Today's Lecture

1. `Tksh` (Jeff Korn)

2. Java, from the top down...

Java: What?

- A computer language designed by Sun Microsystems
- First widely-available *distributed* language
- *Object-oriented*, like C++
- *Interpreted*, hence very portable
- *Safe*, supposedly...
- *Secure*, supposedly...

Java: Why?

- Burgeoning Internet
  (currently, the number of bits that can be put on fiber grows faster than processor speed)
- For many, Mosaic first made the Internet usable
- However, static web pages are boring
- When browsing, many cycles to spare
- Why not download code to run into browsers?
- Some things Java enables:
  - more interactive web pages
  - real-time graphics in web pages
  - automatic code distribution
  - dynamic code loading
- High-level applications for the Net have not yet appeared...

Java: Where?

- Must use server `yuma` (Sparc, Solaris)
  (also for Windows 95/NT; coming for most other platforms)
- Java tools (in `yuma:/opt/java/bin`):

  `javac`
  compiler

  `java`
  interpreter

  `javap`
  disassembler

- Later, you will also need:

  `/usr/princeton/new/netscape/netscape`
  Version 2.0 of Netscape

  `appletviewer`
  Testbed for distributed Java applications

Java: Project Structure

Broadly, two sorts of Java programs can be built:

1. Stand-alone applications
2. Applets

The first is like a C program; the second may be shipped across the Net (and run in a browser, for example).

You need to know (1) before (2)

Java's facilities for structuring a project:

- Packages
- Interfaces
- Classes/Objects

Java: Packages

Java uses the filesystem for high-level project structure.

For example, if `-classpath ~cs333/java/classes` is given to the Java compiler, your program has access to the *packages* rooted at `~cs333/java/classes`

We'll delve into classes in a bit – for now just assume a class is a building block that you've designed.

As an example, suppose we are building (a simulator of) a city...

Some packages in our project may be:

```
~cs333/java/classes/city/traffic/stops
~cs333/java/classes/city/traffic/intersections
~cs333/java/classes/city/traffic/streets
...
~cs333/java/classes/city/government/budget
```


Java: Packages (cont'd)

A Java program file declares it to be a package with the `package` statement:

```
package city.commerce.stores;
```

A Java program file may use existing classes (building blocks) through the `import` statement:

```
import city.traffic.*;
```

`package` and `import` statements are relative to the class path.

A common package in Java programs is: `java.lang.*`
(akin to `stdlib.h` in C, for example)

Packages help to logically structure a project by imposing a physical (filesystem) structure on it.

One can write Java programs without using packages, but packages are invaluable in team projects.

Java: Interfaces

An *interface* is a mechanism that communicates the functionality available in a building block (*i.e.* class) for external use.

An *implementation* (or implementations) subsequently implement an interface.

```
public interface ManhattanConnection {
        public void CollectToll(float total);
        public void CloseForRepairs();
}

public interface LincolnTunnel extends ManhattanConnection {
        public final int nTubes = 3;
public float AirQuality();
}

public interface GW_Bridge extends ManhattanConnection {
        public final int SpanInFeet = 3500;
}
```

Note that the *methods* declared in an interface have no body – their implementation is separate.

Interfaces are powerful because they communicate only what's necessary to use a program and (hopefully) nothing more.

One can write Java programs without using interfaces, but interfaces are invaluable in team projects.

Java: Classes

Classes are the primary building block in Java

Class libraries enable code reuse

The Java API is a collection of class libraries

Classes are blueprints

An *instance* of a class is an *object*

A class declaration specifies the functionality (*methods*) and state (*instance variables*) that objects instantiated from it will have.

When you introduce a new class into a program, you in essence introduce a new type.

Classes are organized into a *hierarchy*; a class may have *subclasses* and *superclasses*.

(In practice, one must (unfortunately) extend this simplistic class/object model.)

Java: Sub– and Superclasses

A *subclass* of a class *c* *extends* the class *c*.

Following the blueprint analogy, this amounts to additions to the master blueprint.

Subclasses allow construction of objects with behavior slightly different from an existing class without having to reinvent the new object.

```
class Building extends Object {
        public int nFloors;
        public int nStairCases;

        public void inCaseOfFire() { ringAlarms(1); }
        ...
}

class TallBuilding extends Building {
        public int nElevators;

        public void inCaseOfFire() { ringAlarms(2); }
        ...
}

class SkyScraper extends TallBuilding {
        public int nExpressElevators;

        public void inCaseOfFire() { ringAlarms(10);
                                     notifyMayor(); }
        ...
}
```

A class may *override* methods in its superclasses.

Java: Sub– and Superclasses (cont'd)

The topmost class in the Java hierarchy is called *Object*

All other classes—supplied or user written—are subclasses of `Object`

Equivalently, `Object` is a superclass of every class.

A class is said to be *derived* from its immediate superclass.

Within the code for a class, the keyword `this` refers to the instance of the class (an object) currently executing.

The keyword `super` refers to the object's immediate superclass.

```
class Tabloid extends NewsPaper {
        public int Circulation = 10000;

        public void PrintCopies() {
                if (Television.OnStrike && \
    super.Circulation > this.Circulation) {
                        nPrint(super.Circulation);
                } else {
                        nPrint(this.Circulation);  // 'this' optional
                }
        }
        ...
}
```


Java: A Menagerie of Modifiers

For *methods*:

`public`

  method may be invoked from any other class

`private`

  method may only be invoked from within this class

`protected`

  method may only be invoked from subclasses

`final`

  method may not be overridden

`abstract`

  method must be implemented in a subclass (only the declaration is given in this class)

`static`

  method is not instantiated per object; instead, this method is available directly through the class and is, as such, not associated with an object

Java: A Menagerie of Modifiers (cont'd)

For *variables*:

`public`

variable accessible from other classes

`private`

variable accessible only from within this class

`protected`

variable accessible only from subclasses

`final`

variable may not be changed (*i.e.*, it's a constant)

`static`

variable is not instantiated per object; instead, this variable is available directly through the class and is, as such, not associated with an object

Java: Modifier Example


```
class Lotto extends Revenues {
        protected static double Booty;

        protected static void ChangeBooty(double ChangeInBooty) {
                Booty += ChangeInBooty;
        }

        private int RandomInt(int n) {  // in range 0 to n-1, inclusive
                return (Math.random() * n);
        }

        protected final int RollDie () {
                return RandomInt(6)+1;
        }

        public abstract void LoudAdvertisement(String Slogan);
        ...
}
```


Java: Objects

Objects come into existence when a class is instantiated.

This is accomplished with the keyword `new`:

```
        City Princeton = new City();
        City[] Yuma; /* initialized to null */          City[] NYC = new
City[5];
```

After instantiation, an object may be referenced; *e.g.*,

- `Yuma = new City();`
-

```
for (int i = 0; i < 5; i++) {
        NYC[i] = new City();
        NYC[i].IsBorough = true;
}
```

- `if (Princeton.Population < NYC[0].Population) ···`

Java: Object Constructors

Class instantiation invokes a *constructor* for the class.

Constructors are either explicit or implicit.

Explicit constructors have the same name as their class.

If Java does not find an explicit constructor for the class, it invokes the class's superclass constructor, and so on.

Implicit constructors commence with invocation of the superclass constructors.

```
class City {
        public int Population = 1234;
        public City(int RealPopulation) {
                // <-- implicit call to superclass constructor here
                Population = RealPopulation;
        }
        public City(float AvgPopulation) {
                // <-- implicit call to superclass constructor here
                Population = AvgPopulation;  // coerce to int
        }
        public City() {  // Java provides this by default
                // <-- implicit call to superclass constructor here
        }
        ...
}
```

Constructors (and regular methods) may *overload* their parameter types.

Java selects the appropriate constructor (method) based on the types in the method's call.

Java: Garbage Collection

Java automatically detects when an object is no longer accessible to your program.

This process of automatically reclaiming the storage for unreachable objects is called *garbage collection* (GC).

No "dangling pointer" errors (*i.e.*, in C, inadvertently calling `free(p)` before last use of `p`).

No (actually, fewer) "space leak" problems (*i.e.*, in C, forgetting to `free` something that's no longer in use).

How does Garbage Collection Work?
Runtime system periodically:

1. Checks to see if memory is running low
2. If so, it
   (a) Traces all reachable data (objects)
   (b) Reclaims data (objects) not traced

GC adds some runtime overhead, but drastically reduces programmer time.

Java: Hello, World


```
/* HelloWorld.java */

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

/* end of HelloWorld.java */
```

```
$ ls
HelloWorld.java
$ javac HelloWorld.java
$
$ ls
HelloWorld.class  HelloWorld.java
$
$ java HelloWorld
Hello, world!
$
```


Java: Bytecodes for HelloWorld.java


```
$ javap -c HelloWorld
Compiled from HelloWorld.java
class HelloWorld extends java.lang.Object {
    public static void main(java.lang.String []);
    HelloWorld();

Method void main(java.lang.String [])
   0 getstatic #6 <Field java.lang.System.out Ljava/io/PrintStream;>
   3 ldc #1 <String "Hello, world!">
   5 invokevirtual #7
       <Method java.io.PrintStream.println(Ljava/lang/String;)V>
   8 return

Method HelloWorld()
   0 aload_0
   1 invokenonvirtual #8 <Method java.lang.Object.<init>()V>
   4 return

}
$
```


Java: Package & Class Example


```
/* StopObject.java */

package city.traffic.stops;

public class StopObject extends Object {
        /* class variables */
        private static int nextId = 0;

        /* instance variables */
        protected int Id;

        /* constructors */
        public StopObject() {
                Id = nextId++;
        }
}

/* end of StopObject.java */
```

```
/* StopSign.java */

package city.traffic.stops;

import city.traffic.stops.StopObject;

public class StopSign extends StopObject {
        public int nWay;

        public StopSign() {
                nWay = 2;
        }

        public StopSign(int n) {
                nWay = n;
        }
}

/* end of StopSign.java */
```



```
/* StopLight.java */

package city.traffic.stops;

import city.traffic.stops.StopObject;

public class StopLight extends StopObject {
        public StopLight() {
        }

        private int State = 0;  /* red=0, orange=1, green=2 */
        private final int nStates = 3;

        public int Change() {
                State = (State+1) % nStates;
                return State;
        }

        private boolean Blinking = false;

        public boolean toggleBlink() {
                Blinking = !Blinking;
                return Blinking;
        }
}

/* end of StopLight.java */
```



```
/* FourWayStopLightIntersection.java */

package city.traffic.intersections;

import city.traffic.stops.*;

public class FourWayStopLightIntersection extends Object {
        private boolean NS_MayGo = false;

        private StopLight N, S, E, W;

        public FourWayStopLightIntersection() {
                N = new StopLight();
                S = new StopLight();
                E = new StopLight();
                W = new StopLight();

                E.Change();
                W.Change();
        }

        public void ImpendingChange() {
                /* change lights to orange */
                if (NS_MayGo) {
                        N.Change();
                        S.Change();
                } else {
                        E.Change();
                        W.Change();
                }
        }
```



```
        public void Change() {
                /* assumes ImpendingChange has been invoked! */
                N.Change();
                S.Change();
                E.Change();
                W.Change();
                NS_MayGo = !NS_MayGo;
        }
}

/* end of FourWayStopLightIntersection.java */
```

Next Lecture

Java, from the bottom up...

Browse the Java pages on the course's web page