Java Review

Lecture 2
CS211 - Fall 2006

Announcements

- Java Bootcamp (two identical sessions)
  - Tuesday, 8/29, 7:30-10:30pm, Upson B7
  - Wednesday, 8/30, 7:30-10:30pm, Upson B7
  - Tutorial & solutions are available online
- Assignment 1 has been posted and is due Thursday, Sept 7 at 4:30 PM
  - A1 companion files are online since last night
- Check that you appear correctly within 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 tonight
  - TA office hours start next week
  - Instructor office hours have already started
- Watch the course web page for ongoing announcements
  - www.cs.cornell.edu/Courses/cs211
- Registering for ENGRD211 or COM S211?
  - Engineering now says they don’t care what Engineers sign up for
- Sections start this week
  - Section Notes (available online by Thurs) may be useful for A1

Today's Plan

- A short, biased history of programming languages
- Review some Java/OOP concepts
- Warn about some Java pitfalls

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
- Idea for improvement
  - Let's 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 passers
- 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.
  - Thus, FORTRAN (mathematical FORmula TRANslating system) was designed with efficiency very-much in mind.
- FORTRAN introduced many of the ideas typical of programming languages.
  - Assignment
  - Loops
  - Conditionals
  - Subroutines.

**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
  ```

**ALGOL**

- Sample code
  ```
  comment Sum of squares
  begin
  integer i, sum;
  for i:=1 until 10 do
    sum := sum + i*i;
  end
  ```
- 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: Makes it easy 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 (late 60s).
  - Smalltalk was developed at Xerox PARC in the 70s.

**Java**

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

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

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

- Parameters vs. 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

```java
boolean findval(int y) {
    L here = this;
    while (here != null && here.val != y) {
        here = here.next;
    }
    return here;
}
```

- Parameters passed to program on command line
  - A class method; don't need an object to call it
  - Can be called from anywhere

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

Static (Class) Members

- A class can have fields and methods of its own
  - Declared as "static"
  - Do not need an instance of the class to use them
  - Only one copy in entire program; access by using class name

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

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 an instance method

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

Static vs. Instance Example

```java
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);
    }
}
```
Names

- Refer to fields, methods in own class by unqualified name  
  `serialNumber`
- Refer to static fields in another class by qualified name  
  `Widget.nextSerialNumber`
- Refer to instance fields with qualified name  
  `a.serialNumber`

Example

```java
System.out.println(a.serialNumber)
```

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, 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 value null
  - Can only compare null with `==`
  - Other uses cause `NullPointerException`

== vs. equals()

- `==` tests whether variables hold identical values
- Works fine for primitive types

For reference types (e.g., `Strings`), you usually want to use `equals()`:

- `==` means “are they the same box”
- Usually not what you want!

- To compare object contents, define an `equals()` method

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

Two different strings, `x = "hello"`; `y = "hello"`; `x == y`?

```
x
hello
```

```
y
hello
```

Use `x.equals("hello")`

Mutually equals `"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]`
- The length is fixed for any one array

```java
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)

```java
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

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 ith 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:

  ```java
  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:

  ```java
  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

Generics (New Feature of Java 1.5)

- Old

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

- New

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

  * 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
  - 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