CS/ENGRD 2110
Object-Oriented Programming and Data Structures
Fall 2012
Doug James

Lecture 2: Java Review

Academic Excellence Workshop
CS 2110 AEW

SIGN UP FOR 1 CREDIT S/U COURSE!!!
- Students enrolled in AEWs, on average, earn a HIGHER GRADE in the course than those not enrolled in the AEW
- Get EXTRA PRACTICE solving problems so that course assignments are easier to complete
- FREE FOOD every class!!!

CLASS TIME: Fridays 2:30PM-4:25PM HLS 401

To add the course: just add ENGRG 1011 on Student Center
Email Jennifer Doughty (jad359) for more details or if you have questions

Outline

- A brief (biased) history of programming languages
- Review of some Java/OOP concepts
- Java tips, trick, and pitfalls

interested in Information Science?
IS/ISST major? Minor? Not sure?

Join ISSA, the Information Science Student Association!
Come to our first meeting of 2012-13
THURSDAY (9/6) at 4:30pm in Upson Lounge
There will be pizza and soda!

Questions? Want to join our listserv?
Contact zip2@cornell.edu

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

- 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

- 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

FORTRAN

• Initial version developed in 1957 by IBM
• Example code
  ```fortran
  C SUM OF SQUARES
  ISUM = 0
  DO 100 I=1,10
  ISUM = ISUM + I*I
  100 CONTINUE
  ```

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)

ALGOL

• ALGOL = ALGOrithmic Language
• Developed by an international committee
• First version in 1958 (not widely used)
• Second version in 1960 (widely used)
• Sample code
  ```java
  C SUM OF SQUARES
  ISUM = 0
  FOR I = 1 TO 10
    ISUM = ISUM + I*I
  END
  ```

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

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

Java & 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
We assume you already know Java...

- Classes and objects
- Static vs instance fields and methods
- Local variables
- Primitive vs reference types
- Private vs public vs package
- Constructors
- Method signatures
- Arrays
- Subtypes and Inheritance, Shadowing

Java is object oriented

- In most prior languages, code was executed line by line and accessed variables or record
- In Java, we think of the data as being organized into objects that come with their own methods, which are used to access them
  - This shift in perspective is critical
  - When coding in Java one is always thinking about “which object is running this code?”

Dynamic vs. Static

- Some kinds of information is “static”
  - There can only be one instance
  - Like a “global variable” in C or C++ (or assembler)
  - In languages like FORTRAN, COBOL most data is static.
- Object-oriented information is “dynamic”
  - Each object has its own private copy
  - When we create a new object, we make new copies of the variables it uses to keep its state
  - Languages like C and C++ allow us to allocate memory at runtime, but don’t offer a lot of help for managing it
- In Java this distinction becomes very important

Constructors

- Called to create new instances of a class
- Default constructor initializes all fields to default values (0 or null)

```java
class Thing {
    int val;
    Thing(int val) {
        this.val = val;
    }
    Thing() {
        this(3);
    }
}
```

Static Initializers

- Run once when class is loaded
- Used to initialize static objects

```java
class StaticInit {
    static String[] courses = new String[2];
    static {
        courses[0] = "CS 2110";
        courses[1] = "CS 2112";
    }
    public static void main(String[] args) {
        ...
    }
}
```

Static methods and variables

- If a method or a variable is declared “static” there will be just one instance for the class
  - Otherwise, we think of each object as having its own “version” of the method or variable
- Anyone can call a static method or access a static variable
- But to access a dynamic method or variable Java needs to know which object you mean
### Static vs Instance Example

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

    Widget() {
        serialNumber = nextSerialNumber;
        nextSerialNumber++;
    }

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

### Names

- Refer to my static and instance fields & methods of same class/object by (unqualified) name:
  - `nextSerialNumber`
- Refer to static fields & methods in another class using name of the class
  - `Widget.nextSerialNumber`
- Refer to instance fields & methods of another object using name of the object
  - `a.serialNumber`
- Example
  - `System.out.println(a.serialNumber)`
- If an object must refer to itself, use `this`

### 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`

### Avoiding trouble

- Keep in mind that "main" is a static method
  - hence anything main calls needs to have an associated object instance or itself be static
- Use of static methods is discouraged

```java
class Thing {
    int counter;
    static int sequence;

    public static void main(String[] args) {
        int s = ++sequence; // Illegal: counter is associated with an object of type Thing. But which object?
        int s = ++sequence; // Legal: sequence is static too
    }
}
```

### The main Method

- A class method does not need an object to call it
- No return value
- Method must be named `main`
- Parameters passed to program, either from command line or from "Run"/"Debug" dialog box in Eclipse

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

### 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 vs Reference Types**

- **Primitive types**
  - `int, short, long, float, byte, char, boolean, double`
  - Efficient
  - 1 or 2 words
  - Not an Object — unboxed

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

- Efficient
  - 1 or 2 words
  - Not an Object — unboxed

**"==" vs "equals()" for String**

<table>
<thead>
<tr>
<th>What you wrote</th>
<th>Value</th>
<th>What you should write</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;xy&quot; == new String(&quot;xy&quot;)</td>
<td>False</td>
<td>&quot;xy&quot;.equals(new String(&quot;xy&quot;))</td>
</tr>
<tr>
<td>&quot;xy&quot; == &quot;xy&quot;</td>
<td>True</td>
<td>&quot;xy&quot;.equals(&quot;xy&quot;)</td>
</tr>
<tr>
<td>&quot;xy&quot; == &quot;x&quot;+&quot;y&quot;</td>
<td>True</td>
<td>&quot;xy&quot;.equals(&quot;x&quot;+&quot;y&quot;)</td>
</tr>
</tbody>
</table>

Use of "==" quite tricky for Strings—see Equals.java

**Arrays**

- Arrays are reference types
- Array elements can be reference types or primitive types
  - E.g., `int[] a[]`, `Object[] a[]`
- 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!

**Accessing Array Elements Sequentially**

```java
public class CommandLineArgs {
    public static void main(String[] args) {
        System.out.println(args.length);
        // old-style
        for (int i = 0; i < args.length; i++) {
             System.out.println(args[i]);
        }
        // new style
        for (String s : args) {
             System.out.println(s);
        }
    }
}
```

**Class Hierarchy**

Every class (except `Object`) has a unique immediate superclass, called its parent.
Overriding

• A method in a subclass **overrides** a method in superclass if:
  * both methods have the same name,
  * both methods have the same signature (number and type of parameters and return type), and
  * both are static methods or both are instance methods

• Methods are dispatched according to the runtime type of the object (dynamic binding / late binding)

Overriding (cont’d)

Casting and Method Dispatch

```java
class A {
    public int m() {...}
}

class B extends A {
    public int m() {...}
}

B b = new B();
b.m();
A a = new B(); //upcasting
a.m();
```

Always calls methods of the class that was use for creation with "new".

Overriding (cont’d)

Unexpected Consequence

```java
class A {
    public int m() {...}
}

class B extends A {
    private int m() {...} //illegal!
}

A a = new B(); //upcasting
a.m();
//would invoke private method in //class B at runtime!
```

An overriding method cannot have more restricted access than the method it overrides

Overloading Revisited

```java
class Base {
    ...
}
class Derived extends Base {
    ...
}
class Overload{
    public void m (Derived b){
        System.out.println("Overload.m(Derived)");
    }
    public void m (Base a){
        System.out.println("Overload.m(Base)");
    }
    public static void main(String []args){
        Overload t = new Overload();
        Base b = new Base();
        Base d = new Derived();
        t.m(b);
        t.m(d);
    }
}
```

Output:

```
Overload.m(Base)
Overload.m(Base)
```

Overloading Revisited

Remember: overloading resolved at compile time

Accessing Overridden Methods

• Suppose a class S overrides a method m in its parent
• Methods in S can invoke the overridden method in the parent as **super.m()**
• In particular, can invoke the overridden method in the overriding method!
• Caveat: cannot compose super more than once as in **super.super.m()**

Accessing Overridden Methods

```java
class A {
    public m() {...}
}

class B extends A {
    public m() {...}
}

B b = new B();
b.m();
A a = new B(); //upcasting
a.m();
```

• Overriding requires more runtime type checking than overloading
• Overloading can be resolved at compile time

Shadowing

• Like overriding, but for fields instead of methods
  – Superclass: variable x of some type
  – Subclass: variable x perhaps of some other type
  – Method in subclass can access shadowed variable using **super**
• Variable references are resolved using static binding (i.e., at compile-time), not dynamic binding (i.e., not at runtime)
  – Variable reference x uses the static type (declared type) of the variable x, not the runtime type of the object referred to by x
• Shadowing variables is bad medicine and should be avoided

Shadowing

```java
class A {
    public int x = 1; //static binding
}

class Derived extends A {
    public int x = 2; //static binding
}
```

```java
A a = new A();
A b = new Derived();
    //error: incompatible types
    b.x = 10;
```

```java
Derived d = new Derived();
    // ok
    d.x = 10;
    // error: cannot access field 'x' in superclass 'A'
    a.x = 10;
```
**Experimentation and Debugging**

- Don’t be afraid to experiment if you are not sure how things work
  - Documentation isn’t always clear
  - Interactive Development Environments (IDEs), e.g. Eclipse, make this easier
- Debugging
  - Do not just make random changes, hoping something will work
  - Think about what could cause the observed behavior
  - Isolate the bug
- An IDE makes this easier by providing a Debugging Mode
- Can set breakpoints, step through the program while watching chosen variables