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 Olin 216

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

---

**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**

- **Example code**
  - ADD R1 6
  - MOV R1 COST
  - SET R1 0
  - JMP TOP

- **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)

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

- **Result:** High-level languages

---

**High-Level Language**

- **Example code**
  0110 0001 0000 0110
  add reg1

- An idea for improvement
  - Use words instead of numbers

- **Result:** Assembly 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

- **The whole concept was initially controversial**
- **FORTRAN** (mathematical FORMula TRANslating system) was designed with efficiency very much in mind

---

**CS/ENGRD 2110**

Object-Oriented Programming and Data Structures

Spring 2012

Thorsten Joachims

Lecture 2: Java Review
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
```

- FORTRAN introduced many high-level language constructs still in use today
  - Variables & assignment
  - Loops
  - Conditionals
  - Subroutines
  - Comments

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)

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
  - 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();
        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 \textit{static} and \textit{instance} fields & methods of same class/object by (unqualified) name:
  - \texttt{serialNumber}
  - \texttt{nextSerialNumber}
- Refer to \textit{static} fields & methods in another class using name of the class:
  - \texttt{Widget.nextSerialNumber}
- Refer to \textit{instance} fields & methods of another object using name of the object:
  - \texttt{a.serialNumber}

- Example:
  - \texttt{System.out.println(a.serialNumber)}
  - \texttt{out is a static field in class System}
  - The value of \texttt{System.out} is an instance of a class that has an instance method \texttt{println(int)}
  - If an object must refer to itself, use \texttt{this}

A Common Pitfall

local variable shadows field

```java
class Thing {
    int val;
    
    boolean setValue(int v) {
        int val = v;
    }
}
```

- you would like to set the instance field \texttt{val = v}
- but you have declared a new local variable \texttt{val}
- assignment has no effect on the field \texttt{val}

The \texttt{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 \texttt{main}

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

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

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 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 \textit{signatures}
  - The \textit{signature} of a method is its name plus types of its parameters
- Example: \texttt{String.valueOf(...) in Java API}
  - There are 9 of them:
    - \texttt{valueOf(boolean)};
    - \texttt{valueOf(int)};
    - \texttt{valueOf(long)};
    - ...
  - Parameter types are part of the method’s signature

Primitive vs Reference Types

- Primitive types
  - \texttt{int, short, long, float, byte, char, boolean, double}
  - \texttt{x true}
  - \texttt{x true nonzero val}
- Reference types
  - \texttt{objects and arrays}
  - \texttt{String, int[], HashSet}
  - \texttt{usually require more memory}
  - \texttt{can have special value null}
  - \texttt{can compare null with ==, !=}
  - \texttt{generate NullPointerException}
  - \texttt{if you try to dereference null}

- \texttt{nonzero true val 23 next null}
"==" 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

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

Array Elements

- 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

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)
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".

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

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()`

Overloading Revisited

```java
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)
```

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, not the runtime type of the object referred to by r
- Shadowing variables is bad medicine and should be avoided

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