Java and OO, Part 2

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 11A — 03/11/19

Task number one

- If you have your note card from the first class, please place it in front of you…
- If not…

- Get a marker and a piece of card stock
- Fold it in half to make a little tent
- Write the name you'd like to be called on that
- Face the name towards me
- Try to remember to bring that to class for the next several weeks

Acknowledgement & Materials Copyright

- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Goals of the Lecture

- Present a brief introduction to the Java programming language
- Coverage of the language will be INCOMPLETE
- Goal is to highlight some of Java's interesting and OO features

Happy International Women's Day!

https://insights.dice.com/2016/03/14/10-famous-women-in-tech-history/

Sister Mary Kenneth Keller: First Female Computer Science PhD, many others…

Disclaimer

- Although I've been writing a bit of Java now and again for years, I do not use it regularly and I am far from an expert
- Many of you likely have more recent experience with the language than I do
- Please feel free to ask questions, provide corrections or additions, or share other thoughts on the lecture topics with the class as we move through the review
- If I can't answer questions today, I'll get answers back to you

References

Some of the better Java books that I have:

- Head First Java by Sierra & Bates
- Think Java by Downey & Mayfield
  - Free at http://greenteapress.com/thinkjava/
- Effective Java by Bloch
- Thinking in Java by Eckel
- Java by Example by Jackson & McClellan

Also:

- Java Network Programming by Hughes et al.
- Java Network Programming by Harold

Plus hosts of on-line resources and tutorials, like:

- Oracle's Java SE and EE Documentation and OpenJDK Documentation
  - https://docs.oracle.com/javase/tutorial/
- StackOverflow.com
- Java.net

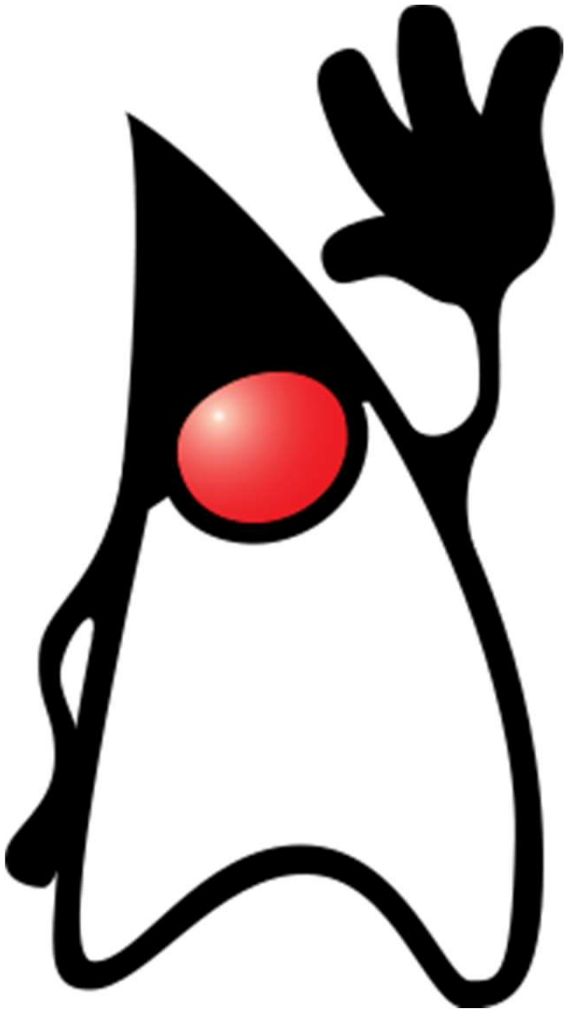
Also just happened to hit a nice article (with build-along examples) on Java (and the Spring framework) in Code magazine by Ted Neward, available online at: https://www.codemag.com/Article/1811081/Java

Java Tools

- Language
  - Open source Java SE implementation – https://openjdk.java.net/
  - ~~Sun's~~ Oracle's licensed Java – https://www.java.com/en/
- IDE
  - Many choices…
  - IntelliJ IDEA – JetBrains – Open Source and Commercial versions – https://www.jetbrains.com/idea/download/
  - Eclipse - https://www.eclipse.org/downloads/packages/release/2018-12/r/eclipse-ide-java-developers
- Library Repository/Tools
  - JHipster – free application generator
  - Apache Maven – build tool and library repository
  - Apache Commons – repo of reusable Java components

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
- The first stable release JDK 1.0 came out in January 1996
- In 2004, Java J2SE 1.5 turned into Java SE 5, a new numbering scheme for releases
- Java SE 8 was released in March 2014 as an LTS (long term support) version that is supported through March 2025
  - I'm using Java 8 for this lecture
- Java SE 11 LTS was released in September 2018 with support through September 2026



History: Simpler and Safer

- James Gosling, one of Java's main designers at Sun, emphasized that Java was both **simpler** and **safer** than C++
- 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 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
- In addition, Java's goal was "write once, run anywhere"
  - The theory being that Java code could run wherever a Java Virtual Machine (JVM) existed, including for embedded devices

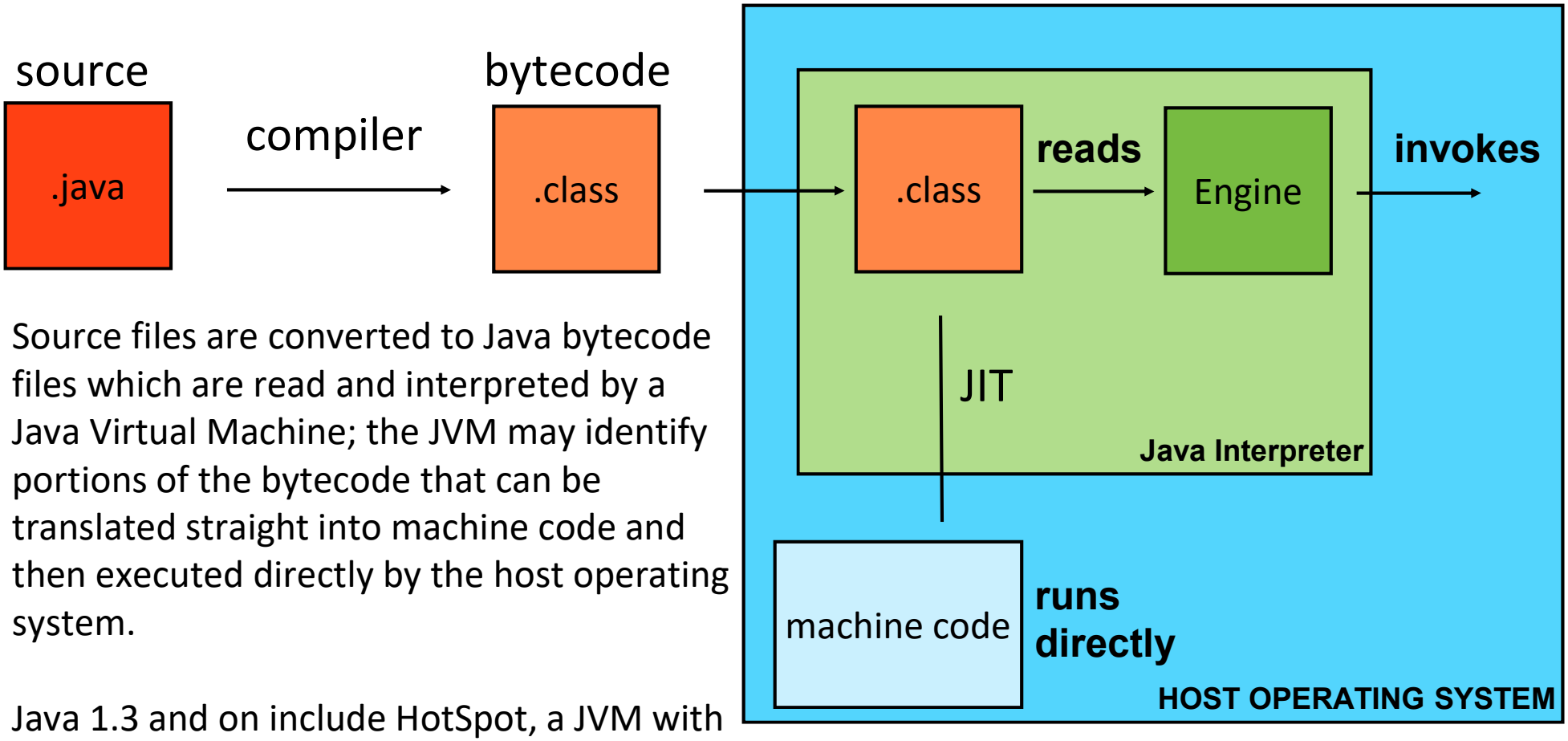
Support for Object Orientation

- More importantly for us
  - Java has a clean object model while still providing access to primitive types (ints, floats, etc.)
    - The initial hybrid compile/interpret 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
- Great language to clearly see OO in practice
  - Exception – multiple inheritance

Opinion: Readability Matters

- Another reason I like Java is it's generally very readable code
  - This makes it easier to understand, review, and maintain other's code
  - A goal of language design standards in organizations and standard tools
  - Related to "egoless programming"
- I like Python for the same reason
- I also like using Lint-type tools to clean code as part of a development cycle
  - Java – see PMD or Checkstyle; Python – pylint, pyFlakes, pycodestyle (pep8)
- C, C++, JavaScript/Node.js… Hmmm.
  - I think those can be MADE readable, but may be a little more work.
- Of course, you can also make unreadable Java and Python too.
- For your reference:
  - Improving readability: https://dzone.com/articles/10-tips-how-to-improve-the-readability-of-your-sof
  - Egoless Programming: https://blog.codinghorror.com/the-ten-commandments-of-egoless-programming/

Java is interpreted – sort of



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.

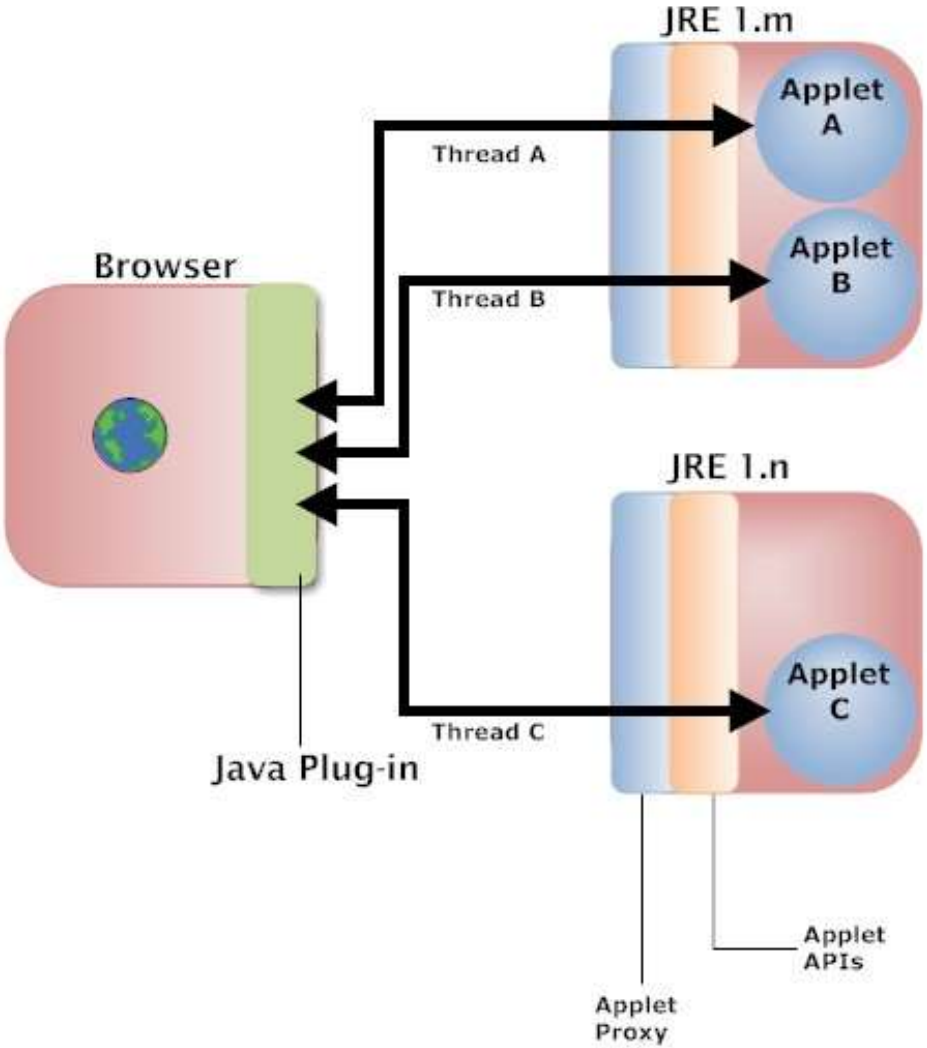
Java 1.3 and on include HotSpot, a JVM with Just In Time (JIT) compilation…

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 example, was originally written using Java, OpenGL, and the Lightweight Java Gaming Library (LWJGL)
    - Follow on versions have mostly been in C++

Java Applets – Rise and Fall

- Java applets were designed for web pages to run Java in a JVM in a browser.
- They required a browser plug-in to run vs. a Java application using a Java Runtime Environment (JRE)
- The Java browser plug-in, over time, proved to have security issues and identified exploits, leading to signed applets
- Java applets were depreciated in Java 9 and are removed from Java 11



https://docs.oracle.com/javase/7/docs/technotes/guides/jweb/applet/applet_execution.html

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

```
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

OO Basics - Inheritance

- 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

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

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
- 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

- 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

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
- 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 (IV)

- IDEs may automatically arrange file structures if not provided
- Here's a snapshot of the file system that Eclipse automatically creates for an example_3 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 {**
- 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.lang.object documentation for Java 8
    - https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html
  - Java 8 API documentation
    - https://docs.oracle.com/javase/8/docs/api/overview-summary.html

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 (II)

- After java.lang.Object, classes can extend via single inheritance
  - All Java classes have one and only one parent
- **Extends** is the inheritance keyword, **implements** is the interface reference keyword
- 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”
- Java also has an operator called instanceof
  - For a Dog object instanceof would return true
    - boolean isDog = aDog instanceof Canine

Classes (III)

- 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

- Note: Java 8 made some changes to interfaces, we'll discuss.

Classes (IV)

- 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
- These are equivalent

```
public class Foo {
}

// or

public class Foo {
   public Foo() {
      super();     //invoke or call parent class constructor
   }
}
```

Constructors (I)

- 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 (II)

- 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 (III)

- 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
- Java is garbage collected – so no destructors, no guarantee when an object will be destroyed
  - You can use a method, finalize, that is called when an object is destructed, but when this happens is not under program control
  - There is a convention of creating a close method to indicate a class is closed and ready to be destructed

Accessibility

- Java supports several levels of accessibility for data and methods in classes
- By default the access is within the package
- Keywords include
  - Private – access only within the original class
    - For data attributes, get and set methods are often used to access information
  - Protected – access within original class and subclasses
  - Public – all access allowed

Java 8 Changes: Interfaces & Lambdas

- Pre-Java 8, Interfaces could only have method declarations
  - This is the traditional no implementation form of an Interface mentioned previously
- Java 8 introduced **default** and **static** methods for Interfaces
- Java 8 also introduced **lambda** expressions, which may change the way some interfaces are implemented

- Interface changes:
  - https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- Lambda example:
  - https://www.geeksforgeeks.org/lambda-expressions-java-8/

Java 8 Changes: Interface Default Methods

- Using the **default** keyword in an interface method definition tells the implementing class it is not mandatory to provide implementation
- Example:

```
public interface Interface1 {

      //to be defined by implementing class
      void method1(String str);

      //provided for implementing class
      default void log(String str) {
            System.out.println("I1 logging::"+str);
      }
}
```

Java 8 Changes: Interface Static Methods

- Using the **static** keyword in an interface method allows an implementing class to call the method directly from the interface, and also allows the interface to use the static method in default method implementations
- Example:

```
public interface MyData {
	default void print(String str) {
		if (!isNull(str))
			System.out.println("MyData Print:" + str);
	}
	static boolean isNull(String str) {
		System.out.println("Interface Null Check");
		return str == null ? true : "".equals(str) ? true :
false;
	}
}
```

- The implementing class could also call the static isNull directly from the interface:

```
boolean result = MyData.isNull("abc");
```

Java 8 Changes: The Diamond Problem

- Because Java allows multiple implementations of Interfaces in a class, it's possible to create a version of the Diamond Problem (usually seen in multiple inheritance)
- For example, if two Interfaces both provided default methods for a method called log, the implementing class would have to override log to avoid a compile time error due to the multiple implementations

```
public class MyClass implements Interface1, Interface2 {
	@Override
	public void log(String str) {
		System.out.println("MyClass logging::"+str);
		Interface1.print("abc"); //static in Interface1
	}
}
```

Java 8 Changes: Lambda Expressions

- Lambda expressions are functional interfaces defined in-line
- They can be passed as objects and can be created without belonging to a class
- Example:

```
interface FunctionInterface
{
    void abstractFun(int x); // An abstract function
}

// Note that class Test is NOT implementing FunctionInterface
class Test
{
    public static void main(String args[])
    {
        // Lambda expression to implement above functional interface
        FunctionInterface fobj = (int x)->System.out.println(2*x);

        // This calls above Lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}
```

Simple Anonymous Class Example

- Allows you to define a class “on the fly” to specify what happens when a particular event occurs

```
public class SuperClass {
  public void doIt() {
    System.out.println("SuperClass doIt()");
  }
}

SuperClass instance = new SuperClass() {
    public void doIt() {
        System.out.println("Anonymous class doIt()");
    }
};

instance.doIt();
```

Anonymous Class Example for Interface

```
public interface MyInterface {
  public void doIt();
}

MyInterface instance = new MyInterface() {
    public void doIt() {
        System.out.println("Anonymous class doIt()");
    }
};

instance.doIt();
```

Anonymous Classes

- 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 as 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
- 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/Example

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 definitions 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.class
  - ButtonExample$2.class ButtonExample.java
- 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 used 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 not strictly necessary 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
    - https://docs.oracle.com/javase/8/docs/api/java/util/List.html

```
public interface List<E> extends Collection<E> {
   boolean add(E e);
   …
   E get(int index);
   …
}
```

- 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
    - https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

```
public interface Map<K,V> {
   V get(Object key)
   V put(K key, V value)
   …
}
```

Map 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
  - Autoboxing = automatic conversion that the **Java** compiler makes between primitive types and their corresponding object wrapper classes

Summary

- Java Fundamentals
  - Relationship of .java to .class to .jar
  - Packages; relationship to file system and .jar files
  - Classes, constructors, interfaces
  - New Java 8 interface elements: default, static
  - New Java 8 lambda expressions
  - Anonymous classes
  - How to use generics

Next Steps

- Optional additional material
  - Dr. Anderson's lecture on Java
    - You can find it on the class Canvas site under Media Gallery, 10 minutes into Lecture 11
    - Again, very similar material as I'm using versions of his slides…
- Next Week
  - Exam grading, project topics and presentation submissions reviews continuing
  - Wednesday 3/13 – Recitation w/Manjunath
  - Lecture for Friday 3/15 - How Experts Design (Chapters 12, 13 in book)
- Things that are due
  - Class Semester Project topic Canvas submission was today
  - Homework 4 – Design for Semester Project – is due Friday 3/22 11 AM
    - Your team can also start to develop preliminary code for your project
    - There will be two homeworks related to code delivery, an interim one on 4/12, and a final delivery on 4/26
- Late Notices

*****Homework 4 and 5 will NOT be accepted after the 1 week late period*****

*****Final Homework 6 submission on 4/26 may NOT be turned in late*****

*****Graduate Presentation submission on 4/15 may NOT be turned in late*****