

Advanced Programming

Lecture 7

Classes, Class *Class* and Class Reflection

Classes

- Classes in programming
- Class *Class*
- Reflection
- Class Reflection
- Inner classes
- Accessing resources from a Jarfile

Class

- A **class** is a fundamental unit within object oriented programming
- Keyword **class** in Java is used to declare a class structure
- Classes are used as templates for objects
- Classes can be used for utility functions also, e.g. the Math class in Java (no object needed)

Classes and the main method

- Classes in Java do not require a main method
- The main method is a method that invokes an application, not all classes will be run as applications
- In a project that runs as an application there is usually only one class that contains the main method, this ‘kicks-off’ the application

Classes...back to HelloWorld

```
public class HelloWorld {  
    public static void main(String[] args) {  
        HelloWorld hellProg = new HelloWorld();  
        hellProg.sayHello();  
    }  
  
    public void sayHello() {  
        System.out.println("Hello World");  
    }  
}
```

This program is acting as a **class AND an application** as it has to create an instance of itself and call a method in the **main**

This is a simple application since it has a main method, but it also has an object created called hellProg, we need to do this to call the method sayHello

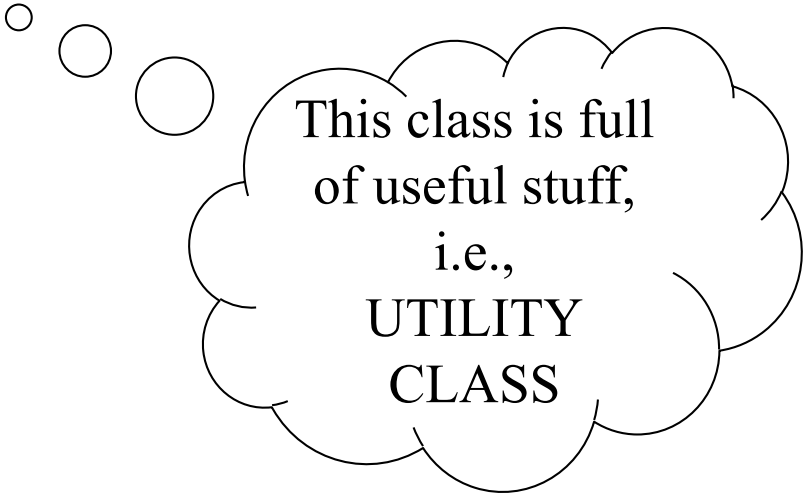
Utility Classes

- Some classes are designed simply to supply a list of useful behaviours, these are called utility classes
- An example of a utility class within Java is the Math class, NOTE: You do not need to create a Math object to call maths methods in Java:

`Math.pow(2,2);` //No need to create a Math object!!!

Utility class - CollegeHelper

```
public class CollegeHelper {  
  
    public static final int MINCLASSSIZE = 10;  
    public static final int MAXCLASSSIZE = 30;  
    public static final int PASSGRADE = 40;  
  
    public static boolean classSizeWithinLimit(int classSize) {  
        if(classSize>=MINCLASSSIZE && classSize<=MAXCLASSSIZE) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
}
```



This class is full
of useful stuff,
i.e.,
UTILITY
CLASS

Using the CollegeHelper

```
if(!CollegeHelper.classSizeWithinLimit(sampleClassList.size  
    ())) {  
    System.out.println("Class not within limits");  
}  
System.out.println("Pass grade is " +  
    CollegeHelper.PASSGRADE);
```

- Note: Just like the Math class we DO NOT need to create an object of the CollegeHelper because this class is not designed to represent objects, it's just full of useful stuff...and that's why it's an example of a UTILITY class

Why use *static* and *final* in CollegeHelper and Utility Classes?

- An attribute that is declared as *final* cannot be changed throughout the course of the program...in our example this makes sense since the MAXCLASSSIZE will not change
- An attribute or method that is declared static is associated per class not per object...this makes sense since we never create an object of type CollegeHelper, and we don't want to!!!

Class *Class*

- The Java runtime environment stores information about objects (runtime type identification)
- This information can be accessed by a programmer using a Class called *Class*

Using class *Class*

- Suppose you had created a Student object like this:

```
Student student1 = new Student()
```

- It's possible to get a Class reference to the Student object student1 using the following:

```
Class classInfo = student1.getClass();
```

The *getName()* method

- One commonly used function in the Class *Class* is the *getName()* function

```
student1.getClass().getName();
```

```
//this is a Class reference to this class and
```

```
//allows information to be retrieved for that
```

```
//class, in this case the name of the class
```

The *forName()* method

- It's also possible to obtain an object of a class by passing the name of a class to a method called *forName()*, all you need is the correct name for the class [so this is an alternative way to create an object, other than using keyword `new!!!`]

```
String className = "Student";
```

```
Class myClassObj = Class.forName(className);
```

The *.class* shorthand

- It's also possible to obtain a Class reference by using a *.class* shorthand:

```
Class myClassObj = Student.class;
```

or

```
student1.class.getName(); //Gets name of class
```

Class method *newInstance()*

- The class *Class* provides a useful method for the retrieval of an object of a class

```
Student student2 = student1.getClass().newInstance();
```

or

```
String className = "Student";
```

```
Object student3=Class.forName(className).newInstance();
```

Creating objects with parameters

- As seen before most objects require information to be passed at the time of creation
- In order to create a Student object and pass constructor information, e.g., `new Student("Paul", 26,1)`; you cannot do this using the class *Class* as shown earlier
- In order to pass constructor information you need to use class Reflection

Class Reflection

- Classes have the capability to ‘investigate’ the functionality of another class using what is called *Class* reflection
- It’s akin to you providing a CV or at an interview describing the skills and capabilities that you have...you could say it’s a reflection of your eligibility for a job

Class Reflection

- Java, therefore, can ask a class, using reflection, to provide lists of methods and constructors etc. that are available at runtime
- This means that classes can be dynamically imported into a Java program and used on the fly...JavaBeans makes great use of this capability

Reflexive classes

- A class that can analyze the capabilities of classes is called a *reflexive* class
- There is a package in Java where you will find the reflexive classes: *java.lang.reflect*

Class Reflection

- There are three classes to describe the attributes, behaviours and constructors found in a class:
 - Field – this represents attributes (in class Student it would be either name, year, age)
 - Method – this represents a method in a class
 - Constructor – this represents a constructor in the class
- These are all found in the *java.lang.reflect* package

Field, Method and Constructor

- Each of the classes has a method called *getName()*, returning the name of the field, method or constructor
- The Field class has a *getType()* method for returning the type of the attribute/field
- The Method and Constructor classes have the methods to report return type and parameters

Field, Method and Constructor

- You can even find out the access of each field, method or constructor using the *getModifiers()* method, i.e. are they private, public, protected or package access.
- The above returns an integer representation of the access, the Modifier class provides methods like *isPublic()*, *isPrivate()* etc. to translate the Modifiers
- The *toString()* method can actually print the modifier type in human-readable form, e.g. “private”

Using Class *Class* for reflection

- Class *Class* provides the following methods:
 - `getFields()` [returning ARRAY of public fields]
 - `getMethods()` [ARRAY of public methods]
 - `getConstructors()` [ARRAY of public constructors]
- Using a loop to traverse the above arrays and calling the *getName()* method on each element of the array could be used to print all details of a class

Using Class *Class* for reflection

- Class *Class* also provides the following methods:
 - `getDeclaredFields()` [returning ARRAY of ALL fields]
 - `getDeclaredMethods()` [ARRAY of ALL methods]
 - `getDeclaredConstructors()` [ARRAY of ALL constructors]
- Again, using loops to traverse the above arrays and calling the *getName()* method on each element of each array could be used to print all details of a class

Inner Class

- An *inner class* is a class that is defined inside another class
- Inner classes are often used in GUIs for capturing events (you may have done this last year but not had the time to think about how it worked)

Inner Classes

- Why use inner classes?:
 - An object of an inner class can access the implementation of the object that created it, including data that would otherwise be inaccessible if declared private
 - Inner classes are like ‘secret’ classes as they can be hidden from other classes in the same package

Inner Classes

- Why use inner classes?:
 - It's possible to create anonymous inner classes, this can be handy sometimes for easy callbacks
 - They tend to be handy from a modularization perspective, when writing GUIs the inner classes help to clearly define event handlers and separate the event handling code from the GUI class itself

Inner Class references outer class

- The inner class gets an implicit reference to the outer class
- To access the outer class
OuterName.this.whatever

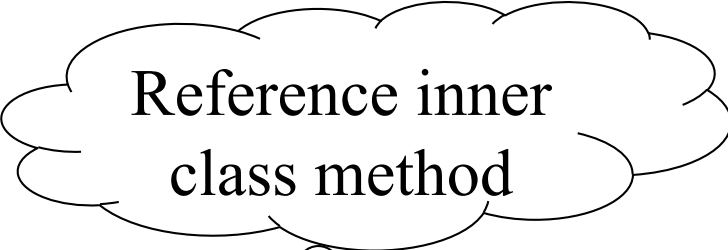
Inner Class references outer class

```
public class OuterClass {  
    int outerInt;
```

```
public static void main(String[] args) {  
    new OuterClass().new Inner().accessOuter();  
    //This is an outer class  
}
```

```
private class Inner {  
    //This is an inner class  
    public void accessOuter() {  
        OuterClass.this.outerInt=1;  
    }  
}
```

```
}
```



Reference inner
class method



References
outer class

Anonymity [Classes and Objects]

- The English word “anonymity” means not having a name
- In Java it means not having a reference
- For example you can create a Student using a reference or NOT using a reference:

```
Student student1 = new Student(); //referenced  
student1.getName(); //use reference
```

Anonymity [Classes and Objects]

- To create a student without a reference (an anonymous object reference) you may do the following:

```
(new Student()).getName(); //An anonymous object
```

- This object can still have methods called on it and can be passed to any method that accepts Student objects, however, there is no explicit named reference to the object when it was created

Anonymous Inner Class

- Classes that are inner classes but have no explicit reference
- They are quick but not easy, and are invisible to the outsider classes


```
public class AnonymousInnerClass extends JFrame {

    public static void main(String[] args) {
        JButton button1;
        AnonymousInnerClass GUI = new AnonymousInnerClass();
        button1 = new JButton("Say Hello");
        GUI.getContentPane().setLayout(new FlowLayout());
        GUI.getContentPane().add(button1);
        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Hello");
            }
        });
        GUI.setSize(300, 300);
        GUI.setVisible(true);
    }
}
```

Class Reflection and Jar Resources

- When a Java project is deployed as a Jarfile it changes the way that resources such as images and multimedia are accessed
- Class Reflection can be used to access the resources available to the Jarfile
- Instead of accessing the images/media from the project directory directly, access to resources becomes relative to Jarfile path

Class Reflection and Jar Resources

- Within a non-Jarfile project you can create an image directly using the image URL, e.g.:

```
resourceImage = new ImageIcon(imageURL);  
imageLabel.setIcon(resourceImage);
```

- The image URL above represents the path to the image in the project directories (e.g. there could be an ‘images’ directory below the working directory for the project) e.g.

```
String imagePath = "file:/" + System.getProperty("user.dir") + "//  
images/";
```

Class Reflection and Jar Resources

- Within a Jarfile this approach will not work as the resources directory will be relative to the Jarfile, i.e. the Jarfile could be downloaded ‘anywhere’
- Class reflection can be used to find the resources relative to the Jarfile directory

```
imgURL = TestResourceAccessFromJar.class.getResource(?path?)
```

- The above line of code uses class reflection to access the class’s resources using the `getResource()` method, the relative path to the resource is included as a parameter

Class Reflection and Jar Resources

- The relative path will be the path to the resource as it is packaged in the Jarfile
- For example if the directory “images” was added to the Jarfile then the relative path is ‘/images/myimage.gif’
- See the TestResourceAccessFromJar.java class on StudentShare and Moodle!

Labwork

Using the CollegeHelper class in this lecture create a class called ClassInvestigater (make it a JFrame) that investigates all of the names of the fields and methods in the CollegeHelper class using class reflection and prints them to a JTextArea. [Hint: get an array of all fields and all methods and loop through each array, call *getName()* on each element of each array]