Java Review

Lecture 2
CS211 Spring 2007

Announcements

- Java Bootcamp
  - too late for the live sessions!
  - but tutorial & solutions still available online
- Assignment 1 has been posted and is due Wednesday, February 7, 11:59pm
- Check that you are in CMS
  - Report any CMS problems to your Section TA (email is fine)
- It's *really* a good idea to start on A1 and check CMS *this week* (well before the assignment is due)

More Announcements

- Available help
  - Consulting starts soon--watch web page for announcement
  - Instructor & TA office hours effective as soon as posted
- Check daily for announcements
  www.cs.cornell.edu/courses/cs211
- Register for ENGRD 211 or COM S 211
- Sections start next w*eek*
  - Section notes will be useful for A1

Today

- A short, biased history of programming languages
- Review of some Java/OOP concepts
- Common Java pitfalls
- Debugging and experimentation

Machine Language

- Used with the earliest electronic computers (1940s)
  - Machines use vacuum tubes instead of transistors
- Programs are entered by setting switches or reading punch cards
- All instructions are numbers

- Example code

```
0110 0001 0000 0110
Add  Reg1         6
```

- An idea for improvement
  - Use "words" instead of numbers
  - Result: Assembly Language

Assembly Language

- Idea: Use a program (an *assembler*) to convert assembly language into machine code
- Early assemblers were some of the most complicated code of the time (1950s)

- Example code

```
ADD R1 6
MOV R1 COST
SET R1 0
JMP TOP
```

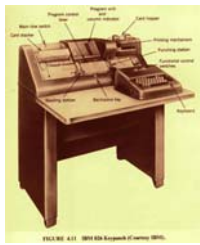
  - Typically, an assembler used *2 passes*
- Idea for improvement
  - Let's make it easier for humans by designing a high-level computer language
  - Result: high-level languages

High-Level Language

- Idea: Use a program (a *compiler* or an *interpreter*) to convert high-level code into machine code
- Pro
  - Easier for humans to write, read, and maintain code
- Con
  - The resulting program will never be as efficient as good assembly-code
    - Waste of memory
    - Waste of time
- The whole concept was initially controversial
  - FORTRAN (mathematical FORmula TRANslating system) was designed with efficiency very much in mind

FORTRAN

- Initial version developed in 1957 by IBM



- Example code

```
C      SUM OF SQUARES
       ISUM = 0
       DO 100 I=1,10
       ISUM = ISUM + I*I
  100 CONTINUE
```

- FORTRAN introduced many high-level language constructs still in use today
  - Variables & assignment
  - Loops
  - Conditionals
  - Subroutines

ALGOL

- ALGOL = ALGOrithmic Language
- Developed by an international committee
- First version in 1958 (not widely used)
- Second version in 1960 (widely used)
- Sample code

```
comment Sum of squares
begin
   integer i, sum;
   for i:=1 until 10 do
      sum := sum + i*i;
end
```

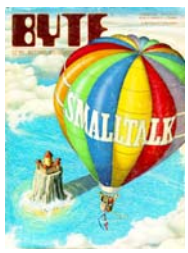
- ALGOL 60 included *recursion*
  - Pro: easier to design clear, succinct algorithms
  - Con: too hard to implement; too inefficient

COBOL

- COBOL = COmmon Business Oriented Language
- Developed by the US government (about 1960)
  - Design was greatly influenced by Grace Hopper
- Goal: Programs should look like English
  - Idea was that *anyone* should be able to read and understand a COBOL program
- COBOL included the idea of *records* (a single data structure with multiple *fields*, each field holding a value)

Simula & Smalltalk

- These languages introduced and popularized *Object Oriented Programming* (OOP)
  - Simula was developed in Norway as a language for simulation in the 60s
  - Smalltalk was developed at Xerox PARC in the 70s
- These languages included
  - Classes
  - Objects
  - Subclasses & Inheritance



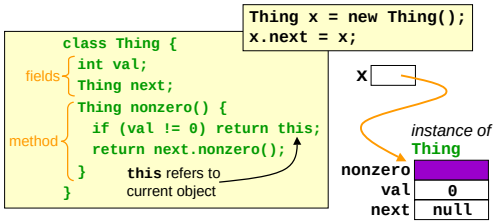
Java – 1995

- Java includes
  - Assignment statements, loops, conditionals from FORTRAN (but syntax from C)
  - Recursion from ALGOL
  - Fields from COBOL
  - OOP from Simula & Smalltalk

Java

Classes

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



Constructors

- New instances of a class are created by calling a *constructor*
- Default constructor initializes all fields to default values (`0` or `null`)

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

```
new Thing(5);
```

val	6
next	null

Static Fields and Methods

- A class can have fields and methods of its own
  - Declare with keyword `static`
  - Do not need an instance of the class to use them
  - Only one copy – access using class name

```
class Thing {
   int val;
   static int numCreated = 0;
   Thing(int v) {
      val = v;
      numCreated++;
   }
   static boolean anyExist() {
      return numCreated != 0;
   }
}
```
can't use `this` in a static method

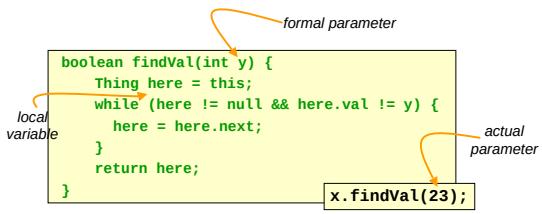
```
if (Thing.anyExist()) {
  int n = Thing.numCreated;
}
```

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

- Methods have 0 or more *parameters/arguments* (i.e., inputs to the method code)
- Can declare *local variables* too
- Both disappear when method returns



A Common Pitfall

local variable shadows field

```
class Thing {
   int val;

   boolean setVal(int v) {
      int val = v;
   }
}
```

- you would like to set the instance field `val = v`
- but you have declared a new local variable `val`
- assignment has no effect on the field `val`

A Common Pitfall

local variable shadows field

```
class Thing {
   int val;

   boolean setVal(int v) {
      int val = v;
   }
}
```

- you would like to set the instance field **`val = v`**
- but you have declared a new local variable **`val`**
- assignment has no effect on the field **`val`**

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...
}
```

No return value

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

Names

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

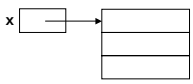
Overloading of Methods

- A class can have several methods of the same name
  - But all methods must have different *signatures*
  - The *signature* of a method is its name plus types of its parameters
- Example: **`String.valueOf(...)`** in Java API
  - There are 9 of them:
    - **`valueOf(boolean);`**
    - **`valueOf(int);`**
    - **`valueOf(long);`**
    - **`...`**
  - Parameter types are part of the method's signature

Primitive Types vs Reference Types

- Primitive types
  - **`int`**, **`short`**, **`long`**, **`float`**, **`byte`**, **`char`**, **`boolean`**, **`double`**
  - Efficiently implemented by storing directly into variable
  - Take a single word or 2 words of storage
  - Not considered an **`Object`** by Java: "unboxed"

- Reference types
  - Objects and arrays
    - **`String`**, **`int[]`**, **`HashSet`**
  - Usually require more memory
  - Can have special value **`null`**
    - Can compare **`null`** with **`==`**, **`!=`**
    - Generates **`NullPointerException`** if you try to dereference it



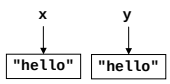
`==` vs `equals( )`

- **`==`** tests whether variables hold identical values
- Works fine for primitive types
- For reference types (e.g., **`String`**), you usually want to use **`equals()`**
  - == means "they are the same object"
  - 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?
```



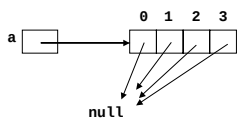
== vs equals( )

```
"xy" == "xy"                "xy".equals("xy")

"xy" == "x" + "y"           "xy".equals("x" + "y")

"xy" == new String("xy")    "xy".equals(new String("xy"))
```

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]`
- The length is fixed for any one array

```
String[] a = new String[4];
```

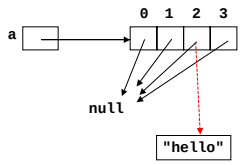


`a.length = 4`

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]`
- The length is fixed for any one array

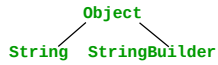
```
String[] a = new String[4];
a[2] = "hello"
```



`a.length = 4`

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
  - Each class has exactly one superclass (except `Object`, which has no superclass)
  - Each class can have zero or more subclasses
- Can use a class anywhere superclass is expected



- Within a class, methods and fields of its superclass are available
  - 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
- `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);
```

- returns `null` if the `Hashmap` does not contain the key
  - Can use `numbers.containsKey(key)` to check this

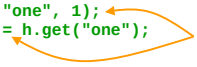
Generics (Java 1.5)

- 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 "box/unbox" an **`int`**

Experimentation and Debugging

- Don't be afraid to experiment if you don't know how things work
  - An *IDE* (*Interactive Development Environment*; e.g., DrJava or Eclipse) makes this easy
- Debugging
  - Do not just make random changes, hoping something will work
  - Think about what could cause the observed behavior
  - Isolate the bug using 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