Java 2

Last time: basic class / object structure
Today: miscellaneous details needed to write a real program
Later: OOP design and inheritance

Static

Ivars or methods may be declared to be static
Static = exists once for the whole class
    Not associated with a particular instance. Instead, the static exists once for the whole class.

Static variable

A static variable is like a global variable for that class. The variable exists just once in the class, instead of once for each instance. The variable is in the namespace of the class, such as Student.someStaticVariable.
Static variables are somewhat rarely used.

Static method

A static method is a like a function that is defined inside the class.
A static method does not execute against a particular instance. There is no receiver.
A "static void hello()" method in the Student class is invoked as Student.hello();
In contrast, a regular method would be invoked with a message send like s.getStress(); where s points to a Student object.
The method "static void main(String[] args)" is special. To run a java program, you specify the name of a class. The system then starts the program by running the static main() function in that class, and the String[] array represents the command-line arguments.
Call a static method like this: Student.foo(), NOT s.foo(); where s points to a Student.
    s.foo() actually compiles, but it discards s as a receiver and translates to the same thing as Student.foo(). The s.foo() syntax is misleading, since it makes it look like a regular message send.

static method / var example

Suppose we modify the Student example in two ways...
-Add a static int "ctor_count" variable that counts the number of Student objects constructed -- increment it in the Student ctor
-Add a static method hello() that prints a greeting to standard output

```
public class Student {
   private int units;
```

```
   // Define a static int counter
   private static int ctor_count = 0;

   public Student(int init_units) {
      units = init_units;

      // Increment the counter
      ctor_count++;
      // ( "Student.ctor_count")
   }

   public static hello() {
      // Clients invoke this method as Student.hello();
      // Does not execute against a receiver, so
      // there is no "units" to refer to here

      System.println("Hello there");
      System.out.println("We've made " + ctor_count + " students");
   }

   // rest of the Student class
   ...
}
```

Typical static method error

Suppose in the static hello() method, we tried to refer to a "units" variable...

```
   public static void hello() {
     units = units + 1;    // error
   }
```

This gives an error message -- it cannot compile the "units" expression because there is no receiver.

Array

Arrays are built-in to Java

An array is declared according to the type of element

Arrays are always allocated in the heap and accessed through pointers

Array Declaration

int[] a; -- a can point to an array of ints (the array itself is not yet allocated)

int a[]; -- alternate syntax for C refugees -- do not use!

Student[] b; -- b can point to an array of Student objects

a = new int[100];

Allocate the array in the heap with the given size

Like allocating a new object

The array is zeroed out when allocated.

Array element access

Elements are accessed 0..len-1, just like C and C++

Java detects array-out--of-bounds access at runtime

a[0] = 1; -- first element

a[99] = 2; -- last element

a[-1] = 3; -- runtime array bounds exception

a.length -- returns the length of the array

Arrays know their length -- cool!
NOT a.length()

Arrays have compile-time types
a[0] = "a string"; / / NO -- int and String don't match
At CT, arrays know their type (int in this case) and trap errors such as above
The other Java collections WILL NOT have this compile time type system error catching (d'oh!), although it is rumored that compile time types are being added for Java 1.5

Student b[] = new Student[100];
Allocates an array of 100 Student pointers (initially all null)
Does not allocate any Student objects -- that's a separate pass

Int Array Code

Here is some typical looking int array code -- fill the array with squares: 1, 4, 9, ...

```
{
   int[] squares;
   squares = new int[100];    // allocate the array in the heap

   int i;
   for (i=0; i<squares.length; i++) {  // iterate over the array
      squares[i] = (i+1) * (i+1);
   }
}
```

Student Array Code

Here's some typical looking code that allocates 100 Student objects

```
{
   Student[] students;

   students = new Student[100];  // 1. allocate the array

   // 2. allocate 100 students, and store their pointers in the array
   int i;
   for (i=0; i<students.length; i++) {
      students[i] = new Student();
   }
}
```

String

Strings are built-in to the Java language
There is a built-in String class that implements many handy methods -- see the docs for the String class for a listing of its many methods
Strings (and char) use 2-byte unicode characters -- work with Kanji, Russian, etc.
String objects are "immutable"
Never change once created
i.e. there is no append() or reverse() method that changes the string state
To represent a different string state, create a new string with the different state
The immutable style is one way of building a class

The immutable style is one way to finesse memory sharing and multi-threading issues

String constants

Double quotes (") build String objects

"Hello World!\n" -- builds a String object with the given chars

System.out.print("print out a string"); / / or use println() to include the endline

String + String

+ concats strings together -- creates a new String based on the other two

String a = "foo";

String b = a + " bar"; / / b is now "foobar"

toString()

Many objects support a toString() method that creates some sort of String rep of the object -- handy for debugging. print(), printLn(), + will use the toString() of any object passed in.

See the docs

Look in the String class docs for the many messages it responds to

length() -- number of chars

String Methods

Here are some of the representative methods implemented in the String class (We'll do a demo of how to look up built-in methods in the library)

int length() -- number of chars

char charAt(int index) -- char at given index (0 based)

int indexOf(char c) -- first occurrence of char, or -1

int indexOf(String s)

boolean equals(Object) -- test if two strings are the same

boolean equalsIgnoreCase(Object) -- as above, but ignoring case

String toLowerCase() -- return a new String, lowercase

String substring(int begin, int end) -- return a new String made of the begin..end-1 substring from the original

Typical String Code

```
{
   String a = "hello";  // allocate 2 String objects
   String b = "there";
   String c = a;     // point to same String -- fine

   int len = a.length();   // 5
   String d = a + " " + b;    // "hello there"

   int find = d.indexOf("there");   // find: 6

   String sub = d.substring(6, 11); // extract: "there"

   d == b;     // false
   d.equals(b);   // true
}
```

StringBuffer

Similar to String, but can change the chars over time. More efficient to change one StringBuffer over time, than to create 20 slightly different String objects over time.

```
{
   StringBuffer buff = new StringBuffer();
   for (int i=0; i<100; i++) {
      buff.append(<some thing>);    // efficient append
   }
   String result = buff.toString();    // make a String once done with appending
}
```

System.out

System.out is a static object in the System class that represents standard output. It responds to the messages...

- println(String) -- print the given string on a line
- print(String) -- as above, but without and end-line

Example

```
System.out.println("hello");     -- prints to standard out
```

Java Primitives

Java has "primitive" types, much like C. Unlike C, the sizes of the primitives are fixed, and there are no unsigned variants.

- boolean -- true of false
- byte -- 1 byte
- char -- 2 bytes (unicode)
- int -- 4 bytes
- long -- 8 bytes
- float -- 4 bytes
- double - 8 bytes

Primitives can be used for local variables, parameters, and ivars.

However, it is not possible to get a pointer to a primitive. Pointers only work for objects and arrays.

There are "wrapper" classes Integer, Boolean, ... that can hold a single primitive value. These classes are "immutable", they cannot be changed once constructed. They can finesse, to some extent, the situation where you have a primitive value, but need a pointer to it.

The boundary between the primitive parts of Java and the true OOP parts are a little awkward.

== vs equals()

== -- compare pointers

equals()

- The default up in Object just does pointer compare
- Classes, such as String, can override equals() to provide deeper byte-by-byte semantics.

String Example

```
String a = new String("hello");
```

String a2 = new String("hello");
a == a2 / / false
a.equals(a2) / / true

Foo Example
Foo a = new Foo("a");
Foo a2 = new Foo("a");
a == a2 / / false
a.equals(a2) / / ??? -- depends on Foo overriding equals()

Garbage Collector GC

String a = new String("a");
String b = new String("b");
a = a + b; / / a now points to "ab"
Where did the original a go?
- It's still sitting in the heap, but it is "unreferenced" or "garbage" since there are no pointers to it. The GC thread comes through at some time and reclaims garbage memory.
- GC slows Java code down a little, but eliminates all those malloc()/free() bugs. The GC algorithm is very sophisticated.

Stack vs. Heap
- Remember, stack memory, is much, much faster than heap memory for allocation and deallocation.

Destructor
- In C++, the "destructor" is an explicit notification that the object is about to be destroyed. Java does not really have that feature -- the constructor marks creation, but destruction is indefinite. The "finalizer" feature is a lame attempt, but I do not recommend using it.

Declare Vars As You Go

In Java, it's possible to declare new local variables on any line.
This is a handy way to name and store values as you go through a computation...

```
public int method(Foo foo) {
   int a = foo.getA();
   int b = foo.getB();
   int sum = a + b;
   int diff = Math.abs(a - b);
   if (diff > sum) {
      int prod = a * b;
      ...
   }
}
```

Packages

Note: we're basically not using packages

```
package binky; // statement at start of file
```

The class defined in the file will be "in" the given package
Use reverse domain name to ensure package name uniqueness

```
   package edu.stanford.binky;
```

```
   package edu.stanford.binky.util;
```

Allow code from different sources to be combined without conflicts
Package hierarchy is represented in the compiled form (either in a file system or zip or jar file) -- the .class files for the above packages will be stored in a hierarchy: /edu/stanford/binky/[util/]
Clients use "import" statement to specify which packages they are using

Package Code

Declare that the Foo class is in the `edu.stanford.binky` package namespace.

```
package edu.stanford.binky;
public class Foo {
   public static void Bar() {}
}
```

Default Package

If there is no package declaration, the code all goes in the "default" package.
For assignments, we will use the default package.

Fully Qualified Name

Every Class has a fully qualified name, such as edu.stanford.binky.Foo. the compiler always works in terms of the fully qualified name. The short name, just "Foo", is only used in .java source files that have enough import statements that enable the short form.
Client code -- without an import

```
{
   edu.stanford.binky.Foo.bar(); // "fully qualified" Foo
}
```

Client code with import

```
import edu.stanford.binky.*;  // import all classes immediately in binky
import edu.stanford.binky.Foo;   // -or- import Foo only.

...

{
   Foo.bar();  // "short form" Foo
}
```

Compiling and Running

See the course page for info on how to compile and run on the various platforms.
The easiest way to compile on Unix is "javac *.java" in the directory that contains your .java files.
To run on Unix, use "java ClassName" to run the main() method of that class.

Compile Time Import

import adjusts the namespace at compile time -- when the compiler sees something like "Foo.bar();", knowing that the fully qualified name of the class you want to use is something like com.micorsoft.Foo.bar().

At compile time (CT), the compiler checks that the referenced classes and methods exist and match up logically, but it does not link in their code.

The ".*" version of import does not search sub-directories -- it only imports classes at that immediate level (D'oh!).

Having lots of import statements will not make your code any bigger or slower, but it may make your compiles a little slower.

Classpath

A set of locations (directories or jar or zip files) where the system searches for classes.

If the classpath is the empty string (you can often get away with that), then the classpath defaults to the current directory (".") + the built in system classes.

Compile Time (CT)

- Used at compile time to check that classes, methods, etc. exist -- e.g. for a reference to Foo
  1) Look for a definition of Foo in the default package in all the classpath locations
  2) Look for a definition of Foo in each of the imported packages in all the Classpath locations

Run Time (RT)

- To run a java program, the name of a class is specified -- java looks for a static main() function in that class. On Unix, run with the command "java RunMe" -- searches for a class named "RunMe" and searches for a static main() function in that class.
- Java loads classes at runtime as needed -- if not found get a java.lang.NoClassDefFoundError exception from the runtime system
- In Java 1.1, the classpath needed to include the "core" java classes as well, typically in a file called "classes.zip". In Java 1.2, the Java installation is supposed to automatically know where the core java classes are, so they should not be included in the classpath, so often the Classpath is just the empty string.

jar files

- A jar file is just a bunch of .class files gathered together in a single archive file (like a .zip).
- Jar files must be added explicitly to the classpath -- being in the right directory is not enough. So if the file "foo.jar" contains classes needed at runtime, they can be added with the "-classpath" option.."%java -classpath foo.jar ClassToRun".