8. Features of Java

In the following part of the notes some important Java features are discussed. However this does not mean that such features are possessed by Java alone.

8.1 The Java API

Java API ("Application Programming Interface") is a huge collection of library routines. In the Java API, classes and interfaces are grouped in packages.

All these classes are written in Java and run on the JVM. Java classes are platform independent but the JVM is not platform independent. You will find different downloads for each OS.

8.2 Arguments Passing by Reference and by Value

Parameter Passing is a mechanism used to pass parameters to a procedure (subroutine) or function (in Java both procedures and functions are called 'methods'). The most common methods are to pass the value of the actual parameter ('*call by value*'), or to pass the address of the memory location where the actual parameter is stored ('*call by reference*'). The latter method allows the procedure to change the value of a variable that is passed to a parameter, whereas the former method guarantees that the procedure will not change the value of a variable passed to a parameter. Other more complicated parameter-passing methods exist.

Consider the following parts of a Pascal program.

```
procedure P (a : integer; VAR b : integer);
begin
     a := 3;
     b := 5
end;

begin
     alpha := 1;
     gamma := 50;
     P (alpha, gamma);
     writeln (alpha, gamma);
end.
```

Diagram 89. A Pascal Program with two kinds of Parameter-Passing: by Value and by Reference

When the procedure P is called, the parameter 'a' is assigned the value of 'alpha' (which is 1). The parameter 'b' however is assigned the address of 'gamma'. Then inside the procedure the value of 'a' is changed to 3 and the value of 'b' is changed to 5. This causes no change in the variable 'alpha' but it changes the value of the parameter 'gamma' from 50 to 5.

In Java everything is passed by value. This needs some explanation. Since the value of an object is its reference, when we pass this reference (by value) it has the same effect as passing by reference. So remember that the values of variables are always primitives or references.

The following program, consisting of two classes, shows what happens when a primitive value and a reference value are passed by value.

```
class ValRef
{
    public static void main(String[] args)
    {
        int x=0;
        //x is a variable that has a primitive value equal to 0
        Integer p = new Integer ();
        //p is an object containing an integer equal to 0

        giveMeATen (x);
        System.out.println(x);
        //the value 0 is displayed i.e. the value of x is not
        //changed

        giveMeATenAgain (p);
        System.out.println (p.i);
        //the value 10 is displayed i.e. the integer i of object p
        //is changed
    }

    static void giveMeATen (int y)
    {   y = 10; }

    static void giveMeATenAgain (Integer k)
    {   k.i = 10; }
}

public class Integer
{
    int i;

    public Integer()
    {
        i = 0;
    }
}
```

Diagram 90. A Java program with a Primitive Value and a Reference Value passed by Value.

8.3 Static Classes

The following Java program consists of two classes "StaticDemo" and "StaticDemoProg". In the class "StaticDemo" two variables are defined:

- normalVar is a normal instance variable. This means that every object of the class StaticDemo would have a particular value for its variable normalVal. Say object1 would have its variable normalVal equal to 7 and object2 would have its normalVal equal to 12.
- staticVar has different features. It is like a global variable and if one object changes its value this value is changed for all objects that are instances of the class StaticDemo.
- Note that the static variable staticVar can be called from any object (e.g. object1.staticVar = 25) or from the class itself (e.g. StaticDemo.staticVar = 9).

```
class StaticDemo
{
   int normalVar; //this is a normal instance variable
   static int staticVar; //this is a static variable
}
class StaticDemoProg
{
   public static void main (String args[])
   {
      StaticDemo object1 = new StaticDemo();
      StaticDemo object2 = new StaticDemo();

      object1.normalVar = 12;
      object2.normalVar = 15;

      System.out.println ( object1.normalVar + ", " +
                                                 object2.normalVar );

      object1.staticVar = 25;

      System.out.println ( object1.staticVar + ", " + object2.staticVar );

      StaticDemo.staticVar = 9;

      System.out.println ( object1.staticVar + ", " + object2.staticVar );
   }
}
```

Diagram 91. Instance and Static Variables

Therefore the output of the above program is:

```
12, 15
25, 25
9, 9
```

Methods can also be “static” methods or “instance” methods. The difference between a static method and a normal (instance) method is that the static method can be called through its class name without any object of that class being created.

The following Java program is made up of two classes. There are no instance methods. To call the method “nicePlace” no object is created. The output, after executing “StatMethProg” is the string “Msida”.

```
class StatMeth
{
   static String nicePlace()
   {
      return "Msida";
   }
}

class StatMethProg
{
   public static void main (String args[])
   {
      System.out.println ( StatMeth.nicePlace() );
   }
}
```

Diagram 92. Program with no Instance Methods

Static methods are restricted to call only other static methods and static data.

A class that contains a static method or any other static property like a static variable is itself called static. However it can also contain other non-static methods and properties.

8.4 Abstract Methods and Classes

An abstract class is a class that is declared ‘abstract’. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, the class itself must be declared abstract, as in the following class.

```
public abstract class GraphicObject
{
        // declare fields
        // declare non-abstract methods

        abstract void draw();
}
```

Diagram 93. A Class that included Abstract Methods has to be Abstract itself.

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line colour, fill colour) and behaviours (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviours are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.

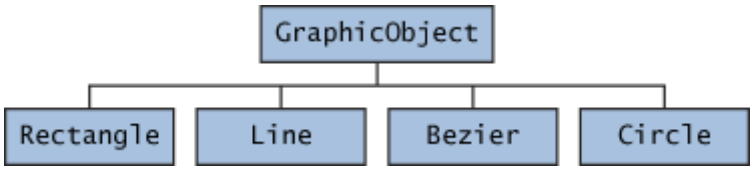

Diagram 94. Classes Rectangle, Line, Bezier and Circle inherit from GraphObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject
{
   int x, y;
   ...
   void moveTo(int newX, int newY)
```

```
{
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Diagram 95. The Abstract Class GraphicObject

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods as shown in the following program scheme.

```
class Circle extends GraphicObject
{
    void draw()
    {
       ...
    }

    void resize()
    {
       ...
    }
}

class Rectangle extends GraphicObject
{
   void draw()
    {
       ...
    }

   void resize()
    {
       ...
    }
}
```

Diagram 96. The Classes Circle and Rectangle derived from the Class GraphicObject

8.5 Errors and Exceptions

A program often encounters problems as it executes. It may have trouble reading data, there might be illegal characters in the data, or an array index might go out of bounds. There is a difference between an exception and an error. An exception is a problem that occurs when a program is running. Often the problem is caused by circumstances outside the control

of the program, such as bad user input. When the Java run-time system catches an exception it halts the program and prints an error message. When an exception occurs, the Java virtual machine creates an object of class Exception which holds information about the problem.

An 'error' is also a problem that occurs when a program is running. But an error is too severe for a program to handle. The program must stop running. An error is represented by an object of class Error. Programs can be written to recover from Exceptions, but programs can't be written to recover from Errors.

Class Exception and class Error both descend from class Throwable. A Java method can "throw" an object of class Throwable.

```
public class Exceptions
{
   public static void main(String Args[])throws
                                    ArrayIndexOutOfBoundsException
   {
      int[] array = new int[3];
      try {
            for(int i=0;i<4;++i)
            {
               array[i] = i;
            }
         }

      catch(ArrayIndexOutOfBoundsException e)
         {
            System.out.println("Oops, we went too far!");
         }
   }
}
```

Diagram 97. Catching an Array Exception

Have a look at the program shown above. The part of the 'main' header 'throws ArrayIndexOutOfBoundsException' is optional and the program will work perfectly without it. However, inserting it will declare that an exception can be 'thrown'. This is to say that an exception can occur.

The 'try' part indicates that the code following the 'try' reserved word may cause an exception. In case the exception is caused, i.e. it is thrown, then the code following the 'catch' reserved word will be executed. The 'catch' part in the program will be executed only in the case the exception is an array-out-of-bounds exception.

Now look at the following program.

```
public class exceptions2
{
      public static void main(String Args[])throws
                                    ArrayIndexOutOfBoundsException
      {
            int[] array = new int[3];
            try
               {
                  for(int i=0;i<4;++i)
                  {
                        array[i] = i;
                  }
                  System.out.println(array);
               }

            catch(ArrayIndexOutOfBoundsException e)
               {
                  System.out.println("Oops, we went too far!");
               }

            finally
               {
                  System.out.println("OK");
               }
      }
}
```

Diagram 98. Including the 'finally' block after the 'catch' block

In addition to the 'try' and 'catch' blocks we find in this program the 'finally' block. This is a block of code which should be free of exceptions in all circumstances as it will be run always, whether an exception is thrown or not.

When more than one type of exception can occur the program can have a sequence of 'catch' blocks as shown in the following program.

```
public class Exceptions3
{
   public static void main(String Args[])
   {
      int[] array = new int[3];
      try
      {
         for(int i=0;i<3;++i)
         {
            array[i] = i;
         }
         array[0] = 2/0;
```

```
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
           System.out.println("Oops, we went too far!");
        }

        catch(ArithmeticException e)
        {
           System.out.println("Cannot Divide by Zero!");
        }

        catch(Exception e)
        {
           System.out.println("An Unknown Error has Occurred");
        }

        finally
        {
           System.out.println("OK");
        }
    }
}
```

Diagram 99. Program with a cascade of 'catch' statements

The three 'catch' blocks work in this way: if an array-index-out-of-bounds exception occurs then the string "Oops, we went too far!" will be displayed. If an arithmetic exception occurs (as is the case in the above program where array[0] is assigned a number divided by zero) then the string "Cannot Divide by Zero!" will be displayed. For any other exception the string "An Unknown Error has Occurred" will be displayed. The string "OK" will be displayed in any case.

8.6 The Arraylist

Java ArrayList represents an automatic re-sizable array. ArrayList supports both Iterator and ListIterator for iteration but it's recommended to use ListIterator as it allows the programmer to traverse the list in either direction. The following program makes use of the ArrayList.

```
import java.util.ArrayList;

public class AboutArraylists
{
   static void ViewArraylist (ArrayList al)
   {
      int l = al.size();
```

```
      for (int i = 0; i<l; i++)
         System.out.print (al.get(i) + " ");
      System.out.println();
   }

  static void main (String[ ] args)
  {
      ArrayList ListTest = new ArrayList();

      ListTest.add ("James");
      ListTest.add ("Anne");
      ListTest.add ("Isabel");
      ListTest.add (7);

      ViewArraylist (ListTest);

      ListTest.remove(2);
      ViewArraylist (ListTest);

      ListTest.add ("Paul");
      ViewArraylist (ListTest);

      ListTest.add (1, "Paul");
      ViewArraylist (ListTest);

      ListTest.remove ("Anne");
      ViewArraylist (ListTest);

      ListTest.remove ("Paul");
      ViewArraylist (ListTest);

      int x;
      x = (int) ListTest.get(1);
      System.out.println (x+5);

  }
}
```

Diagram 100. Writing, Reading, Deleting and Making Calculations involving an Array List

The output from the above program is shown on the right.

```
James Anne Isabel 7
James Anne 7
James Anne 7 Paul
James Paul Anne 7 Paul
James Paul 7 Paul
James 7 Paul
12
```

Diagram 101. Output

8.7 Serialization in Java

When you serialize an object in Java, you convert the data into byte streams so that later they can be converted back into the original data. If this sounds confusing, think of serialization in the following terms. You're working on a document and then save it to your hard drive. You are, in manner of speaking, serializing the data so you can retrieve that copy later on. Serialization makes the transfer of data on networks much easier and more efficient.

Let us define a class called Student (shown below)

```
public class Student implements java.io.Serializable
{
   public int IdNo;
   public String Name;
   public String Address;

   int[] Marks = new int[3];
}
```

Diagram 102. Writing, Reading, Deleting and Making Calculations involving

Now let us define a class that serializes an instant of the class Student. This is shown below.

```
import java.io.*;

public class SerializeDemo
{
   public static void main(String [] args)
  {
     Student stud = new Student();
     stud.IdNo = 2586;
     stud.Name = "Sandy Bright";
     stud.Address = "55, Top Hill Road, Zurrieq";
     stud.Marks[0] = 87;
     stud.Marks[1] = 57;
     stud.Marks[2] = 66;

     try
     {
       FileOutputStream fileOut =
         new FileOutputStream("student.ser");
       ObjectOutputStream out =
         new ObjectOutputStream(fileOut);
       out.writeObject(stud);
       out.close();
       fileOut.close();
```

```
    }catch(IOException i)
    {
       i.printStackTrace();
    }
  }
}
```

Diagram 103. A Class that Serializes an Object

Next, we define a class that makes use of the serialized instance. This is given below.

```
import java.io.*;

public class DeserializeDemo
{
  public static void main(String [] args)
  {
    Student stu = null;
    try
    {
      FileInputStream fileIn =
        new FileInputStream("student.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      stu = (Student) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i)
    {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c)
    {
      System.out.println("Student class not found");
      c.printStackTrace();
      return;
    }

    System.out.println ("Deserialized Student...");
    System.out.println ("Id No: " + stu.IdNo);
    System.out.println ("Name: " + stu.Name);
    System.out.println ("Address: " + stu.Address);
    System.out.println ("Mark 1: " + stu.Marks[0]);
    System.out.println ("Mark 2: " + stu.Marks[1]);
    System.out.println ("Mark 3: " + stu.Marks[2]);
  }
}
```

Diagram 104. A Class that Reads a Serialized Object and Displays it

We will now give another example that makes use of an ArrayList.

```
import java.util.ArrayList;

public class Employee implements java.io.Serializable
{
   public int IdNo;
   public String Name;
   public String Address;

   ArrayList ExtraHours = new ArrayList();
}
```

Diagram 105. This Serializable Class includes an Array List

The following class is used to save the serialized data structure.

```
import java.io.*;

public class SerializeDemoEmp
{
   public static void main(String [] args)
  {
     Employee emp = new Employee();
     emp.IdNo = 5406;
     emp.Name = "George Borg";
     emp.Address = "55, Top Hill Road, Zurrieq";

     emp.ExtraHours.add (7);
     emp.ExtraHours.add (4);

     try
     {
       FileOutputStream fileOut =
         new FileOutputStream("employee.ser");
       ObjectOutputStream out =
         new ObjectOutputStream(fileOut);
       out.writeObject(emp);
       out.close();
       fileOut.close();
     }catch(IOException i)
     {
        i.printStackTrace();
     }
  }
}
```

Diagram 106. A Program that Saves a Serialized Object that includes an Array List

Finally this last part makes use of the file that contains the serialized data.

```
import java.io.*;
import java.util.ArrayList;

public class DeserializeDemoEmp
{
  public static void main(String [] args)
  {
    Employee employ = null;
    try
    {
      FileInputStream fileIn =
        new FileInputStream("employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      employ = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i)
    {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c)
    {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }

    System.out.println ("Deserialized Employee...");
    System.out.println ("Id No: " + employ.IdNo);
    System.out.println ("Name: " + employ.Name);
    System.out.println ("Address: " + employ.Address);

    int s = employ.ExtraHours.size();
    int tot = 0;
    int temp;
    for (int i = 0; i<s; i++)
     {
       System.out.println (employ.ExtraHours.get(i));

       temp = (int) employ.ExtraHours.get(i);
       tot = tot + temp;
     }
    System.out.println ("Extra hours = " + tot);
  }
}
```

Diagram 107. Reading a Serialized Object that includes an Array List