# JAVA REVIEW

Lecture 2
CS2110 Fall 2008

## Announcements

- Assignment 1 has been posted
  - Due Wednesday, September 9, 11:59pm.
    - *No partners on A1. (Groups of 2 allowed on A2-A5).*
- Check that you are in CMS
  - Materials available in CMS
  - Report any problems to Bill Hogan, cs22110 administrative assistant (whh@cs.cornell.edu)

## Announcements

- It's *really* a good idea to start on A1 and check CMS *this week* (well before the assignment is due)
- Sections start this week
  - Section material will be useful for A1
- 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**

  Newsgroup also worth watching

## Today — A Smorgasbord

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

## Machine Language

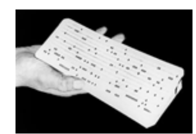
- Used with the earliest electronic computers (1940s)
  - Machines use vacuum tubes instead of transistors
- Programs are entered by setting switches or reading punch cards
- All instructions are numbers

- 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
```

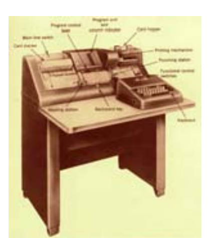
- Idea for improvement
  - Let's make it easier for humans by designing a high level computer language
- Result: high-level languages

## 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 was usually less efficient than the best possible 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 (become a major success)
- 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

## COBOL

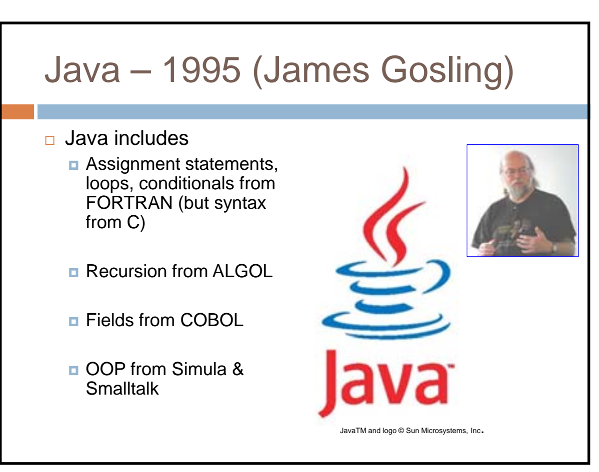
- COBOL = COmmon Business Oriented Language
- Developed by the US government (about 1960)
  - Design was greatly influenced by Grace Hopper
- Goal: Programs should look like English
  - Idea was that *anyone* should be able to read and understand a COBOL program
- COBOL included the idea of *records* (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 in the 60s
  - Smalltalk was developed at Xerox PARC in the 70s
- These languages included
  - Classes
  - Objects
  - Subclassing and inheritance

## Java – 1995 (James Gosling)

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



## In theory, 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

## … but even so

- Even standard Java features have some subtle aspects relating to object orientation and the way the type system works
- Let's touch on a few of these today
- We picked topics that will get you thinking about Java the way that we think about it!

## 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 and Static

- Some kinds of information is "static"
  - There can only be one instance
  - Like a "global variable" in C or C++ (or assembler)
- Object-oriented information is more "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
- In Java this distinction becomes very important

## Names

- The role of a name is to tell us
  - Which class is being referenced, although sometimes this is clear from the context
  - Which object is being referenced, unless we're talking about a static method or a static variable
- 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
  - this.i = i;

## The main Method

Can be called from anywhere

Associated with the class; don't need an instance (an object) to invoke it

No return value

Method must be named **main**

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

Parameters passed to program on command line or, in Eclipse, can be defined in the "Run" configuration dialog box (which the same as the "Debug" one…)

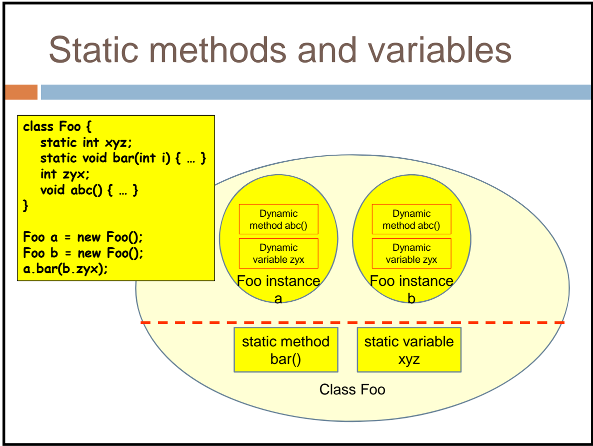
# Static methods and variables

- 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 methods and variables

```
class Foo {
    static int xyz;
    static void bar(int i) { … }
    int zyx;
    void abc() { … }
}

Foo a = new Foo();
Foo b = new Foo();
a.bar(b.zyx);
```




# Static methods and variables

```
class Thing {
    static int s_val;        // One for the whole class
    int o_val;               //  Each object will have its own personal copy

    static void s_method()  // Anyone can call this
    {
        s_val++;            // Legal: increments the shared variable s_val
        o_val = s_val;      // Illegal: Which version of o_val do we mean?
        o_method(s_val);    // Illegal: o_method needs an object reference
    }

    void o_method()
    {
        s_val++;            // Legal
        this.s_val++;       // Illegal: s_val belongs to the class, not object
        o_val = s_val;      // Legal:  same as this.o_val = s_val
        s_method();         // Legal:  calls the class method s_method()
        o_method();         // Legal:  same as this.o_method();
    }
}
```

# Avoiding trouble

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

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

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

# Class Hierarchy

superclass of Puzzle and Array and EPuzzle → Object

superclass of EPuzzle subclass of Object → Puzzle

Array

subclass of Puzzle and Object → EPuzzle

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

# Constructors

- Called to create new instances of a class
- Default constructor initializes all fields of the class  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();
```

## What about non-class variables?

- Those are *not* automatically initialized, you need to do it yourself!
- Can cause confusion

```
class Thing {
    int val;

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

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

this.val was automatically initialized to zero, but undef has no defined value! Yet the declaration looks very similar! In what way did it differ?

## Finalizers

- Like constructors but called when the object is deallocated
- Might not happen when you expected
  - Garbage collector decides when to actually deallocate an object
  - So objects can linger even when you no longer have a reference to them!
  - For this reason, we tend not to use finalizers – they add an undesired form of unpredictability

## 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) {
        ...
    }
}
```

## Static vs Instance Example

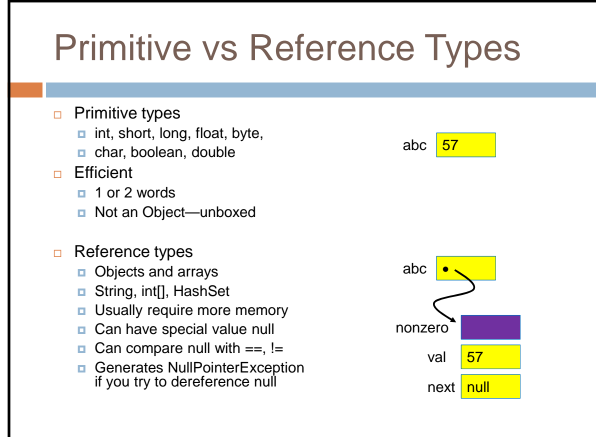
```
16
class Widget {
    static int nextSerialNumber = 10000;
    int serialNumber;
    Widget() {
        serialNumber = nextSerialNumber++;
    }
    public static void main(String[] args) {
        Widget a = new Widget();
        Widget b = new Widget();
        Widget c = new Widget();
        System.out.println(a.serialNumber);
        System.out.println(b.serialNumber);
        System.out.println(c.serialNumber);
    }
}
```

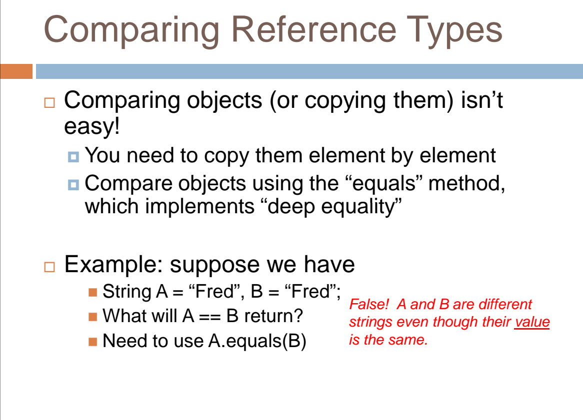
## Names

- 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

## 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 the 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

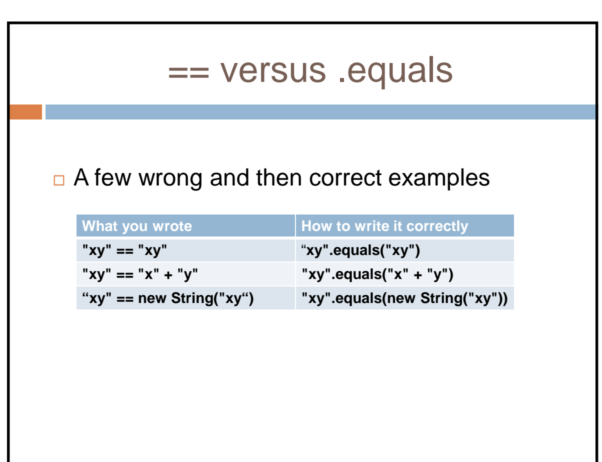



## Comparing Reference Types

- You can define "equals" for your own classes
- Do this by overriding the built in "equals" method:

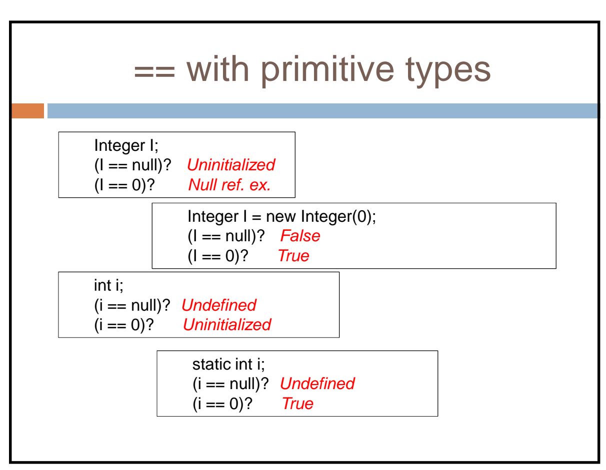
  boolean equals(Object x);
- But if you do this, must also override Object.hashCode() (more on this later)




| What you wrote | How to write it correctly |
|---|---|
| "xy" == "xy" | "xy".equals("xy") |
| "xy" == "x" + "y" | "xy".equals("x" + "y") |
| "xy" == new String("xy") | "xy".equals(new String("xy")) |

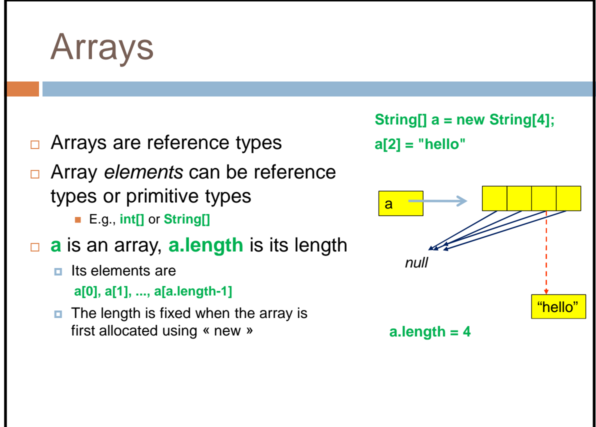
## == with primitive types

- Puzzle: why do Integer comparisons work?
  - Integer I = 7;
  - (I == 7)? *True, but not obvious why!*
  - (I == new Integer(7)) *False*
- … the first comparison only works because Java auto-unboxes I to compare it with int 7.
- If it had autoboxed the 7, the comparison would have failed! Lucky Java gets this right...



## Arrays

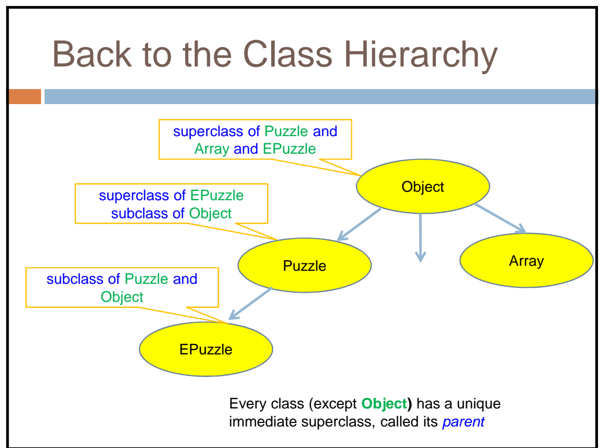
- Arrays are reference types
- Array *elements* can be reference types or primitive types
  - E.g., int[] or String[]
- a is an array, a.length is its length
  - Its elements are a[0], a[1], ..., a[a.length-1]
  - The length is fixed when the array is first allocated using « new »




## 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);
        }
    }
}
```

## Back to the Class Hierarchy




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

## Inheritance

- A subclass *inherits* the methods of its superclass
- Example: methods of the Object superclass:
  - equals(), as in A.equals(B)
  - toString(), as in A.toString()
  - … others we'll learn about later in the course
- … every object thus supports toString()!

## 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 actual, underlying object

## 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! This is very useful
- Caveat: cannot compose super more than once as in
    - super.super.m()

## Unexpected Consequences

- 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 foo = new B(); // upcasting
foo.m();            // would invoke private method in
                    // class B at runtime
```

## … a nasty example

```
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                          // Shadows variable i in class A.
    int f() { return -i; }              // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);        // Refers to B.i; prints 2.
        System.out.println(b.f());      // Refers to B.f(); prints -2.
        A a = (A) b;                    // Cast b to an instance of class A.
        System.out.println(a.i);        // Now refers to A.i; prints 1;
        System.out.println(a.f());      // Still refers to B.f(); prints -2;
    }
}
```

The “runtime” type of “a” is “B”!

## 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 (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

## … back to our earlier example

```
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                          // Shadows variable i in class A.
    int f() { return -i; }              // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);        // Refers to B.i; prints 2.
        System.out.println(b.f());      // Refers to B.f(); prints -2.
        A a = (A) b;                    // Cast b to an instance of class A.
        System.out.println(a.i);        // Now refers to A.i; prints 1;
        System.out.println(a.f());      // Still refers to B.f(); prints -2;
    }
}
```

The “declared” or “static” type of “a” is “A”!

## **Array** vs **ArrayList** vs **HashMap**

Three extremely useful constructs (see Java API)

- **Array**
  - Storage is allocated when array created; cannot change
  - Extremely fast lookups
- **ArrayList** (in **java.util**)
  - An “extensible” array
  - Can append or insert elements, access i’th element, reset to 0 length
  - Lookup is slower than an **array**
- **HashMap** (in **java.util**)
  - Save data indexed by keys
  - Can lookup data by its key
  - Can get an iteration of the keys or values
  - Storage allocated as needed but works best if you can anticipate need and tell it at creation time.

## **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 doesn’t contain key
  - Can use numbers.containsKey(key) to check this

## 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");
```

## Experimentation

- All of this adds up to some pretty confusing stuff you'll need to learn!
- Don't be afraid to experiment by writing little code fragments and seeing if they compile and what they do.
- But don't write random code hoping that it might work by some miracle.
- Examples in the Sun online JDK manual can be really helpful!
  - Cut and paste from Sun JDK manual is *not* considered to be a violation of academic integrity.
  - So go for it!

## Debugging

- Debugging
  - **Do not just make random changes, hoping something will work. This never works.**
  - Think about what could cause the observed behavior
  - Isolate the bug. Focus on the first thing that goes wrong.
- An IDE helps by providing a *Debugging Mode*
  - Can set breakpoints, step through the program while watching chosen variables
  - When program pauses at breakpoint, or dies, can look at values of variables it was using