that System is a member of a package called java.lang
  - If that’s the case, why did the compiler let us access System without an import statement?: i.e., import java.lang.System
    - The answer is that everything in java.lang is automatically imported into all Java programs

Classes (I)

- You define a class in Java like this
  - **public class Employee {**
- This is the same as saying
  - **public class Employee extends java.lang.Object {**

- How do we know?
  - Well, you use to be able to examine the output of the command
    - “javap HelloWorld”

Classes (II)

- The output of javap **used** to look like this; it was “fixed” in Java 7.
- $ javap HelloWorld
  - Compiled from "HelloWorld.java"
  - public class HelloWorld extends java.lang.Object {
    - public HelloWorld();
    - public void sayItLikeYouMeanIt();
    - public static void main(java.lang.String[]);
  - }

Even though we didn’t say it explicitly, HelloWorld extends java.lang.Object by default

Everything is a java.lang.Object

- All classes in Java extend from java.lang.Object
  - It defines a set of methods that can be invoked on any Java object, including arrays
    - (Listed on next slide)
  - For details
    - Java 7 Documentation of java.lang.Object
    - Full Java 7 API

java.lang.Object's methods

- Copying objects
  - clone()
- Equality and Identity
  - equals(Object)
  - hashCode()
- Garbage Collection
  - finalize()
- Reflection
  - getClass()
- Threading
  - notify(), notifyAll(), wait()
- Printing/Debugging
  - toString()

Classes (III)

- After java.lang.Object, classes can extend via single inheritance
  - All Java classes have one and only one parent
- To allow a class to have more flexibility with respect to its type, Java provides the notion of interfaces
  - public class Dog extends Canine implements Pet {
- This says that Dog IS-A Canine but can also act as a Pet
  - Multiple interface names can appear after “implements” separated by commas “Pet, BedWarmer, BottomlessPit”

Classes (IV)

- When a class implements an interface, the compiler requires that all methods defined in the interface appear in the class; if not the class must be declared abstract and a subclass must implement the missing methods
  - public interface Pet {
  - public void takeForWalk();
  - }
- In this example, Dog must provide an implementation of takeForWalk() or else be declared abstract

Classes (V)

- Constructors
  - When a new instance of a class is created, its constructor is called to perform initialization
  - A constructor looks like a method that has the same name as the class but with no return type specified
- If you do not define a constructor, Java creates a default constructor with no arguments
  - This constructor simply calls the default constructor of the superclass

Constructors (I)

- These are equivalent
- public class Foo {
- }

- public class Foo {
  - public Foo() {
    - super();
  - }
- }

Constructors (II)

- The purpose of constructors is to initialize an object
  - The JVM does some initialization for you
    - It will set all attributes to default values
      - primitive types (int, float, etc.) get set to zero
      - reference types (classes) get set to null
- The constructor is then called to do any other initialization that you need

Constructors (III)

- Constructors can have arguments
    - (as we've seen in various examples this semester)
- If you want to use one of these you simply pass in the arguments when creating a new object
    - public PhDStudent(String name) {
- is invoked with the call
    - PhDStudent Gandalf = new PhDStudent("Gandalf");

Constructors (IV)

- Funny (Difficult) Rules about Constructors
  - If you don't define one, Java creates the default one
    - If you do define one, Java doesn't create a default one
  - If you don't call super() on the very first line of the constructor, then Java inserts a call to the default constructor of the superclass
    - If you do call super() or one of the other constructors of the superclass, Java doesn't insert such a call

Anonymous Classes (I)

- Sometimes you need to define a class “on the fly” to specify what happens when a particular event occurs
  - Common when implementing a graphical user interface
    - A button gets clicked and we need an instance of java.awt.event.ActionListener to handle the event
    - We could implement this handler in a separate file a class that implements ActionListener that specifies what to do
    - We would then create an instance of that class and associate it with the button

Anonymous Classes (II)

- The problem?
  - What if you have 10 buttons that all require different implementations of ActionListeners; you would have to create 10 different .java files to specify all the logic
    - This is not scalable
- The solution
  - Anonymous Classes
  - We create the ActionListener instance on the fly

Demo

Discussion of Example (I)

- This simple example of using two anonymous classes demonstrated a lot of interesting things
  - Anonymous classes are defined “in line” by saying first
    - new
  - because we are both defining a class AND creating an instance of it; we then provide
    - a classname or interface name with parens and an open bracket
- followed by method defs and a closing bracket

Discussion of Example (II)

- The compiler will then
  - define a new class,
  - compile it to bytecode
  - AND at run-time the interpreter will create an instance of this unnamed (i.e. anonymous) class
- It does this because this in-line definition occurs inside a method call
  - button.setActionListener(<ANONYMOUS CLASS>);
- or
  - button.setActionListener(new ActionListener() { … });

Discussion of Example (III)

- Where do these new classes get stored?
  - In the same directory that all the other .class files go
- Directory of the “with” example before we compile
  - ButtonExample.java
- Directory after we compile
  - ButtonExample$1.class
  - ButtonExample$2.class
  - ButtonExample.class
  - ButtonExample.java

Which is which? Use javap to find out

- The $1 and $2 classes are the autogenerated anonymous classes

Discussion of Example (IV)

- Finally, look at what we had to do when we decided to implement the same program without using anonymous classes
  - We had to create one standalone class that implements ActionListener and one that implements Runnable
  - We had to change the structure of the main program
    - We had to instantiate each of the standalone classes, initialize them, and plug them in to the appropriate places
- Anonymous classes are simpler, more compact and more expressive of our intentions

How to use Java Generics

- Java provides a way to do “generic” data structures
- The idea is fairly simple
  - In procedural languages, we use to have to implement collections like this
    - List of String, List of Integer, List of Employee
      - Each list (or data structure) was written with a specific type of content in mind
- This is silly since the API and semantics of the data structure are independent of its contents

Java Generics

- Take a look at the definition of the List interface in java.util
    - Java documentation for List
- public interface List<E> extends Collection<E> {
    - boolean add(E e);
    - …
    - E get(int index);
    - …
- }

What's with the E?

- The E (which may stand for "element") is a placeholder that says
  - We are defining the API for a List that contains elements of type E
  - If I add() an E, I can get() that E back
- Specifically, E is a placeholder for a type

Example: List of String

- I can create a List that holds Strings
  - List<String> strings = new LinkedList<String>();
- Passing “String” inside of the angle brackets, tells the interpreter to create a version of List where “E” gets replaced by “String”
- Thus
  - boolean add(E e);
- becomes
  - boolean add(String e);

List of List of String

- If I wanted a list in which each element is itself a List of Strings, I can now easily do that:
- List<LinkedList<String>> crazy_list =
  - new LinkedList<LinkedList<String>>();
- In this case E equals “LinkedList<String>” and get() would become
  - LinkedList<String> get(int index);
- meaning when I call get() on crazy_list, I get back a LinkedList that in turn contains Strings

Generic Map

- You should now understand the interface to Map
  - Java documentation for java.util.Map
- public interface Map<K,V> {
  - V get(Object key)
  - V put(K key, V value)
  - …
- }

Quick Example

- Map<String, Integer> ages = new HashMap<String,Integer>();
- ages.put(“Max”, 20);
- ages.put(“Miles”, 30);
- int ageOfMax = ages.get(“Max”);
- System.out.println(“Age of Max: ” + ageOfMax);

- Produces: Age of Max: 20
- Note: “autoboxing” of int and Integer values

Wrapping Up

- Java Fundamentals
  - Relationship of .java to .class to .jar
  - Packages; relationship to file system and .jar files
  - Classes, constructors, interfaces
    - anonymous classes
    - how to use generics

Coming Up Next

- Lecture 12: Introduction to Android
- Lecture 13: Introduction to Objective-C