Advanced Java

Lecture 24

Summary

- `final` classes, methods, and variables
- Access levels (`public`, `protected`, `private`, default)
- Anonymous classes
- `instanceof`
- Multiple constructors
- `Enum`s
- Instance variable declarations
- `this`
- File IO
- Compiling and running Java

Final Classes and Methods

- `final` keyword can be used for classes, methods, and variables
- Similar meaning for classes and methods, but different for variables
- A `final` class cannot be subclassed and a `final` method cannot be overridden
- If a class is not intended by the programmer to be subclassable then it should be `final`
- A method should be `final` if it is integral to the proper functioning of the class
- Prevents someone else from incorrectly trying to extend the class or override the method

```
public final class DoNotSubclass{

}
```

```
public class Example {
  public final void reallyComplicated(){
  }
}
```

Final Variables

- You have used `static final` variables when declaring constants
- A `final` variable can only be assigned a value once and cannot be reassigned
- Must be assigned by the time the object has been constructed
- If the variable is a reference to an object, the variable cannot be changed but the object can still be mutated
- If you know that a variable should never be changed, good practice to make it final.

```
public class Example {
  public final int _myInt;
  public final Point _myPoint;

  public Example() {
    _myPoint = new Point(5,6);
    _myPoint.move(5,0); //okay
    _myPoint = new Point(2,3); //error!
   //other error: _myInt not assigned
  }
}
```

Access Levels

- Can apply to classes, methods, and instance variables
- `public` can be accessed from any class – use for methods (not variables) that you want to expose, i.e., all but a few `private` helper methods
- `protected` can be accessed from a subclass or any other class in the package (can't apply to classes, just methods and variables)
  - Java unfortunately doesn't have an access level which only allows the subclass access
  - Many of you discovered `protected` while doing Tetris. We do not encourage its use because it violates encapsulation principles (we'll take off if we see it in final projects)
- Default or package protected (no access modifier) can be accessed from any class within the package
  - Violates encapsulation as well, and not allowed in CS15
- `private` can only be accessed within the class and any inner class

Anonymous Classes (used in Android)

- Anonymous classes are classes without a name
- Allow you to define a class and instantiate it in one step
- Use when you need to use an inner class just once
- Has to either implement an interface or extend from another class
- Pros:
  - Use in place of an inner class
  - Allows you to define a subclass or implement an interface on the fly
- Cons:
  - Can produce hard to read code
  - Code is not reusable

Anonymous Class Syntax

- Syntax: `new X(){`

  `Class Definition`

`}`

  - Where X is the superclass of or the interface implemented by the anonymous class

- If you need a constructor with parameters, pass in the appropriate arguments inside the parentheses right before the first {.

```
public class MainPanel{
  public MainPanel(){
    JButton quit = new JButton();
    quit.addActionListener(new
       ActionListener(){
       public void actionPerformed(
          ActionEvent e){
          System.exit(0);
       }
    });
  }
}
```

Instanceof

- `instanceof` allows you to test if an instance (reference stored at a variable) is an instance of a certain class
  - Tests actual type of instance, but will return true for superclasses as well
- Useful in very limited circumstances
- Should not replace good polymorphic design
- “Anytime you find yourself writing code of the form ‘if the object is of type T1, then do something, but if it's of type T2, then do something else’, slap yourself.” – Scott Meyers (author of Effective C++ and Brown Ph.D.)

```
public void contrivedExample() {

  Animal creature1 = new Dog();
  Animal creature2 = new Shark();
  Animal creature3 = null;

  creature1 instanceof Dog //true
  creature1 instanceof Shark //false
  creature2 instanceof Animal //true
  creature3 instanceof Object //false
}
```

Multiple Constructors (1/2)

- Method overriding applies to constructors too
  - You can define multiple constructors for a class for the convenience of yourself or other programmers
  - But, they must have different method signatures

```
public class Book{

    private String _title;
    private String _blurb;

     public Book(String title, String blurb) {
     _title = title;
     _blurb = blurb;
     }

     public Book(String title) {
     _title = title;
     _blurb = “Default Blurb”
     }

     public Book() {
     _title = “Default Title”;
     _blurb = “Default Blurb”;
     }
}
```

Multiple Constructors (2/2)

- Frequently, to avoid repeating code, multiple constructors will call each other
- Note using `this` to call other constructors

- Can also create a *copy constructor*, which sets up current instance to mirror the given parameter

```
public class Book{
     private String _title;
     private String _blurb;

     public Book() {
          this("Default Title");
     }

     public Book(String title) {
          this(title, "Default Blurb");
     }

     public Book(String title, String blurb) {
          _title = title;
          _blurb = blurb;
     }
     public Book(Book other) {
          _title=other.getTitle().copy();
          _blurb=other.getBlurb().copy();
     }
     //getters and setters elided
}
```

Enums (1/5)

- The `enum` type falls somewhere between primitives (like `int` or `double`) and objects—it has methods, but can be treated like a primitive
- You can think of an enum as your own primitive type with a limited set of possible values. (Imagine if integers could only be 1, 2, or 3).
- We usually use enums to model a fixed set of constants, such as compass directions
- If you're doing Pacman, you'll be using these!
- Naming convention is ALL_CAPS
- Enums are not instantiated. Rather, from any other class, you can refer to an enum like this:

```
public enum TrafficLight {

    RED, GREEN

}
```

```
TrafficLight.RED
```

Enums (2/5)

- Because enums are a lot like primitives, one of the cool things you can do with them is use them in switch statements:
- Note that enums define types, just like classes and interfaces define types. So we can declare variables of our enum type:
  - `TrafficLight favLight = TrafficLight.GREEN;`
- Or we can pass enums around in parameters and return them from methods, as in the example above.

```
public TrafficLight getOpposite(TrafficLight light){
switch(light) {
    case TrafficLight.RED:
        return TrafficLight.GREEN;
    case TrafficLight.GREEN:
        return TrafficLight.RED;
}
}
```

Enums (3/5)

- Enums can also define their own methods!

```
public enum TrafficLight {

     RED, GREEN;

     public TrafficLight getOpposite() {
          switch(this) {
               case RED:
                return GREEN;
               case GREEN:
          return RED;
     }
     }
}
```

- Now, if we want to get the opposite direction:

```
TrafficLight light = TrafficLight.RED;
TrafficLight opposite = light.getOpposite();
```

Enums (4/5)

- You can think of the Enum type as a superclass and each of its constants as a subclass
- This means that each enum can have its own methods

```
public enum Operation {
  PLUS {
    public double doOperation(double a, double b){
      return a+b;
    }
  },
  MINUS {
    public double doOperation(double a, double b){
      return a-b;
    }
  },
  MULTIPLY {/*elided*/},
  DIVIDE {/*elided*/};
  public abstract double doOperation(double a,
double b);
}
```

Enums (5/5)

- But there's more! Enums can also have their own instance variables and constructors.
- Even though you can define constructors for enums, you still never instantiate them!
- Enum constructors cannot be public
- Rather, each constant is declared with parameters that get passed into the constructor and initialize the instance variables of that constant

```
public enum LimitedColor{

    //constants are initialized here
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    //declare instance variables common to eac
   // constant
    private int _r, _g, _b;

    //constructor
    private LimitedColor(int r, int g, int b) {
        _r = r;
        _g = g;
        _b = b;
    }

    public int getRed() {
        return _r;
    }
}
```

Instance Variable Declaration

- Did you know you could declare instance variables like this?

```
public class Andy {

    private int _age = 1000000;
    private boolean _looksOld =
   false;
}
```

- Might see this, but we want you to use the constructor to initialize instance variables

- Yep, you can initialize instance variables when you declare them. You can even do this!

```
public class Lecture {
    private Water _andysWater = new Water();
  }
```

Another Use of "this" (1/2)

- As you know, this is used to refer to the instance of the class we are currently defining
- Well, it's actually optional. If a method is called but not on an instance, Java will automatically look for the method in the current class or one of its superclasses
- From now on, not required to use this when calling methods

```
public class MyClass {

    public MyClass() {
        this.myMethod();
    }

    public void myMethod() {
        //something silly
    }
}
```

```
public class MyClass {

    public MyClass() {
        myMethod();
    }

    public void myMethod() {
        //something silly
    }
}
```

Another Use of "this"

- In fact, `this` can be used to access any member of the current class, including instance variables
- In fact, some people will write constructors like this:
- But we ask you to use `this` when calling a class's own methods for the sake of consistency
- It is also important to remember that you are calling the method on an instance of an object
- Similarly, we **don't** use `this` to refer to instance variables, because that is inconsistent from the way we normally use variables
- Still required to use _ for instance variables

```
public class MyClass {

     private int a, b;
     public MyClass(int a, int b) {
          this.a = a;
          this.b = b;
     }

}
```

Handling Input (1/3)

- Can think of many scenarios where we want to read contents of some file or ask user to supply us with information via command line
- How do we handle this input in Java?
  - Answer: with streams!
- You can think of streams as some sequence of bytes flowing from one source (such as a file or command line input) to some destination (i.e., your program)
- It can be messy to deal with raw byte data directly
  - Have to define how many bytes to read in at a time
  - Have to explicitly allocate memory to store each sequence of bytes we read in
  - There has to be a better way …

Handling Input (2/3)

- `java.util.Scanner`!
  - converts the raw byte data into character data that human beings can understand
  - one common practice is to make a `File`, handles communication between Java and the Operating System, and pass it to the `Scanner`.
  - Use nextLine() to get the next line of the file in String format.

```
public class ReadData {
  public ReadData() {
    Scanner input;
    try {
     input = new Scanner(new java.io.File(
         "/home/gchatzin/someFile"));
       String line;
       while((input.hasNextLine()){
     line=input.nextLine();
         //do something with data
       }
    } catch (FileNotFoundException e) {
       // handle the exception somehow
    }
  }
}
```

Handling Input (3/3)

- What about files with `floats`, `ints`, or `bytes`, instead of characters?
- Do same thing, except use `nextFloat`.
- Similar methods for the other data types as well.
- File I/O can get very complicated. For Sketchy, we provide support code that handles some of the complexity (ex: exceptions).

```
public class ReadData {
  public ReadData() {
    Scanner input;
    try {
     input = new Scanner(new java.io.File(
         “/home/gchatzin/someFile”));
       float value;
       while((input.hasNextFloat()){
     value=input.nextFloat();
         //do something with data
       }
    } catch (FileNotFoundException e) {
       // handle the exception somehow
    }
  }
}
```

Java Without Eclipse

- Java does exist outside Eclipse (or any IDE)
  - You can edit .java files in general text editors, like Sublime, and then run the code from the command line
  - Important to understand in order to work with other programming languages (and for CS16)
- Remember, there are 2 steps to running Java code
  - Compilation - converting your code to a format which the Java Virtual Machine can run
    - Finds all of your syntax errors, etc.
    - Eclipse does this constantly in the background, so you never had to explicitly compile your code
  - Execution - actually launching the program

Compiling Your Code (1/2)

- In CS15, you can compile your code by running javac *.java in the directory where your code is located
- Notice that we did not have to specify the support code's jar location, like we did when setting up your Eclipse project.
- This works because we have done some magic behind the scenes for you, but when you are no longer a CS15 student, this magic goes away.
- So, how does the compiler know where the support code is?

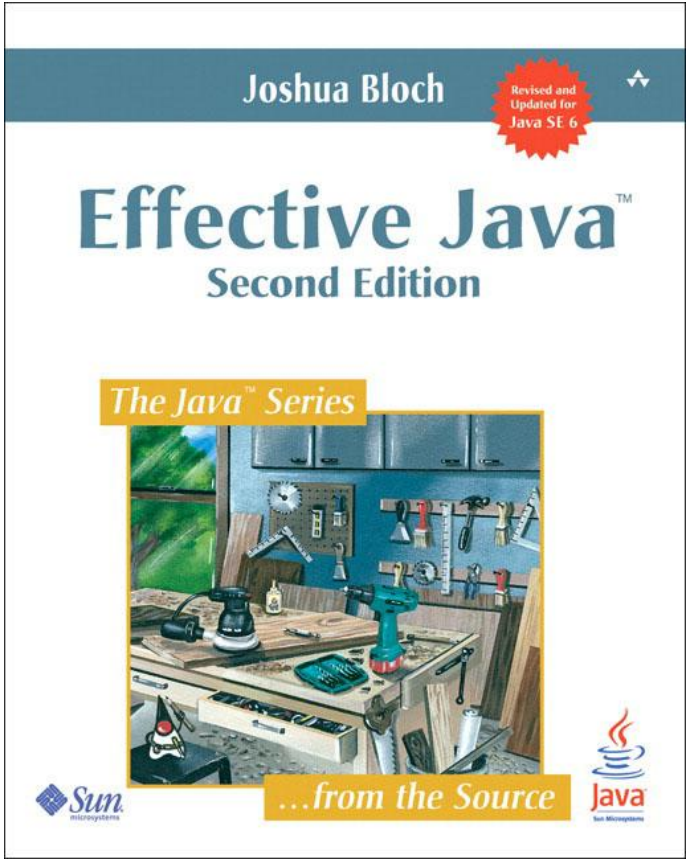
Compiling Your Code (2/2)

- The classpath!
  - A list of directories with java code
  - Everyone has their own classpath
  - Tells the java compiler where the java code is
  - Your current directory is always included in the classpath
- See what's on your classpath by running:
  echo $CLASSPATH
- We have added /home/<yourLogin>/course/cs015 and /course/cs015/lib to your classpath
  - this is why you can use javac *.java to compile
- To compile using directories that are not on your classpath use:
  javac –cp /home/gchatzin/someDir:/home/gchatzin/otherDir *.java

Running Your Code

- To run your code, you can run java <package>.App from the directory with the same name as <package>
- This works for the same reason that compiling does
  - We have added several directories to your classpath
- Normally you must be located in the directory one level above <package> to run the program
- For example, if you were trying to run Tetris.App which is contained in ~/course/cs015/Tetris/ then you would need to run java Tetris.App from ~/course/cs015/

Effective Java



Terminal Commands

- Navigation
- cd, cd .., cd ~, cd -
- pwd
- mkdir
- ls
- ls –a
- ls –l
- rm
- rmdir
- Rm –rf
- mv
- cp

- Useful to Know
- Tab complete
- Up/down arrows
- ctrl-z and bg
- man
- cat
- sort
- ctrl-a and ctrl-e
- Using aliases

- Vim
- Text editor
- w, q, wq, q!
- :/<toFind>
- :<lineNum>
- :set nu
- u for undo
- y to copy
- p to paste

- Fun Stuff
- cowsay
- sl
- figlet
- floor
- snoop
- finger