Human-Computer Interaction

Practical session – Java

Outline

- OOP concepts in Java: Encapsulation, Inheritance, Polymorphism and Abstraction
- Exceptions, ArrayList, and Scanner

Inheritance

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another.

We group the "inheritance concept" into two categories:

subclass (child) - the class that inherits from another class
superclass (parent) - the class being inherited from

To inherit from a class, use the extends keyword.
In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

Example

```
class Vehicle {
 protected String brand = "Ford"; // Vehicle attribute
public void honk() { // Vehicle method
System.out.println("Tuut, tuut!");
 }
}
 class Car extends Vehicle {
private String modelName = "Mustang"; // Car attribute
 public static void main(String[] args) { // Create a myCar object
Car myCar = new Car();
// Call the honk() method (from the Vehicle class) on the myCar object
myCar.honk();
// Display the value of the brand attribute (from the Vehicle class) and the
value of the modelName from the Car class
System.out.println(myCar.brand + " " + myCar.modelName);
}
}
```

The **brand** attribute in **Vehicle** was set to a protected access modifier. If it was set to private, the Car class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

The final Keyword

```
final class Vehicle {
...

 }
 class Car extends Vehicle {
...
}
```

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users.
To achieve this, you must:

- declare class variables/attributes as private
- provide public **get** and **set** methods to access and update the value of a private variable

```
public class EmployeeRecord {
    private String name;

    public String getName(){
        return name;

    }
    public void setName(String nm){
        name=nm;

    }
    public static void main(String [] args){
        EmployeeRecord emp = new EmployeeRecord();

        emp.setName("Helen");
       String e= emp.getName();
        System.out.println(emp.name);

    }
}
```

Polymorphism

Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.

Polymorphism uses those methods to perform different tasks.

This allows us to perform a single action in different ways.
For example, think of a superclass called Animal that has a method called animalSound().

Subclasses of Animals could be Pigs, Cats, Dogs, Birds.

And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
public class Animal {

    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}
class  Bird extends Animal {
    public void animalSound() {
        System.out.println("The Bird tweets ");
    }
} class Dog extends Animal {
    public void animalSound() {
        System.out.println("The dog bark");
    }

}
```

Abstraction

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces**.
The abstract keyword is a non-access modifier, used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body.
The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

- A class which contains the **abstract** keyword in its declaration is known as abstract class.
- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (**public void get();**)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

```
abstract class Animal {
 public abstract void animalSound();
public void sleep() {
System.out.println("Zzz");
}
}
```

Try to generate an object.

```
Animal myObj = new Animal(); // will generate an error
```

```java
// Abstract class
public abstract class AbstractAnimal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("asleep");
    }
}

// Subclass (inherit from Animal)
class Dogs extends AbstractAnimal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The Dog Bark");
    }
}

class Main {
    public static void main(String[] args) {
        Dogs dog = new Dogs(); // Create a Dogs object
        dog.animalSound();
        dog.sleep();
    }
}
```

Interface class

- The interface in Java is a mechanism to achieve abstraction.
- There can be only abstract methods in the Java interface, not the method body.
- It is used to achieve abstraction and multiple inheritances in Java using Interface.
- The *implements* keyword is used to inherent an Interface

Example:

```
public interface InterfaceAnimal {
   // interface

      public void animalSound(); // interface method (does not have a body)
      public void run(); // interface method (does not have a body)
   }
```

- Try to implement this interface

Exceptions

```
public class Test {
   public static void main(String args[]) {

      String [] names = {"James", "Larry", "Tom", "Lacy"};

            System.out.println(names[4]);
   }
}
```

Try to compile and execute the above program.

in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4
at Test.main(Test.java:7)

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs:

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Catching Exceptions

- A method catches an exception using a combination of the **try** and **catch** keywords.
- A try/catch block is placed around the code that might generate an exception.
- Code within a try/catch block is referred to as *protected code*, and the syntax for using try/catch looks like the following:

```
try {
// Protected code
} catch (ExceptionName e1)
{
// Catch block
}
```

Example

```
public class Test {
   public static void main(String args[]) {

   try {
      String[] names = {"James", "Larry", "Tom", "Lacy"};

      System.out.println(names[5]);
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception Thrown :" + e);}
   System.out.println("out of the block");
   }
   }
```

Output

```java
import java.io.File;
import java.io.FileReader;


public class FilenotFound_Demo {
    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

Multiple Catch Blocks

- A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:
- Syntax

```
try {
// Protected code } catch (ExceptionType1 e1) {
// Catch block } catch (ExceptionType2 e2) {
 // Catch block
} catch (ExceptionType3 e3) {
// Catch block }
```

Example

```
try {

    File file = new File("E://file.text");
    FileReader fr = new FileReader(file);
} catch (FileNotFoundException e){
    System.out.println("Exception thrown" + e);
}
```

ArrayList

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);
    }
}
```

ArrayList

The ArrayList class is a resizable array, which can be found in the java.util package. The difference between a built-in *array* and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).

While elements can be added and removed from an ArrayList whenever you want.

The syntax is also slightly different:

Example

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList
object
```

The above example is to create an ArrayList object called **cars** that stores strings:

Methods

- Add Items
- [object name].add(var)
- Access items
- [object name].get(index)
- Change items
- [object name].set(index, var)
- Remove items
- [object name].remove(index)
- Remove all items
- [object name].clear()

Examples

```
import java.util.ArrayList;

public class MyArrayList {

	public static void main(String[] args) {
		ArrayList<String> cars = new ArrayList<String>();
		cars.add("Volvo");
		cars.add("BMW");
		cars.add("Ford");
		cars.add("Mazda");
		for (int i = 0; i < cars.size(); i++) {
			System.out.println(cars.get(i));
		}
	}
}
```

```
import java.util.ArrayList;

public class Books {
   String book;
    ArrayList <String> bookList;
    public static void main(String[] args ){

    }

}
```

Important NOTE

- **ArrayList is a class, and it takes Objects only (means it does not take primitives).**
-

Examples

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid


ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Wrapper classes

- **int = Integer**
- **double = Double**
- **char = Character**
- **Boolean = Boolean**
- **long = Long**

Example

```
public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);
    for (int i = 0; i < myNumbers.size(); i++) {
        System.out.println(myNumbers.get(i));
        }
}
```

Scanner class

- Scanner class is used to enable user's input.
- The Scanner class can be found in the java.util package.

- Example:

```
Scanner myObj = new Scanner(System.in); // Create a Scanner object
```

- The following example is a java program to add tow numbers.

Example

```
import java.util.Scanner;

public class MyScannerClass {
  public static void main(String[] args) {
    int x, y, sum;
    Scanner myObj = new Scanner(System.in); // Create a Scanner object
    System.out.println("Type a number:");
    x = myObj.nextInt(); // Read user input

    System.out.println("Type another number:");
    y = myObj.nextInt(); // Read user input

    sum = x + y;  // Calculate the sum of x + y
    System.out.println("Sum is: " + sum); // Print the sum
  }
}
```

Thank you