



Java Review

Lecture 2
CS2110 Fall 2008

Announcements

- Assignment 1 has been posted
 - Due Wednesday, September 10, 11:59pm
 - Materials available in CMS
- Check that you are in CMS
 - Report any 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)
- Available help
 - Consulting will start very soon—watch for announcements
 - Instructor & TA office hours are in effect
- Check daily for announcements
<http://courses.cs.cornell.edu/cs2110>

2

More Announcements

- Sections start this week
 - Section material will be useful for A1
- Dexter out of the country 9/3 - 9/7 (Nottingham, England)
 - 9/4 guest lecturer: Juan Altmeyer Pizzorno



3

Today — A Smorgasbord

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

4

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



5

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`

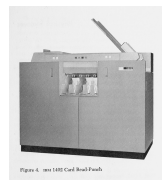
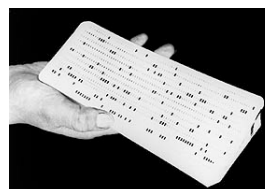


Figure 6.100 UNIVAC Card-Read Punch



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

6

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



7

FORTRAN

- Initial version developed in 1957 by IBM

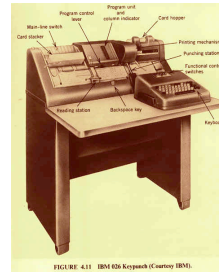


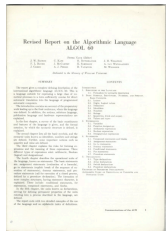
FIGURE 4.11 IBM 605 Keypunch (Courtesy IBM)

- 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

8

ALGOL



- Sample code


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

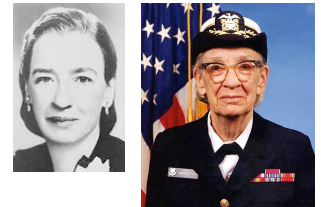
- ALGOL = ALGORithmic Language
- Developed by an international committee
- First version in 1958 (not widely used)
- Second version in 1960 (widely used)

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

9

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)

10

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



11

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™ and logo © Sun Microsystems, Inc.

12

We will assume you already know something about ...

- 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

13

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 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);
    }
}
```

A Common Pitfall

local variable shadows field

```
class Thing {
    int val;

    boolean setVal(int v) {
        val 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`

17

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 on command line

18

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**

19

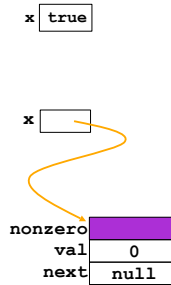
Overloading of Methods

- 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

20

Primitive vs Reference Types

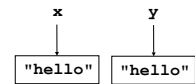
- Primitive types
 - `int, short, long, float, byte, char, boolean, double`
 - Efficient
 - 1 or 2 words
 - Not an **Object**—*unboxed*
- Reference types
 - Objects and arrays
 - `String, int[], HashSet`
 - Usually require more memory
 - Can have special value `null`
 - Can compare `null` with `==`, `!=`
 - Generates `NullPointerException` if you try to dereference `null`



21

== VS equals ()

- `==` tests whether variables hold identical values (*shallow equality*)
- Works fine for primitive types
- For reference types (e.g., `String`), you usually want to use `equals()` (*deep equality*)
- Two different strings with value "hello"
 - `x = "hello";`
 - `y = "hello";`
 - `x == y?`
- To compare object *contents*, override `Object.equals()`
 - `boolean equals(Object x);`
- But if you do this, must also override `Object.hashCode()` (more on this later)



22

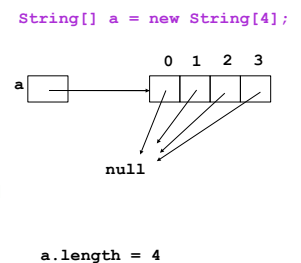
== VS equals ()

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

23

Arrays

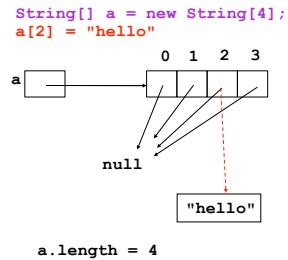
- Arrays are reference types
- Array *elements* can be reference types or primitive types
 - E.g., `int[]` or `String[]`
- If `a` is an array, `a.length` is its length
- Its elements are `a[0], a[1], ..., a[a.length-1]`
- The length is fixed



24

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



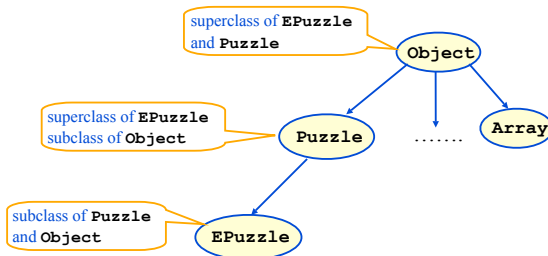
25

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);  
        }  
    }  
}
```

26

Class Hierarchy



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

27

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

28

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()`

29

Unexpected Consequence

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

```
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!
```

30

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 `x.v` 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

31

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

32

HashMap Example

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

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

To retrieve a number:

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

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

33

Generics and Autoboxing

- Old (pre-Java 5)

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

- New (generics)

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

- New (generics + autoboxing)

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

34

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

35