LESSON

6

EXTENSION TO CLASSES AND METHODS

CONTENTS

6.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define method overloading
- Explain the passing objects to methods and passing arguments
- Discuss the concept of returning objects, recursion, and nested classes
- Identify command line execution

6.1 INTRODUCTION

The words "class" and "object" are used so much in Object-oriented programming that it is easy to get the terms mixed up. Generally speaking, a class is an abstract representation of something, whereas an object is a usable example of the thing the class represents. The one exception to this rule is shared class members, which are usable in both instances of a class and object variables declared as the type of the class.

Methods represent actions that an object can perform. For example, a "Car" object could have "Start Engine," "Drive," and "Stop" methods. You define methods by adding procedures, either Sub routines or functions, to your class.

6.2 METHOD OVERLOADING

In Java, it is possible to create methods that have the same name, provided they have different signatures. A signature is formed from its name, together with the number and types of its arguments. The method in return type is not considered a part of the method signature. Method overloading is used when objects are required to perform a similar task but by using different input parameters. When we call a method in an object, Java matches up the method name first, and then the number and type of parameters, to decide which method to execute. This process is known as polymorphism.

```
public class Movie
{       float price;
        public void setprice( )
        {
                price = 3.50;
        }
        public void setprice (float newprice)
        {
                price = newprice;
        }
}
        Movie mov1 = new Movie( );
                        mov1.setprice( );
                        mov1.setprice (3.25);
```

As you can see setprice( ) is overloaded two times. The first version takes no argument and sets the price to 3.50 and second constructor takes single argument which sets the price to 3.25.

When an overloaded method is called, Java looks for a match between the argument used to call the method and method's parameters. However, this match need not be exact. In some cases Java's automatic type conversion takes the important role in overloaded resolution.

Example:

```
public class Movie
{
        float price;
        public void disp( )
        {
          System.out.println ("No parameter");
     }
     public void disp (int x, int y)
     {
          System.out.println ("x =" + x + "y =" + y);
     }
     public void disp (double x)
     {
          System.out.println ("x =" + x);
     }
}
class overload
{
     public static void main (String args[ ])
     {
          Overload ob = new overload( );
          int i = 100;
          ob.disp( ) // invokes no argument constructor
          ob.disp (10, 20) // invokes disp (int, int)
          ob.disp (i) // invokes disp (double)
          ob.disp (100.50) // invokes disp (double)
     }
}
```

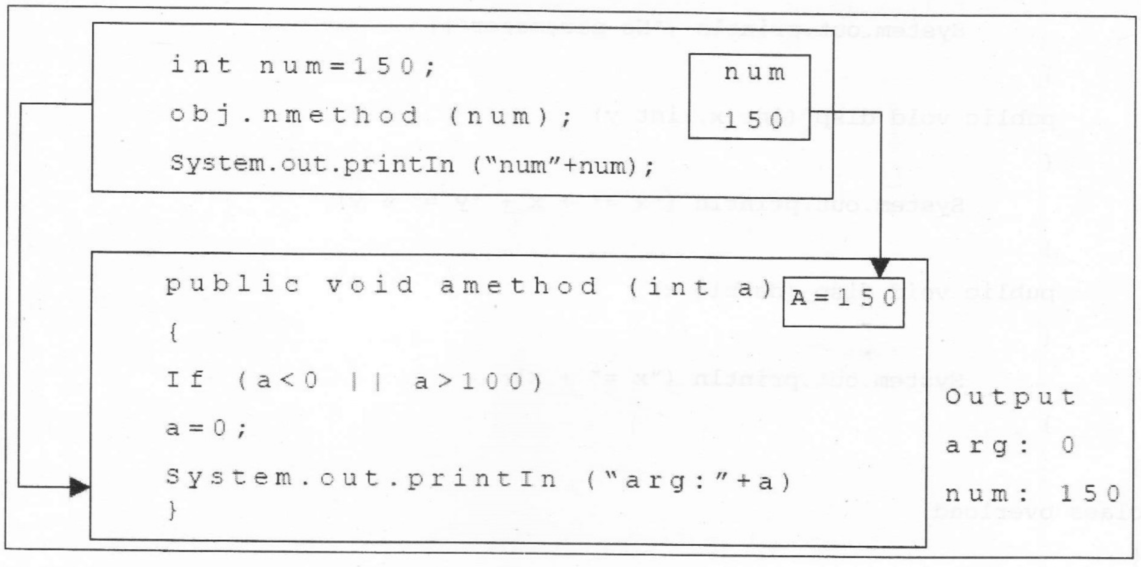
Class Movie does not define disp (int). When disp (int) is called from main( ), when no matching method is found, Java automatically converts an integer into a double. Java uses its automatic type conversion only if no exact match is found.

Java's standard class library has abs( ) method. This method returns absolute value of all type of numeric types e.g. int, double, float. This method is overloaded in Math class. Whereas in 'C' language there are three methods - abs( ), fabs( ), labs( ) - to return absolute of int, float and long values respectively, the java.awt.Graphics class has six different drawImage methods.

As far as Java is concerned, these are entirely different methods. The duplicate names are really just a convenience for the programmer.

6.3 PASSING OBJECTS TO METHODS AND PASSING ARGUMENTS

When a primitive value is passed into a method, a copy of its value is passed into the method argument. If the method changes the value of the argument in any way, only the local argument is affected. When the method terminates, this local argument is discarded.



```
int num=150;
obj.nmethod (num);
System.out.printIn ("num"+num);
```

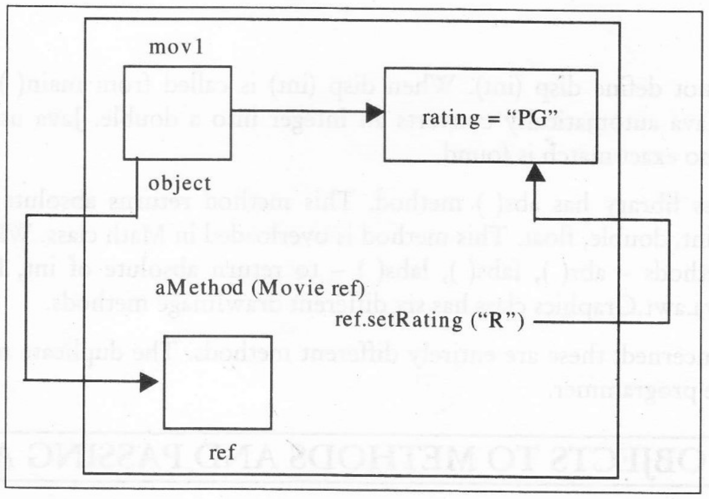
```
public void amethod (int a)
{
If (a<0 || a>100)
a=0;
System.out.printIn ("arg:"+a)
}
```

```
Output
arg: 0
num: 150
```

This way of method calling is called call-by-value.

When an object reference is passed to a method the argument refers to the original object :

```
Movie mov1 = new movie ("Gone with the Wind");
        mov1.setRating ("PG");
        anobj.amethod (mov1);
              public void aMethod (Movie ref)
              {
                    ref.setRating ("R");
              }
```



When mov1 is passed into aMethod( ), the method receives a reference to the original Movie object. When aMethod( ) terminates, the original Movie object referred to by mov1 has a rating of 'R' not 'PG'. Inside the subroutine, 'ref' is used to access actual argument specified in call. This means that the changes made to the parameter will affect the argument used to call the subroutine. This way of method calling is known as call-by-reference.

6.4 RETURNING OBJECTS

A method can return primitive datatypes or an object.

Example:

```
class test
{
        int a;
        test (int i)
        {
                a = i;
        }
        test incrbyten( )
        {
                test temp = new test (a+10);
                return temp;
        }
}
class Retob
{
        public static void main (String args[ ])
        {
                test ob1 = new test (2);
                test ob2;
                ob2 = ob1.incrbyten( );
                System.out.println ("ob1.a:" + ob1.a);
                System.out.prinln ("ob2.a:" + ob2.a);
        }
}
```

Output :

```
ob1.a:2
ob2.a:12
```

Each time when incrbyten() is invoked, a new object is created, and a reference to it is returned to the calling routine.

One of the most important uses of object parameters involves the constructor. You may want to construct a new object so that it is initially the same as the existing object.

```
class rectangle
{
        int width;
        int height;
        rectangle (int w, int h)
    {
              width = w;
              height = h;
        }
        rectangle (rectangle r)
        {
              width = r.width;
              height = r.height;
        }
}
class ABC
{
        public static void main (String args[ ])
        {
              Rectangle rect1 = new Rectangle (10, 15);
              Rectangle rect2 = new Rectangle (rect1);
        }
}
```

6.5 RECURSION

Recursion is a process by which a function calls itself repeatedly, until some specific condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of the previous result.

For e.g. to calculate the factorial of a positive integer quantity, the following expression would be given:

n! = 1x2x3 - - - - xn where n is the specified integer. It can also be written as

n! = n x (n-1) !

This is a recursive statement of the problem in which the desired action is written in terms of the previous result.

Now 1! = 1 by definition. This provides a stopping condition for recursion.

When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a stack until the condition that terminates the recursion is encountered.

The function calls are then executed in reverse order as they are "popped" off the stack.

```
class f
{
        int num;
        int fact (int n)
{
                if (n = = 1)
                return (1);
                num = n * fact (n-1);
                return num;
            }
    }
    class ABC
    {
            public static void main ( )
            {
                int Result;
                f f1 = new f( );
                Result = f1.fact (5);
                System.out.println ("Factorial" + Result);
            }
    }
```

When fact is called with an argument of 1, the function returns 1; otherwise it returns product of fact (n-1) ′ n. To evaluate this expression, fact( ) is called with n-1. This process repeats until n equals 1 and the calls to the method begin returning. When a method calls itself, new local variables and parameters are allocated storage on the stack and the method code is executed with these new variables from the start. A recursive call does not create a new copy of the method, only the arguments are new. As each recursive call returns, the local variables and parameters are removed from the stack and the execution resumes at the point of the call inside the method.

The main advantage of recursive methods is that they can be used to create clearer and simpler versions of algorithms than can their iterative algorithms. Recursive programs may execute little slowly because of added overhead of the additional function calls. Recursive programs may cause Stack overrun. You must have an if statement to force the method to return to calling starting function.

6.6 NESTED AND INNER CLASSES

Inner classes are classes which are defined within other classes. These inner classes have the same scope and access as other variables and methods defined within the same class i.e., the scope of a nested class is bound by the scope of its enclosing class.

```
public class PrivateAirport extends Airport
{
        String owner;
        private Airport (String str){owner = str;}
        class PrivateFlight extends Flight
        {
            string flightowner;
```

```
                private Flight( ) {flightowner = owner;}
        }
}
Flight getFlight( )
{
        return new PrivateFlight( );
}
        }


class Airport {                     Class Flight {
        String Airportname;         int altitude;
                :                         int speed;
                :                               :
                }
                      :
}
```

There are four types of inner classes:

- Static Inner Class
- Member Inner Class
- Local Inner Class
- Anonymous Inner Class

Static inner classes only have access to the static component of the outer class with no access to instance components directly. The static inner class can access instance components through its object.

Member inner classes are non static classes. These can access all the components of the outer class. An instance of the outer class needs to exist to be able to create a new instance of the member inner class.

```
public class outer
{
       public class inner
     {
        public void innerMethod( )
        {
        // 'this' refers to an instance of
        // Outer.inner
        // 'Outer.this' refer to an instance of outer
        // 'super' super class of inner (object)
        // 'outer.super' superclass of outer
        }
       }
}
```

Member inner classes are compiled to separate class files. In this case, the outer/inner.class is created.

Local inner classes are declared within the code block of a method. They have access to final variables only. Local inner classes cannot be declared public, protected, private or static and cannot have static members.

Anonymous inner classes are local inner classes with no class name. They are commonly used to implement user interface adapter to perform event handling. These classes can access local variables in the method and parameters passed to the method only if they are defined final.

6.7 STRING HANDLING

A string is a sequence of characters. Java provides String class to manipulate strings, passed as arguments to methods.

```
System.out.println ("Hello world");
String str = "Action";
String str = new String ("Anil");
```

Java strings are more reliable than 'C' strings because there is lack of bound-checking in C. A Java string is not a character array and is not Null terminated.

To get the length of string, use length method of the string class:

```
int m = str.length( );
```

Strings can be concatenated with '+' operator:

```
String FullName = name1 + name2;
System.out.println (Firstname + "Kumar")
```

StringBuffer Class

String creates strings of fixed length while StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string or append another string to the end.

String and StringBuffer classes are discussed in a later lesson in detail.

6.8 COMMAND LINE EXECUTION

Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution. We can write Java programs that can receive and use the arguments provided in the command line:

```
public void static main (String args[ ])
```

where args is a string object which contains array of strings. Any argument passed through command line are passed to array args [ ].

Example:

Java Test One Two Three

One	→	args [0]
Two	→	args [1]
Three	→	args [2]

Individual elements can be accessed using an index.

```
class Comparetext
{
        public static void main (String args[ ])
        {
                int count, i = 0;
                String string;
                count = args.length;
        System.out.println ("No of arguments = "+ count);
                while (i < count)
                {
                        string = args [i];
        i = i+1;
System.out.println (i + "th argument is "+string);
                }
        }
}
```

The above program will display all the parameters given from the command line.

6.8.1 finalize ( ) Method

JVM is supposed to run an object's finalize method after it has decided that an object has no references and thus, should be recovered, but before actually reclaiming the memory.

Java manages memory automatically– so an object does not need to explicitly free up any secondary memory it might have allocated. To allow an object to clean up resources other than memory, such as open files, Java allows a class to provide a finalize( ) method. Unfortunately, there is no guarantee as to when this will happen. It is implementation specific.

If an object holds another resource such as a file, the object should reclaim it. Finalize method is called automatically before garbage collection.

Check Your Progress

Fill in the blanks:

1. .... are parameters that are supplied to the application program at the time of invoking it for execution.
2. Static inner classes only have access to the .... component of the outer class with no access to instance components directly.
3. A string is a sequence of characters. Java provides String class to .... strings, passed as arguments to methods.
4. .... classes are declared within the code block of a method.

6.9 LET US SUM UP

In Java, it is possible to create methods that have the same name, provided they have different signatures. A signature is formed from its name, together with the number and types of its arguments. When a primitive value is passed into a method, a copy of its value is passed into the method argument. If the method changes the value of the argument in any way, only the local argument is affected. When the method terminates, this local argument is discarded. Recursion is a process by which a function calls itself repeatedly, until some specific condition has been satisfied. The process is used for repetitive computations in which each action is stated. Inner classes are classes which are defined within other classes. These inner classes have the same scope and access as other variables and methods defined within the same class i.e., the scope of a nested class is bound by the scope of its enclosing class. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

6.10 KEYWORDS

JVM: Java Virtual Machine

Command Line Arguments: Parameters that are supplied to the application program at the time of invoking it for execution.

Recursion: Process by which a function calls itself repeatedly, until some specific condition has been satisfied.

Inner Classes: Classes which are defined within other classes.

6.11 QUESTIONS FOR DISCUSSION

1. What is Method overloading? Explain with an example.
2. What is the difference between nested and inner classes?
3. Explain recursion and returning of objects.
4. Define string handling and command line execution.

Check Your Progress: Model Answers

1. Command line arguments
2. Static
3. Manipulate
4. Local inner

6.12 SUGGESTED READINGS

E. Balaguruswami, *Programming with Java*, Tata McGraw-Hill.

Davis, Stephen R., *Learn Java Now*, Microsoft Press.

Naughton, Patrick, *The Java Hand Book*, OSborne McGraw- Hill.

Herbert Schildt, *The Complete Reference Java 2*, Tata McGraw-Hill.

LESSON 7 INHERITANCE

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the basic concepts of inheritance
- Define multiple hierarchy
- Discuss method overriding
- Identify abstract classes

7.1 INTRODUCTION

The concept of overloading was also discussed in the previous lesson. In this lesson you will be able to extend the concept of overloading in inheritance. In the previous lesson you have studied the definition of public and private in detail. In this lesson you will be able to appreciate the meaning and usage of protected modifier in detail. You will also be able to use super keyword to refer to the constructor, method or variable of the superclass.

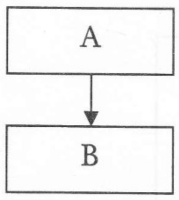
7.2 INHERITANCE BASICS

Reusability is one of the important aspects of the OOP paradigm. Java supports the concept of reusing the code that already exists. Inheritance allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes each adding those things that are unique to it. A class that is inherited is called superclass. The class that does the inheriting is called a subclass.

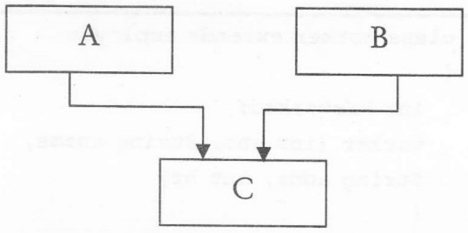
7.2.1 What is Inheritance?

Inheritance defines a relationship between classes where one class shares the data structure and behaviours of another class. It enables software reuse by allowing you to create a new class based on the properties of an existing class. As a result, the developer is able to achieve greater productivity in a short time. The mechanism of deriving a new class from an old one is called inheritance. Inheritance may take different forms:

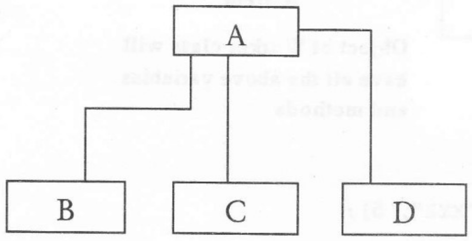
1. Single Inheritance



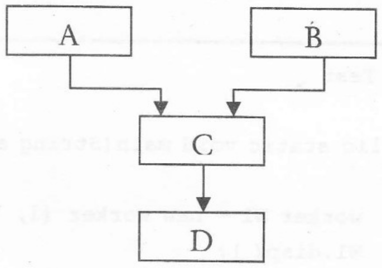
2. Multiple Inheritance (interfaces)



3. Hierarchical Inheritance



4. Multilevel Inheritance



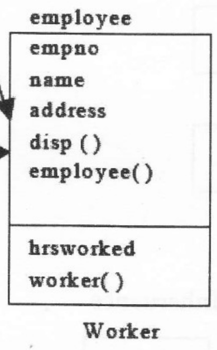
7.2.2 Defining Subclass

```
class subclassname extends superclassname
      {
      variable declarations;
      method declarations;
      }
```

The subclass now contains its own variables and methods as well as those of the superclass.

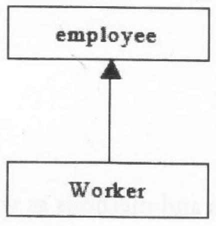
```
class employee
{
   int empno;
   String name;
   String address;
   employee (int eno, String ename, String addr)
   {
       empno = eno;
       name = ename;
       address = addr;
   }
   disp( )
   {
       System.out.println ("eno"   +   empno   +   "name"
       + name);
   }
}
```

```
class worker extends employee
{
   int hrsworked;
   worker (int eno, String ename,
   String addr, int hr)
   {
   super (eno, ename, addr);
   hrsworked = hr;
   }
}
```



Object of Worker class will have all the above variables and methods

```
class Test
{
   public static void main(String args[ ])
   {
       worker W1 = new worker (1, "ABC", "XYZ", 5);
       W1.disp( );
   }
}
```



The employee class defines the attributes and methods that are relevant for all kinds of worker's details. Attributes such as empno, name and address. Methods such as disp( ) to display employee's empno and name. You might need to represent a specific type of employee in a particular way. You can use inheritance to define a separate subclass of employee for each different type of employee. For example you might define manager, engineer, scientist etc.

Each subclass automatically inherits the attributes and methods of employee, but can provide additional attributes and methods. Attributes such as number of projects, type of project, duration of project, status can be stored in scientist class. Methods such as reviewing the project progress, setting the current status can be added in the scientist subclass. Subclass can also override a method from the superclass if they want to provide more specialized behaviour for the method. If the subclass method has the same name, parameter list and return type as a superclass method, then you can say that the subclass method overrides the superclass method.

7.2.3 Member Access and Inheritance

A subclass includes all of the members of its superclass; it cannot access those members of the superclass that have been declared as private. When you create an object, you can call any of its public methods plus any public methods declared in its super class.

```
class A
{
  int i;
  private int j ;
  void getdata (int x, iny y)
  {
      i = x;
      j = y;
  }
}
class B extends A
{
  int total;
  void sum( )
  {
      total = i + j;
  }
}
class AB
{
  public static void main (String args[ ])
  {
      B SubOb = new B( );
      SubOb.getdata (10, 12);
      SubOb.sum( );
 System.out.println ("Total" + SubOb.total);
  }
}
```

The program will not compile because the reference to j inside the sum( ) method of B causes an access violation. Subclass has no access to j. A class member that has been declared as private will remain private to its class; it is not accessible by any code outside its class, including subclasses.

7.2.4 Superclass Variable and Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class Box
{
  double w;
  double h;
  double d;
  Box (Box 0b)
  {
      w = 0b.w;
      h = 0b h;
      d = 0b.d;
  }
  Box (double x, double y, double z)
  {
      w = x;
      h = y;
      d = z;
  }
  double volume( )
  {
      return w * h * d;
  }
}
class BoxWeight extends Box
{
  double weight;
  BoxWeight (double x, double y, double z, double r)
  {
      w = x;
      h = y;
      d = z;
      weight = r;
  }
}
class RefDemo
{
  public static void main (String args[ ])
{
BoxWeight weightbox = new BoxWeight (3, 5, 7, 8.37);
```

```
            Box plainbox = new Box( );
            double vol;
            vol = weightbox.volume( );
            System.out.println ("Volume =" + vol);
            plainbox = weightbox;
            vol = plainbox. volume( );
      System.out.println ("Volume of plainBox is" + vol);
            System.out.println ("Weight of plainBox is" +
      plainbox.weight);
        }
      }
```

Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weight box object. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by superclass. Plainbox can't access weight, even when it refers to a BoxWeight object, because superclass has no knowledge of what a subclass adds to.

7.2.5 Using Super to Call Superclass Constructors

Super has two general forms. The first calls the superclass constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass

- Super (parameter-list);
- Super membername;

A subclass constructor is used to construct the instance variables of both the subclass and the super class. The super keyword invokes the constructor method of the super class. Super may only be used within a subclass constructor method as the first statement. The parameter in the super call must match the order/type of the instance variable declared in the superclass.

```
      class Box
      {
        double w;
        double h;
        double d;
        Box (double x, double y, double z)
        {
            w = x;
            h = y;
            d = z;
        }
     }
      class BoxWeight extends Box
      {
        double weight;
        BoxWeight (double x, double y, double z, double m)
```

```
    {
        super (x, y, z);
        weight = m;
    }
}
class demoSuper
{
  public static void main (String args[ ])
  {
        BoxWeight mybox1 = new BoxWeight (10, 20, 15, 34.3);
        System.out.println ("Weight of mybox1 is ="+mybox1.weight);
  }
}
```

When a subclass calls super( ), it is calling the constructor of its immediate superclass. Call to super( ) avoids duplicating the code in subclass but it implies that subclass must be granted access to these members. However, in some cases we would like to keep the variable private to itself. Access to these private variables of the superclass can be achieved by the keyword super.

Another use of Super

The second form of super always refers to the superclass of the subclass in which it is used. It is most applicable to the situation in which member names of a subclass hide members by the same name in the superclass.

Example:

```
class A
{
  int i;
}
class B extends A
{
  int i;

 B (int a, int b)
{
        super.i = a;        // i in A
        i = b;              // i in B
  }
}
class demosuper
{
  public static void main (String args[ ])
  {
        B b1 = new B(1, 2);
  }
}
```

In the above example, class A and class B both have instance variable called i. Class B will also inherit superclass version of variable i. To refer to superclass version of variable i, use super.i.

7.3 MULTILEVEL HIERARCHY

In multilevel hierarchy, each subclass inherits all of the traits found in all of its superclasses. It is perfectly acceptable to use a subclass as superclass of another.

```
class Box
{
  private double width;
  private double height;
  private double depth;
  Box (double w, double h, double d)
  {
      width = w;
      height = h;
      depth = d;
  }
double volume( )
{
  return (width * height * depth);
}
}
class BoxWeight extends Box
{
  double weight;
  BoxWeight (double w, double h, double d, double m)
  {
      super (w, h, d);
      weight = m;
  }
}
class shipment extends BoxWeight
{
  double cost;
  Shipment (double w, double h, double d, double m, double c)
{
  super (w, h, d, m);
  cost = c;
}
class DemoShipment
{
  public static void main (String args[ ])
  {
```

```
            Shipment Shipment1 =
        new Shipment (10, 20, 15, 10, 3.41);
         double vol;
            vol = shipment1.volume( );
            System.out.prinln ("Volume =" + vol);
        System.out.println ("Weight" +Shipment1.weight);
            System.out.println ("ShippingCost" +, Shipment1.cost);
         }
       }
```

Because of inheritance, shipment can make use of previously defined classes of Box, BoxWeight, adding only the extra information it needs for its own specific application. Hence, inheritance allows reuse of code.

In the above example, subclass BoxWeight is used as a superclass to create the subclass called Shipment. Shipment inherits all of the traits of BoxWeight and Box and adds a variable called cost.

7.4 METHOD OVERRIDING

Inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass because subclass inherits all of the methods of its superclass. However, a subclass can modify the behaviour of a method in a superclass by overriding it. To override a superclass method, the subclass defines a method with exactly the same signature and return type as a method somewhere above it in the hierarchy. Then, when the method is called, the method defined in the subclass is invoked and executed instead of the one in super class. This is known as overriding. In the previous example we change worker class as follows:

```
class worker extends employee
{
  int hrsworked;
  worker (int eno, String ename, String addr, int hr)
  {
      super (eno, ename, addr);
      hrsworked = hr;
  }
disp( )
{
  System.out.println ("eno" + eno + "name" + name);
  System.out.println ("ename" + ename + "hrsWorked" + hrsworked);
}
```

The method in the subclass effectively hides the superclass method. Now, when disp()method is called for, the worker object method defined in the worker object will be invoked rather than the one defined in the employee class. Further, in the above example we do not need to repeat the code which already exists in the parent class. We can modify disp() method of worker class as follows:

```
disp( )
{
super.disp( );
system.out.println ("HrsWorked" + hrsworked);
}
```

Note that the super notation can be used to execute a method in the immediate superclass.

There are certain restrictions on overriding methods:

- ***Access Modifier:*** An overriding method cannot be made more private than the method in the parent class.
- ***Return Type:*** The return type must be the same as in the parent class method.
- ***Exception Thrown:*** Any exception declared must be of the same class as that thrown by the parent class or of a subclass of that exception.

Methods that are declared final cannot be overridden. Methods that are declared private cannot be seen outside the class and therefore, cannot be overridden. However, because the private method name cannot be seen, you can reuse the method name in the subclass, but then this is not the same as overriding.

Difference between Overloading and Overriding

The terms overloading and overriding are applied to situations in which you have multiple Java methods with the same name. Within a particular class, you can have more than one method with the same name as long as they differ in number, type and order of the input parameters. This is described as overloading. For Java, these are two different methods. The duplicate names are just for the convenience of the programmer. Compiler will not allow methods with same signature but with different return type, even if one method is declared in parent class and the other in the subclass.

If a subclass method has the same name, parameter list and return type as superclass method, you say that the subclass method overrides the superclass method.

Overloaded methods are resolved at compile time, based on the arguments you supply. Overridden methods are resolved at run time.

Now consider the following code:

```
worker w1 = new worker ( )
employee e = w1;
w1.disp( ) // which method is called
```

Casting w1 to employee reference does not change the object type. Because the JVM resolved method calls at runtime using the actual object, the worker version disp() method is executed.

7.5 ABSTRACT CLASSES METHOD

An abstract class is simply a class which cannot be instantiated. For example , employeeClass does not provide sufficient details to provide anything meaningful about the employee. It must either be a worker, manager, staff or employee on deputation. Only its nonabstract superclasses may be instantiated. If you try to instantiate abstract class, compiler flags an error.

An abstract method defined within an abstract class must be implemented by all of its subclasses. This technique allows the class designer to decide exactly what behaviours a subclass must be able to perform. Java provides abstract keyword, which indicates that a class or a method is abstract.

```
abstract class employee
        {
          int empno;
          String name;
          abstract void getdata( );
          abstract void displaydata( );
        }
```

When you declare abstract method you just provide the signature for the method which comprises of its name, its argument list and its return type. You do not provide body for it.

Any class with abstract method must also be declared as abstract. This is done so that all classes extending the abstract class and overriding the abstract method are compatible.

Abstract classes can contain methods that are not declared as abstract. Those methods can be overridden by the subclasses, but it is not mandatory.

In the above class employee, we create a subclass called worker and we do not override the getdata( ) method which is declared abstract in the super class. If user calls workerobj.getdata( ), it calls the method available in the super class and that is not the desired method. Hence, the solution could be to declare the method as abstract in the superclass and override it in the subclass.

Check Your Progress

Fill in the blanks:

1. Inheritance defines a .... between classes where one class shares the data structure and behaviours of another class.
2. A subclass includes all of the members of its ....
3. A subclass .... is used to construct the instance variables of both the subclass and the super class.
4. An .... class is simply a class which cannot be instantiated.

7.6 LET US SUM UP

Reusability is one of the important aspects of the OOP paradigm. Java supports the concept of reusing the code that already exists. Inheritance allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. Inheritance defines a relationship between classes where one class shares the data structure and behaviours of another class. It enables software reuse by allowing you to create a new class based on the properties of an existing class. As a result, the developer is able to achieve greater productivity in a short time. A subclass includes all of the members of its superclass; it cannot access those members of the superclass that have been declared as private. Super has two general forms. The first calls the superclass constructor. In multilevel hierarchy, each subclass inherits all of the traits found in all of its superclasses. Inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass because subclass inherits all of the methods of its superclass. An abstract class is simply a class which cannot be instantiated.

7.7 KEYWORDS

Inheritance: A relationship between classes where one class shares the data structure and behaviours of another class.

Abstract Class: A class which cannot be instantiated.

Multilevel Hierarchy: Each subclass inherits all of the traits found in all of its superclasses.

Subclass: It includes all of the members of its superclass; it cannot access those members of the superclass that have been declared as private.

7.8 QUESTIONS FOR DISCUSSION

1. Write a program to explain single and multiple inheritance.
2. What is method overriding? Explain with an example.
3. Explain abstract classes and multilevel hierarchy.

Check Your Progress: Model Answers

1. Relationship
2. Superclass
3. Constructor
4. Abstract

7.9 SUGGESTED READINGS

E.Balaguruswami, *Programming with Java,* Tata McGraw-Hill.

Davis, Stephen R., *Learn Java Now,* Microsoft Press.

Naughton, Patrick, *The Java Hand Book,* OSborne McGraw- Hill.

Sams.net, Java unleased.

Herbert Schildt, *The Complete Reference Java 2*, Tata McGraw-Hill.

LESSON 8

INTERFACES AND PACKAGES

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Define interface
- Understand what a package is
- Explain classpath variable
- Discuss about access protection

8.1 INTRODUCTION

We cannot do multiple inheritance in Java because we cannot create a subclass by extending more than one superclass. This way of multiple inheritance adds to the complexity in the program. Hence, it is not permitted in Java.

To achieve multiple inheritance in Java, we can implement multiple interfaces to create subclasses. At the end of this lesson you should be able to use interfaces. One of the main features of OOP is its ability to reuse the code by extending the classes and implementing interfaces. If we need to reuse the classes from other programs without copying them, we use packages. We will study user defined packages and Java API packages. We will also study visibility and access control of different class members in relation to the packages.

8.2 DEFINING INTERFACE

An interface is like a fully abstract class, except that it cannot have any concrete method or instance variables. It is a collection of abstract method declarations and constants, that is, static final variables. This means that interfaces do not specify any code to implement these methods. Any class that implements an interface must implement all of the methods specified in that interface. A class can implement many interfaces but can extend only one class. The general syntax for defining interface is:

```
interface InterfaceName
{
        variable declaration;
        method declaration;
}
```

Here interface is a keyword and interfaceName is any valid Java variable.

Example:

```
interface Item
{
        static final int code = 1001;
        static final String name = "fan";
        void display ( );
}
```

Variables are declared as constants using static final keywords. Note that the code for display ( ) is not included in the interface. The class that implements this interface must define the code for the method.

8.2.1 Implementation and Application

Interfaces are used as "superclasses" whose properties are inherited by classes. It is, therefore, necessary to create a class that inherits the given interface.

```
class classname extends superclass
        implements interface1, interface2, - - - -
        {
        }
```

Class can extend another class while implementing interfaces.

Example:

```
interface Area
{
          final static float pi = 3.14F
          float compute (float x, float y);
}
class Rectangle implements Area
{
          public float compute (float x, float y)
          {
                    return (x * y);
          }
}
class Circle implements Area
{
          public float compute (float x, float y)
          {
                    return (pi * x * x);
          }
}
class InterfaceTest
{
          public static void main (String args[ ])
          {
                    Rectangle rect = new Rectangle ( );
                    Circle cir = new Circle ( );
                    Area area;  // interface object
                    area = rect;
                    System.out.println (area.compute(10, 20));
                    area = cir;
                    System.out.println ("Area of Circle" + area.compute(10,0));
          }
}
```

As you can see, the version of compute( ) that is called is determined by the type of object that area refers to at runtime. Note that if a class that implements an interface does not implement all the methods of the interface, then the class becomes an abstract class and cannot be instantiated. When you implement an interface method, it must be declared as public.

8.2.2 Variables in Interfaces

Interface can be used to declare a set of constants that can be used in different classes. The constant values will be available to any class that implements the interface.

```
class Students
{
          int rollNumber;
          void getNumber (int n)
```

```
        {
               rollNumber = n;
        }
        void putNumber( )
        {
               System.out.println ("RollNo." + rollNumber);
        }
}
class Test extends Students
{
        float part1, part2;
        void getMarks (float m1, float m2)
        {
               part1 = m1;
               part2 = m2;
        }
        void putMarks ( )
        {
               System.out.println ("Marks obtained");
               System.out.println (part1);
               System.out.println (part2);
        }
}
interface sports
{
        float sport wt = 6.0
        void putwt ( );
}
class Results extends Test implements sports
{
        float total;
        public void putwt ( )
        {
               System.out.println ("sports wt = "+ sportwt);
        }
        void display ( )
        {
               total = part1 + part + sportwt;
               putNumber( );
               putMarks( );
               putwt( );
               System.out.println ("Total score"+ total);
        }
}
class ABC
{
        public static void main (String args[ ]))
        {
                Results stud1 = new Results ( );
                stud1.getNumber (1, 2, 3, 4);
                stud1.getMarks (27.5, 33.0);
                stud1.display ( );
                }
        }
```

In the above example, the class "Results" implements "Sports" interface. The calculation of total marks for object of "Results" class includes sports weightage also.

It is noted that all methods specified in an interface are implicitly public and abstract. Any variable specified in an interface is implicitly public, static and final.

8.2.3 Extending Interfaces

Like classes, interfaces can also be extended. The new subinterface will inherit all the members of the super interface in the manner similar to subclasses.

```
interface ItemConstants
{
        int code = 1000;
        String name = "Fan";
}
interface Item extends ItemConstants
{
        void display ( );
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the method declared in the super interfaces. Instead, it is the responsibility of any class that implements the derived interface to define all the methods.

Interfaces can have access modifiers of public or blank, just like classes. An interface can be defined as extending another interface, similar to class hierarchy, but there is no base interface analogous to the Object class.

Another Example:

Sort is a classic example of the use of an interface. Many completely unrelated classes may use a sort. A sort is a well defined process that does not need to be written repeatedly.

The "Sortable" interface in the example specifies the methods required to make the sort work on each type of object that needs to be sorted. Only the class needs to know its object comparison or sorting rules.

Suppose there is a need to sort employee class database on the basis of empno. Employee class will be created by the organization which knows everything about the employee. Classes required to do sorting can be created by a separate developer who knows about sorting algorithm and nothing about employee class. Thus, example may use a sortable interface which declares one method i.e., compare( ). This method must be implemented by any class that wants to use the sort class methods. Sort class is an abstract class that contains sortObject methods to sort an array of objects. sortObject( ) calls compare( ) method on the objects in the array. Sortapp represents any application containing main that needs to sort list of employees (Figure 8.1).

The same method can be used to sort the department objects. (Implement this in Lab)

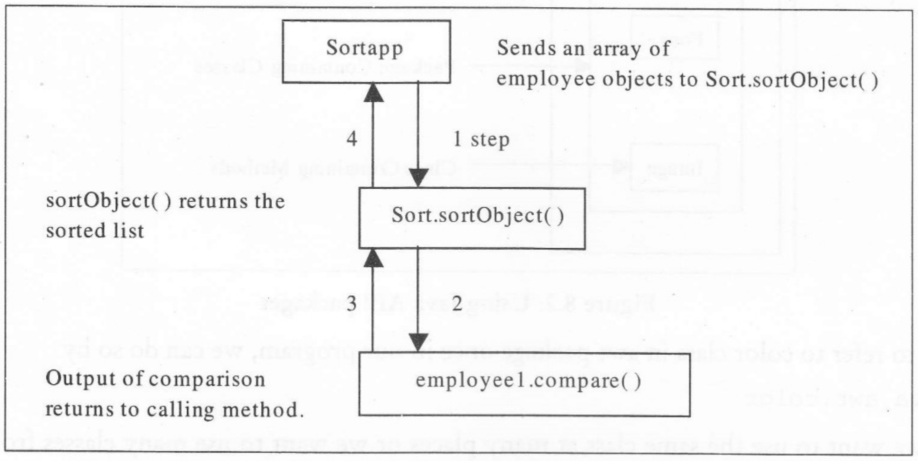

Sortapp
Sends an array of employee objects to Sort.sortObject( )
4
1 step
sortObject( ) returns the sorted list
Sort.sortObject( )
3
2
Output of comparison returns to calling method.
employee1.compare( )

Figure 8.1

8.3 WHAT IS A PACKAGE?

One way of reusing the code is by extending classes and implementing the interfaces. But this is limited to reusing the classes within a program. If we need to use classes from some other programs without physically copying them into the program under consideration, we use packages. Packages act as containers for grouping logically related classes and interfaces.

8.3.1 Benefits of Packages

1. Classes contained in packages of other programs can be easily reused.
2. Two classes in two different packages can have the same name. They can be uniquely identified by packagename.classname.
3. Packages also provide a way for separating "design" from "coding".
4. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

In practical applications, we may have to build our own classes and use existing class libraries for designing user interfaces. Java packages are classified into two types (Figure 8.2):

- Java API package
- User defined packages

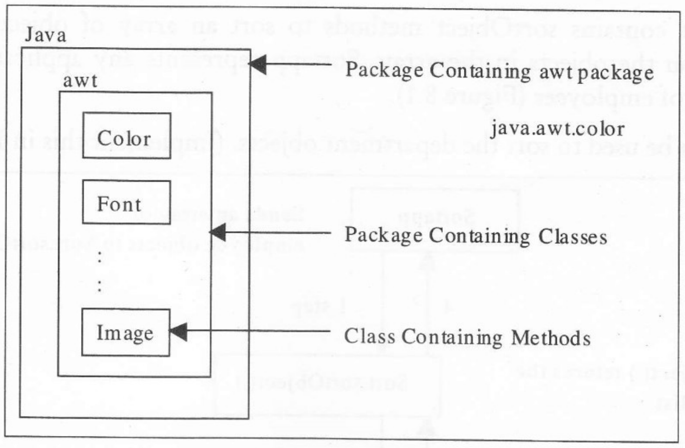


Figure 8.2: Using Java API packages

If we want to refer to color class in awt package once in our program, we can do so by

```
java.awt.color
```

But when we want to use the same class at many places or we want to use many classes from the same package we can use

```
import packagename.classname;
    or
import packagename.*;
```

The second statement imports every class contained in specified package. Packages can be named using the standard Java naming rules.

```
java.awt.Point pts [ ];
```

This statement declares an array of Point type objects using the fully qualified class name.

8.3.2 Creating Packages

To create a package, simply include a package command as the first statement in a Java source file. Any class declared within that file will belong to the specified package. If you omit the package statement, the class names are put into a default package which has no name.

```
package first_package;
public class firstclass
{
}
```

The file should be stored as firstclass.java and should be located in a directory named first_package. The compiled class file should also be stored in the directory that has the same name as the package. Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots - package firstpackage.secondPackage;

Store this package in a subdirectory named

```
first_package\secondPackage.
```

There is no practical limit on the depth of a package hierarchy except that which is imposed by file system. A Java package file can have more than one class definition. In such cases, only one of the classes may be declared as public and that class name with .Java extension is the source file name. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. Most real world packages are spread across many files.

A simple example:

```
Package firstpack;
class Acct
{
      String Name;
      int AcctNo;
      float Balance;
      Acct (String, int A, Float Bal)
      {
            name = n;
            AcctNo = A;
            Balance = Bal;
      }
void disp( )
{
      if (Balance < 0 )
      System.out.println ("Warning");
      System.out.println ("name" + "Rs" + Balance);
}
}
class AccBalance
{
public static void main (String args[ ] )
{
      Acct currentAcct [ ] = new Acct [3];
      currentAcct [0] = new Act ("Anil", 1, 10000);
      current Acct [1] = new Acct ("Sunil", 2, 5000);
      currentAcct [2] = new Acct ("Abhinav", 3, 11000);
      for (int i = 0; i < 3; i++)
      currentAcct [i].disp( );
}
}
```

Store this file with the name AccBalance.java in a directory called firstpack. AccBalance.class should also be stored in the same directory. You can execute this program by giving following at the command line.

```
java firstpack.AccBalance
```

You need to set the appropriate classpath using set classpath command on the command line.

8.3.3 Importing Classes from other Packages

Packages are a good mechanism for compartmentalizing diverse classes from each other. All of the standard classes are stored in some named packages. Classes within packages must be fully qualified with their package name or names with the dot operator. It would be tedious to type in the class name with long dot separated package. By importing a package, a class can be referred to directly, using only its name. In a Java source file, import statements occur immediately following the package statement and before any class definition.

Example:

```
import java.util.*;
class Mydate extends Date { }
OR
class Mydate extends java.util.Date.
```

The general syntax would be

```
import pkg1[.pkg2]. (classname/*);
```

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in package inside of the java package called java.lang.

Given below are two packages.

```
package pack1;
public class Teacher
{ - - - - }
public class student
{- - - - }
package pack2;
public class Courses
{- - - - - }
public class student
{- - - - - -}
import pack1.*;
import pack2.*;
pack1.student stud1;     // OK
pack2.student stud2;     // OK
```

```
      Teacher teacher1;          // OK
      Courses course1;           // OK
```

8.3.4 Subclassing an Imported Class

```
import package2.classB; // Only ClassB from package2 will be imported
class classC extends classB
{
      int n = 20,
      void displayC( )
      {
            System.out.println ("class C");
            System.out.println ("m = "+m);
            System.out.println ("n = "+ n);
      }
}
class ABC
{
      public static void main (String args[ ])
      {
            classC objc = new classC( );
            objc.displayC( );
      }
}
```

Note that the variables m, n have been declared as protected in classB. It would not have been possible if it had been declared as either private or default.

8.3.5 ClassPath Variable

Every package must be mapped to a subdirectory of the same name in the file system. Nested packages will be reflected as a hierarchy of directories in the file system. For e.g. class files of java.awt.image must be stored under the directory java\awt\image.

You can put package directories anywhere in the file system. In addition, you can place whole directory structure of class files within .zip or .jar files. In order to tell the Java compiler where to locate the packages, you must set the CLASSPATH environment variable because the specific location that the Java compiler will consider as the root of any package hierarchy, is controlled by CLASSPATH.

```
      SET CLASSPATH = .;\ABC\myclass.jar
```

When any class reference is encountered, the directory list is searched from start to end. The JVM also checks for .jar or .zip entities in the CLASSPATH entries.

8.3.6 Access Protection

While using packages and inheritance in a program, we should take care of the visibility restrictions imposed by access protection modifiers. The access modifiers are private, public, protected. The least restrictive access modifier is public. Variables and methods declared as public may be seen and used by any other class.

If an access modifier is not specified (default), the variables and methods may be used by any other class within the same package. The next access modifier is protected. These variables can be seen from any subclass of that class. It may also be seen from any class within the package in which it exists.

The most restrictive access modifier is private. A private method or variable may not be accessed by any other class (Table 8.1).

Table 8.1

Access Location \ Access Modifier	Public	Protected	Friendly	Private Protected	Private
Same Class	Y	Y	Y	Y	Y
Sub Class in Same Package	Y	Y	Y	Y	N
Other Classes in Same Package	Y	Y	Y	N	N
Sub Class in Other Package	Y	Y	N	Y	N
NonSub Class in other package	Y	N	N	N	N

Important Packages

a.	java.lang	:	They include classes for primitive types, strings, math functions, threads and exceptions.
b.	java.util	:	Language utility classes such as vectors, hash tables, random numbers, date etc.
c.	java.io	:	They provide facility for the input and output of data.
d.	java.awt	:	Include classes for windows, buttons, lists, menus and so on.
e.	java.net	:	Include classes for communicating with local computers as well as with Internet server.
f.	java.applet	:	Classes for creating applets.
g.	java.sql	:	Classes for sending SQL statements to relational databases.
h.	java.math	:	Classes for extended precision arithmetic.

Check Your Progress

Fill in the blanks:

1. A package is a collection of ....
2. Concept of multiple inheritance is implemented in Java by ....
3. Members of a class specified as private are accessible only to the methods of the class. (True/False.)
4. Any class may be inherited by another class in the same package. (True/False.)

8.4 LET US SUM UP

Java has several levels of hierarchy for code organization, the highest of which is the package. In this lesson we have discussed the following: Packages and their benefits; Different kind of system packages and their requirement us; Naming conventions of package; How to create a package; How to access a package; The ways of using package in a program; How to add more classes to a package.

This lesson has given a brief about how to organize packages and their importance for further usage.

Java does not support multiple inheritance. Since multiple inheritance is an important concept in OOP paradigm, java provides an alternate way of implementing this concept. We have discussed in this lesson: How to design an interface; How to extend one interface by the other; How to inherit an interface; How to implement the concept of multiple inheritance using interfaces.

The concepts discussed in this unit will enable us to build classes using other classes available already.

8.5 KEYWORDS

Package: A Java keyword is used to assign the contents of a file to a package. Package is java's mechanism for grouping classes. Packages simplify reuse and they are very useful for large projects.

Import: A Java keyword is used to import the pre-define system packages and user dfined packages.

Class File: A file containing Machine - independent Java bytecodes. The Java compiler generates .class files for the Interpreter to read.

.Java File: A file containing Java source code.

Interface: A collection of methods and variables that other classes may implement. A class that implements an interface provides implementations for all the methods in the interface.

Multiple Inheritance: When a class simultaneously inherits methods and fields directly from more than one base class.

8.6 QUESTIONS FOR DISCUSSION

1. How do we design a package? How do we add classes or interfaces to a package? How do we execute a class stored in a package?

2. Which keyword can protect a class in a package from accessibility by the classes outside the package?

   (a) Private (b) Protected (c) Final (d) Default (no modifier)

3. We would like to make a member of a class visible in all subclasses regardless of what package they are in. Which keyword would achieve this?

   (a) Private (b) Protected (c) Public (d) Private protected

4. Which of the following keywords are used to control access to a class member?

   (a) Default (b) Abstract (c) Protected (d) Interface public

   (e) Public

5. A package is a collection of

   (a) Classes (b) Interfaces (c) Editing tools (d) Classes and interfaces

6. Study the following code:

(i) package p1;	(ii) public class student	(iii) { }	(iv) class test
(v) { }	(vi) import p1.*	(vii) class Result	(viii) {
(ix) student s1;	(x) Text t1;	(xi) }	

Check Your Progress: Model Answers

1. Private
2. Classes
3. True
4. False

8.7 SUGGESTED READINGS

E.Balaguruswami, *Programming with Java*, Tata McGraw-Hill.

Davis, Stephen R., *Learn Java Now*, Microsoft Press.

Naughton, Patrick, *The Java Hand Book*, OSborne McGraw- Hill.

Sams.net, *Java unleased.*

Herbert Schildt, *The Complete Reference Java 2*, Tata McGraw-Hill.

UNIT IV

LESSON

9

ERRORS AND EXCEPTION HANDLING

CONTENTS

9.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Describe different types of errors
- Define exceptions
- Describe the syntax of exception handling code
- Define java's build in exceptions
- Define Multiple catch statement
- Explain throwing our own exceptions

9.1 INTRODUCTION

As programs become more complicated, making them robust is a much more difficult task. Traditional programming languages like C rely on the heavy use of handlers and cryptic return codes for propagating the abnormal conditions back to the calling methods. Using an exception-handling mechanism, Java provides an elegant way to build programs that are both robust and clear. After completing this lesson you should be able to learn the concept of exception handling. You should be able to write code to catch, handle and throw exceptions and to create your own exceptions. This lesson introduces you to exception handling capabilities built into Java and teaches you to write code to handle exceptions.

9.2 TYPES OF ERRORS

When there is a mistake in a program, it is called an Error. An error can be of two types:

(a) Compile time error

(b) Run-time error

9.2.1 Compile-time Error

A compile time error is shown by the compiler. It may be a syntax error which can be removed by the programmer at compilation time. There may also be an error if you pass wrong-parameters that can also be debugged before the compilation of the program.

Let's consider a sample program to understand the concept of compile time errors:

```
// program with compile time errors
class Error 1
{
   public static void main(string args[ ] )
   {
      system.out.println("Hello Java") //;missing
   }
}
```

When we compile the above program, the java compiler tell us where the errors are in the program. The following message will appear on the screen if we have missed the semicolon:

```
Error 1.java : 7: ';' expected
System.out.println ("Hello java!")
^
1 error
```

Now, we can move to the appropriate line to correct the error. Then we will recompile the program.

Most of the compile-time errors are due to typing mistakes, which are hard to final. The most common errors are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments/initialization
- Bad references to objects
- Use of = in place of = = operator
- And so on.

Other errors may be encountered related to directory path, such as:

```
Javac: command not found
```

9.2.2 Run-time Error

A run-time error occurs due to memory-management problem like same memory allocation is allocated to more than one variable, or there may be a deadlock condition etc. A run-time error is also called as exception. Let's learn its concept in detail.

Most common run-time errors are:

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Casting an instance of a class to one of its subclasses.
- Passing a parameter that is not in a valid range or value for a method.
- Trying to change the state of a thread illegally.
- Using a negative size for array.
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number.
- Accessing a String that is out of bounds of a string.
- And many more.

When such errors are encountered, Java typically generates an error message and aborts the program.

9.3 EXCEPTIONS

An exception is a condition that is caused by a runtime error in the program. In reality, your program may have to deal with many unexpected problems such as missing files, bad user input, dropped network connections etc. If these abnormal conditions are not prevented or at least handled properly, either the program will be aborted abruptly or the incorrect results or status will be carried on, causing more and more abnormal conditions. Java provides an elegant approach to handling these problems with the exception mechanism.

A Java exception is an object that describes an exceptional condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. At some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system or they can be manually generated by your code.

Java exception handling is managed by five keywords:

try, catch, throw, throws, finally.

When an error occurs, its mechanism performs the following tasks:

- Find the problem (hit the exception).
- Inform that an error has occurred (throw the exception).
- Receive the error information (catch the exception).
- Take corrective actions (handle the exception).

When an exception occurs within a Java method, the method creates an exception object and hands it over to the run-time system. This process is called throwing an exception. The exception object contains information about the exception including its type and the state of the program when the error occurred.

When a Java method throws an exception, the Java runtime system searches all the methods in the call stack to find one that can handle this type of exception. This is known as catching the exception. If the runtime system does not find an appropriate exception handler, the whole program terminates.

9.3.1 Types of Exceptions

All exception types are subclasses of the built in class Throwable. The hierarchy of Throwable class is shown in the diagram.

```
java.lang.object
      - java.lang.Throwable
              java.lang.Error
                      Unrecoverable errors
              java.lang.Exception
                      java.lang.Runtime Exception
                              various unchecked exceptions
                      various checked exceptions.
```

Exception class is used for exceptional conditions that user programs should catch. This is also the class to create your own custom exception. The java.lang.Error defines the exception, which are not expected to be caught. e.g. stack overflow. If an error is generated, it normally indicates a problem that will be fatal to the program.

Unchecked (run-time) exceptions are extensions of Runtime Exception class. Exceptions of this type are automatically defined for the program and include things like, divide by zero and invalid array indexing. If a runtime exception occurs and your program does not handle it, the JVM will terminate your program. Checked exceptions are extensions of the exception class - the exceptions must be caught and handled somewhere in your application. Checked exceptions must be handled either by try and catch block or by using throws clause in method declarations.

Uncaught Exceptions

```
class Error2
{
    public static void main (String args[ ])
    {
        int a = 10;
        int b = 5;
        int c = 5;
        int x = a/b-c;
        System.out.println ("x = "+ x);
        int y = a / (b+c);
        System.out.println ("y =" + y);
    }
}
```

When the Java run-time system detects attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Error2 class to stops because once an exception has been thrown, it must be caught and dealt with. In this example, exception is caught by the default handler.

Any exception that is not caught by your program, will be processed by default handler. Java Runtime System generates an error condition which causes the default handler to print the error and to terminate the program.

The errors are printed by Stack Trace. The stack trace will always show the sequence of method invocations that led up to the error. The call stack is useful for debugging because it pinpoints the precise sequence of steps that led to the error.

User Defined Exceptions

You can define your own exception types to handle situations specific to your applications. It is done just by defining a subclass of Exception. All exceptions which you create have the methods defined by Throwable available to them. You can override one or more methods in exception class that you create. For e.g. String toString( ), void print StackTrace( ), String getMessage( ).

9.3.2 Syntax of Exception Handling Code

The programmer can provide for handling of exceptions using the Java keywords try and catch, with or without finally. Try and catch allow you to fix the error. It prevents the program from automatically terminating. A try statement is used to enclose a block of code in which an exception may occur.

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block. The catch block can have one or more statements that are necessary to process the exception. Every try statement should be followed by at least one catch statement. Catch statement is passed a single parameter which is reference to the exception object thrown. If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate. The scope of the variable named in the catch clause is restricted to the catch code block.

A succession of catch clause can be attached to a try statement. If the thrown exception cannot be assigned to the variable in the first catch clause, the JVM looks at the second and so on. If the code block in the try statement executes without throwing an exception, all of the catch clauses are ignored and execution resumes after the last catch.

The order of the catch clause must reflect the exception hierarchy, with the most specific exception first. If the first catch was the most general, none of the others could be reached.

In traditional programming, error handling often makes the code more confusing to read. Java separates the details of handling errors from the main work of the program and they cannot be ignored.

The general form of exception handling block is given below:

```
try
{
   // block of code to monitor for errors
}
catch (ExceptionType exob)
{
   // code to handle exception occurred
}
catch (ExceptionType exob2)
{
   // code to handle exception
}
```

Let us consider an example

```
class Error3
{
   public static void main(String args[ ])
   {
      int a = 10, b = 5, c = 5, x, y;
      try
      {
         x = a/(b-c);
      }
```

```
        catch(Arithmetic Exception e)
        {
        System.out.println("Division by zero");
        }
        y = a/(b+c);
        System.out.println("y =" + y);
    }
}
```

Output:

```
Division by zero
y = 1
```

Note that the program did not stop at the point of exceptional condition. It catches the error, prints appropriate error message and then continues the program execution after catch block.

9.4 MULTIPLE CATCH STATEMENTS

In some cases more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
//Demonstrate multiple catch statements
Class Multicatch {
   public static void main(String args[j) {
      try {
         int a = args.length;
         system.out.println ("a="+a);
         int b = 42/a;
         int c [] ={1};
         c[42] = 99;
         } catch (Arithmetic Exception e) {
           system.out.println ("Divide by 0: "+e);
         } catch (array Index Out of Bounds Exception e) {
           system.out.println ("Array index oob: "+e);
         }
           system.out.println ("After try/Catch blocks");
         }
      }
```

This program will cause a division by zero exception if it is started with no command-line parameters, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an Array Index Out of Bound Exception, since the int array c has a length of l, yet the program attempts to assign a value to c[42].