Java Review

Lecture 2
CS2110 Summer 2008

Announcements

- Java Bootcamp
  - Another session tonight 4-6 in B7 Upson
  - tutorial & solutions also available online
- Assignment 1 has been posted and is due Friday, June 27, 10pm
- Check that you are in CMS after 1pm
  - Report any CMS problems to me ASAP
- It's *really* a good idea to start on A1 and check CMS very soon (like today)

More Announcements

- Available help
  - Office hours on course web site
- Check web page daily for announcements
  - Also monitor newsgroup (working??)
- Last day to register with Summer Sessions office is tomorrow

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 uses *2 passes*

- Idea for improvement
  - Let's make it easier for humans
  - 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)
- ALGOL 60 included *recursion*
  - Pro: easier to design clear, succinct algorithms
  - Con: too hard to implement; too inefficient
- Sample code

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

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

- Sample code

```
MULTIPLY B BY B GIVING B-SQUARED.
MULTIPLY 4 BY A GIVING FOUR-A.
MULTIPLY FOUR-A BY C GIVING FOUR-A-C.
SUBTRACT FOUR-A-C FROM B-SQUARED GIVING RESULT-1.
COMPUTE RESULT-2 = RESULT-1 ** .5.
```

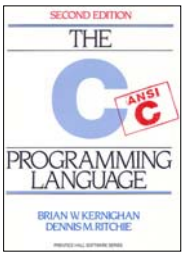
- COBOL included the idea of *records* (a single data structure with multiple *fields*, each field holding a value)
- *Immensely* popular language

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

C

- Developed in 1972 by Dennis Ritchie at Bell Labs
- Idea was a high-level language with nearly the power of assembly language
  - Originally used to write UNIX
- Pro: fast and powerful, simple syntax
- Con: often *too* powerful
  - C trusts the programmer
  - Very difficult to write secure, stable code
- Very widely used
  - Many descendents: C++, C#, Objective C, etc.



Java – 1995

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

Classes

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

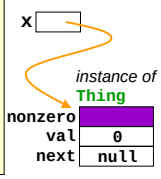
```
Thing x = new Thing();
x.next = x;
```

```
class Thing {
  int val;
  Thing next;
  Thing nonzero() {
    if (val != 0) return this;
    return next.nonzero();
  }
}
```

fields: `int val;` `Thing next;`

method: `Thing nonzero() { ... }`

`this` refers to current 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

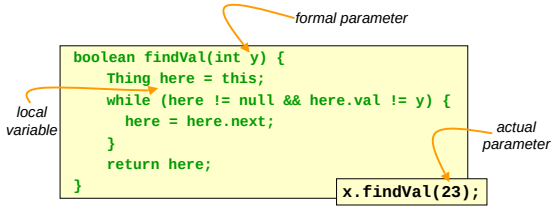
```
if (Thing.anyExist()) {
  int n = Thing.numCreated;
}
```

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

Formal parameter shadows field

```
class Thing {
   int val;

   boolean setVal(int val) {
      val = 10 * val;
   }
}
```

- assignment has no effect on the field **`val`**

A Common Pitfall

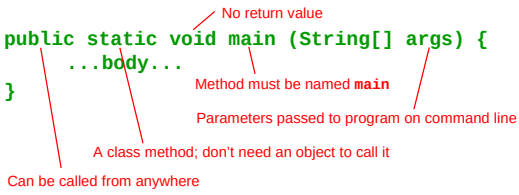
Formal parameter shadows field

```
class Thing {
   int val;

   boolean setVal(int _val) {
      val = 10 * _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:



Names

- Refer to fields & methods in own class by (unqualified) name
  - **`serialNumber, nextSerialNumber`**
- Could also use **`this`** reference (but rarely needed)
  - **`this.serialNumber, this.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)`**

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
  - Return type is *not* part of the signature

Types

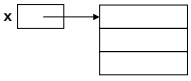
- Primitive types
  - `int`, `short`, `long`, `float`, `byte`, `char`, `boolean`, `double`
  - Efficiently implemented by storing directly into variable
  - Not considered an `Object` by Java: "unboxed"

x `true`

Type	Size	Range
byte ☐	1 byte	[-128, 127]
short	2 bytes	[-32768, 32767]
int	4 bytes	[-2,147,483,648, 2,147,483,647]
long	8 bytes	[-9,223,372,036,854,775,808, 9,223,372,036,854,775,807]
float	4 bytes	Approx. ±[1.40e-45, 3.40e+38]
double	8 bytes	Approx. ±[4.94e-324 to 1.80e+308]
boolean		{true, false}
char	2 bytes	[0, 65535]

Reference Types

- 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



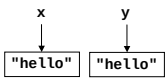
`==` vs `equals( )`

- `==` 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?
```



Fine print on Strings…

- The `String` class is special
  - It's the only class that is allowed to overload operators
    - E.g. `String s = "x" + "y"`
    - No other class is allowed to do this

```
Widget w = w1 + w2
```

  - String objects are *immutable*: it is not possible to change the contents of a `String` object after it has been constructed
  - If the same string literal appears multiple times in a program, the compiler *might* create only one object as an *optimization*

```
"xy" == "xy"
```
true/false?
```
"xy" == "x" + "y"
```
true/false?
```
"xy" == new String("xy")
```
false

```
"xy".equals("xy")
```
true
```
"xy".equals("x" + "y")
```
true
```
"xy".equals(new String("xy"))
```
true

Another common pitfall

```
class Widget {
   public static void main(String[] args) {
      Widget a;

      System.out.println(a.serialNumber);
   }
}
```

- `Widget a;` just creates a reference to a `Widget` object
  - But not a `Widget` object!
  - `a` is automatically initialized to null
  - So `a.serialNumber` throws a `NullReferenceException`
- Fix it by creating a new object: `Widget a = new Widget();`

Yet another pitfall

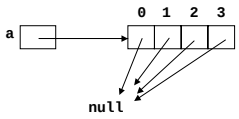
- The `=` operator copies references, not objects

```
class Widget {
   public int modelNumber;

   public static void main(String[] args) {
      Widget widget1 = new Widget();
      Widget widget2 = widget1;

      widget1.modelNumber = 13;
      widget2.modelNumber = 14;
      System.out.println(widget1.modelNumber);
      System.out.println(widget2.modelNumber);
   }
}
```

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];
```

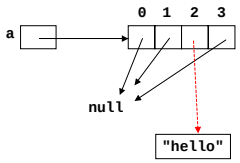


`a.length = 4`

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

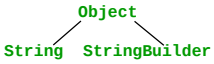
```
String[] a = new String[4];
a[2] = "hello"
```



`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`
  - 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
  - Try to isolate the bug
    - Use print statements combined with binary search
    - Comment out unrelated parts of the program
    - But make sure to keep backup copies of your code!!
  - An IDE makes this easy by providing a *Debugging Mode*
    - Can step through the program while watching chosen variables