CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Spring 2011
Thorsten Joachims

Lecture 2: Java Review

Outline

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

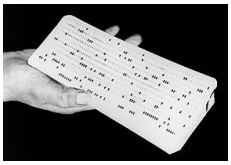
- An idea for improvement
  - Use words instead of numbers
- Result: Assembly Language

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
```



- Idea for improvement
  - Let's make it easier for humans by designing a high-level computer language
- Result: high-level languages

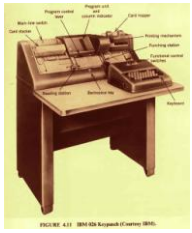
High-Level Language

- **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
  - Comments

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

7

COBOL

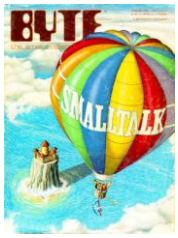
- COBOL = COmmon Business Oriented Language
- Developed by the US government (about 1960)
  - Design was greatly influenced by Grace Hopper
- Goal: Programs should look like English
  - Idea was that *anyone* should be able to read and understand a COBOL program

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

8

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



9

Java – 1995

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

10

We assume you already know Java…

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

11

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`)

```
class Thing {
   int val;

   Thing(int val) {
      this.val = val;
   }

   Thing() {
      this(3);
   }
}
```

```
Thing one = new Thing(1);
Thing two = new Thing(2);
Thing three = new Thing();
```

14

Static Initializers

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

```
class StaticInit {

   static Set<String> courses = new HashSet<String>();

   static {
      courses.add("CS 2110");
      courses.add("CS 2111");
   }

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

15

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

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

   Widget() {
      serialNumber = nextSerialNumber++;
   }

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

17

Names

- Refer to my static and instance fields & methods 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 of another object 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 must refer to itself, use `this`

18

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`

19

The `main` Method

Can be called from anywhere

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

No return value

Method must be named `main`

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

Parameters passed to program, either from command line or from "Run"/"Debug" dialog box in Eclipse

20

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

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

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

Overloading of Methods

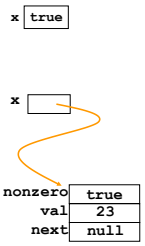
- A class can have several methods of the same name
  - But all methods must have different *signatures*
  - The *signature* of a method is its name plus types of its parameters
- Example: `String.valueOf(...)` in Java API
  - There are 9 of them:
    - `valueOf(boolean);`
    - `valueOf(int);`
    -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

22

22

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`



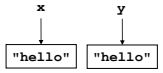
23

"==" is not "`equals()`"

- `==` tests whether variables hold identical values
  - shallow equality
  - works fine for primitive types
- `equals()` test whether two objects (e.g., `String`) contain equivalent data
  - deep equality
  - need to use for reference types

Two different strings with value `"hello"`

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



- To compare object *contents*, override `Object.equals()`
- But if you do this, must also override `Object.hashCode()` (more on this later)

24

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

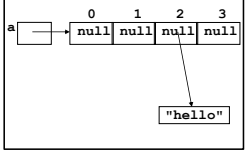
What you wrote.	Value?	What you should write.
"xy" == new String("xy")	False	
"xy" == "xy"	True	"xy".equals("xy")
"xy" == "x" + "y"	True	"xy".equals("x" + "y")

25

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!

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



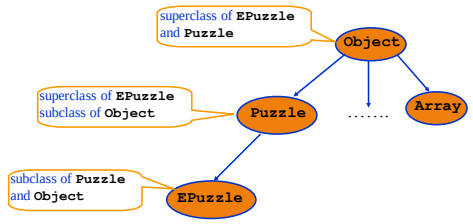
26

Accessing Array Elements Sequentially

```
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);
      }
   }
}
```

27

Class Hierarchy



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

28

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)

29

Unexpected Consequence

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

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

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

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

30

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()**

31

Shadowing

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

32

Overloading Revisited

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

Output:

```
Test.m(Base)
Test.m(Base)
```

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 iterator of the keys or the values

34

HashMap Example

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

```
Map<String, Integer> num =
      new HashMap<String, Integer>();
num.put("one", new Integer(1));
num.put("two", new Integer(2));
num.put("three", new Integer(3));
```

- To retrieve a number:

```
Integer n = num.get("two");
```

- returns **null** if the **HashMap** does not contain the key
  - Can use **num.containsKey(key)** to check this

35

Generics and Autoboxing

- Old (pre-Java 5)

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

- New (generics)

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

- New (generics + autoboxing)

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

36

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

37