Lecture 2: Java Review

CS 211 Spring 2006
Andrew Myers

Announcements

- Assignment 1 due next Thursday, 4PM
- Java Bootcamp tonight, 7:30PM, Upson B7
- Consulting has started (Upson 360, Sun-Wed)
- Watch the web page for ongoing announcements (www.cs.cornell.edu/Courses/cs211)

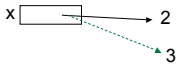
Plan

- Review some Java/OOP concepts
- Warn about pitfalls
- Discuss some Java 1.5 features

Values and variables

- A program uses different kinds of *values*
- *Variables* hold a *mutable* reference to a value

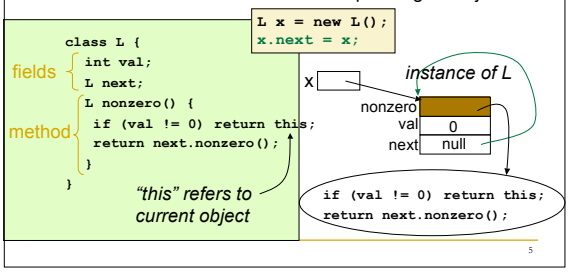
```
int x = 2;
x = 3;
```



- Values have different *types*
- *Primitive* values: 1, 2, 3, true, false, 3.0, null, …
- *Object* values:
  - Created by program, using a *class* defined by the program

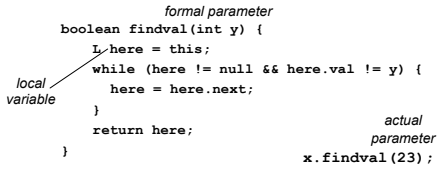
Classes

- A *class* defines how to make objects
  - Defines *fields*: variables that are part of object
  - Defines *methods*: named code operating on object



Parameters vs. locals

- Methods have 0 or more parameters/arguments: inputs to method code
- Can declare local variables too
- Both disappear when method returns



Constructors

- New instances of a class are created by calling a *constructor*
- Default constructor initializes all fields to default values (0 or null)
- Attached to class, not a method of the object

```
class L {
   int val;
   L next;
   L(int v) {
      val = v+1;
      next = null;
   }
}
```

```
new L(5);
```

val	6
next	null

Static (class) members

- Class can have fields and methods of its own
  - declare as "static"
  - do not need an instance of the class to use them
  - only one copy in entire program: access using class name

```
class L {
   int val;
   L next;
   L(int v) {
      num_created++;
   }
   static int num_created;
   static boolean any_exist() {
      return num_created != 0;
   }
}
```

can't use "this" here

```
if (L.any_exist()) {
  int n = L.num_created;
}
```


Programs

- A program is a collection of classes
  - Including built-in Java classes
- A running program does computation using instances of those classes.
- Program starts with a main method, declared as:

```
public static void main (String[] args) {
   ... body ...
}
```

Method must be named "main"

Parameters passed to program on command line

A class method; don't need an object to call it

Can be called from anywhere

Command Line Interface

- Command line arguments are contained in the String array parameter of the main method

```
class Foo {

  public static void main(String[] args) {
    System.out.println(args.length);
    for (int i = 0; i < args.length; i++) {
      System.out.println(args[i]);
    }
  }
}
```

- > java Foo
  0
  > java Foo asdf zxcv ss
  3
  asdf
  zxcv
  ss
  > java Foo hello world
  2
  hello
  world
- Try your programs in a command window, not just in Eclipse/DrJava
  - That's how we test them
  - Behavior may be a little different

Static vs. Instance Example

```
class Widget {
    static int nextSerialNumber = 10000;
    int serialNumber;

    Widget() {serialNumber = nextSerialNumber++;}

    Widget(int sn) {serialNumber = sn;}

       public static void main(String[] args) {
          Widget a = new Widget();
          Widget b = new Widget();
          Widget c = new Widget();
          Widget d = new Widget(42);
          System.out.println(a.serialNumber);
          System.out.println(b.serialNumber);
          System.out.println(c.serialNumber);
          System.out.println(d.serialNumber);
       }
    }
```


Parameters and Local Variables

- Parameters and local variables exist only while the method is running
  - Use parameters to pass input data to a method
  - Use local variables for temporary data used in a method
- Use fields for persistent data or data shared by several methods

Names

- Reference fields, methods in own class by unqualified name
  - serialNumber
  - nextSerialNumber
- Reference static fields in another class by qualified name
  - Widget.nextSerialNumber
- Reference instance fields with qualified name
  - a.serialNumber
- Example
  - System.out.println(a.serialNumber)
    - out is a static field in class System
    - The value of out is an instance of a class that has a method println(int)
- If an object has to refer to itself, use *this*

Overloading of Methods

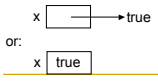
- A class can have several methods of the same name
  - But they must have different formal parameter types
  - *Signature* of a method is its name plus types of parameters, result type
- Example: `String.valueOf(...)` in Java API
  - There are 9 of them:

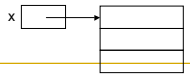
```
valueOf(boolean);
valueOf(int);
valueOf(long);
…
```

  - Parameter types are part of the name of the method

Primitive Types vs. Reference Types

- Primitive types
  - int, long, float, byte, char, boolean, ...
  - Efficiently implemented by storing directly into variable
  - Take a single word or 2 words of storage
  - Not considered objects by Java: "unboxed"
- Reference types
  - Objects defined by classes, or arrays
    - `String, int[ ], HashSet`
  - Take up more memory, have higher overhead
  - Can have special unboxed value `null`
    - Can only compare null with ==, !=
    - Other uses cause `NullPointerException`





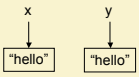
== vs. equals( )

- == tests whether variables refer to same things
- Works fine for primitive types
- For reference types (e.g., Strings), use == *only* if you mean actual identity of the two objects
  - == is "are they the same box"
  - Usually not what you want!
- To compare object contents, define an equals() method.

```
boolean equals(Object x);
```

Two different strings,] with value "hello"!

```
x = "hello";
y = "hello";
x == y?
```

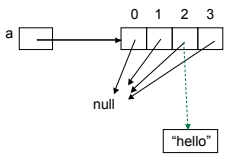


- Use x.equals("hello")
- Not x == "hello".

Arrays

- Arrays are reference types
- Array *elements* can be reference types or primitive types
  - E.g., int[] or String[]
- If a is an array, `a.length` is its length
- Its elements are `a[0], a[1], ..., a[a.length - 1]`
- Elements are mutable but length is fixed for any one array

```
String[] a = new String[4];
a[2] = "hello"
```

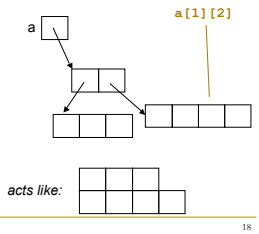


`a.length = 4`

Multidimensional arrays

- Multidimensional arrays are really arrays of arrays
  - E.g., `int[][]` is an array of integer arrays (`int[]`)
  - Multidimensional arrays can be *ragged* (i.e., all the arrays in the 2nd dimension need not be the same length)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];
```



acts like:

The Class Hierarchy

- Classes form a hierarchy
- Class hierarchy is a tree
  - Object is at the root (top)
  - E.g., String and StringBuilder are subclasses of Object
- The hierarchy is a tree because
  - Each class has at most one superclass
  - Each class can have zero or more subclasses
- Can use a class where superclass is expected
- Within a class, methods and fields of its superclass are available
  - But must use *super* for access to *overridden* methods

Array vs. ArrayList vs. HashMap

- Three extremely useful constructs (see Java API)
- Array
  - Storage is allocated when array created; cannot change
- ArrayList (in java.util)
  - An "extensible" array
  - Can append or insert elements, access i^{th} element, reset to 0 length
  - Can get an iteration of the elements
- HashMap (in java.util)
  - Save data indexed by keys
  - Can lookup data by its key
  - Can get an iteration of the keys or the values

HashMap Example

- Create a HashMap of numbers, using the names of the numbers as keys:

```
HashMap numbers = new HashMap();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

To retrieve a number:

```
Integer n = (Integer)numbers.get("two");
if (n != null) System.out.println("two = " + n);
```

- Caveat: returns null if does not contain the key
  - can use numbers.containsKey(key) to check this

New Feature: Generics

- Old

```
HashMap h = new HashMap();
h.put("one",new Integer(1));
Integer s = (Integer)h.get("one");
```

- New

```
HashMap<String, Integer> h =
    new HashMap<String,Integer>();
h.put("one", 1);
int s = h.get("one");
```

Another new feature: Automatic boxing/unboxing

- No longer necessary to do a class cast each time you take an element out of the HashMap

New Feature of J2SE 5.0: Enum

- Old

```
class Enum1 {
   public static final int WINTER = 0;
   public static final int SPRING = 1;
   public static final int SUMMER = 2;
   public static final int FALL   = 3;
   public static void main(String[ ] args) {
      System.out.println(WINTER);
   }
}
```

- New

```
enum Season { WINTER, SPRING, SUMMER, FALL }

class Enum2 {
   public static void main(String[ ] args) {
      System.out.println(Season.WINTER);
   }
}
```

- The first program prints 0; the second prints WINTER

Experimentation and Debugging

- Don't be afraid to experiment if you don't know how things work
  - An IDE (e.g., Eclipse) makes this easy
- Debugging
  - Think about what can cause the observed behavior
  - Isolate the bug using, for example, print statements combined with binary search
  - An IDE makes this easy by providing a *Debugging Mode*
    - Can step through the program while watching chosen variables