LAB MANUAL

Department of Information Technology

Government College of Engineering, Karad.

IT2509 : Programming Lab I

T.Y.B.Tech (Information Technology)

Academic Year: 2021-22

Term-I

Programming Lab-I Plan

Sr. No	Title
1.	Implement program for Class, Objects and Methods
2.	Implement program for Constructor and Method Overloading
3.	Implement the concept of Inheritance and Interface
4.	Implement the program for Exception Handling
5.	Implement the concept of Multithreading
6.	Implement the concept of I/O Programming
7.	Implement Program for Applet with AWT controls
8.	Implement the frame and window concept using Swing
9.	Program to demonstrate Event handling concept.
10.	Implement a Client-Server Network programming
11.	Program to demonstrate various methods of Collection class
12.	Program for Database Connectivity using JDBC and ODBC

Experiment No. 01

Title : Implement the Concept of Classes, Object and Method

Aim : To implement the concept of,

1) How to define a class
2) How to create objects
3) How to add data fields and methods to classes
4) How to access data fields and methods to classes

Objective: Class is the building block of the Java Programming. For implementing Java program it's necessary to learn the classes, Objects and Methods.

Theory :

- Java is a true Object Oriented language and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the "state" and "behavior" of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them. They provide a convenient method for packaging a group of logically related data items and functions that work on them.
- A class essentially serves as a template for an object and behaves like a basic data type "int". It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as encapsulation, inheritance, and polymorphism.
- A class is a collection of fields (data) and methods (procedure or function) that operate on that data.
- The basic syntax for a class definition:

```
class  ClassName [extends SuperClassName]
{
        [fields declaration]
        [methods declaration]
}
```

- Bare bone class – no fields, no methods

```
public class Circle {
                    // my circle class
              }
```

Adding Fields: Class Circle with fields

Add fields:

```
public class Circle {
                public double x, y;  // centre coordinate
                public double r;     //  radius of the circle
                }
```

The fields (data) are also called the instance variables.

Adding Methods:

- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.

The general form of a method declaration is:

```
type MethodName (parameter-list)
        {
                Method-body;
        }
```

Adding Methods to Class Circle:

```
 public class Circle {
    public double x, y; // centre of the circle
    public double r;    // radius of circle

    //Methods to return circumference and area
    public double circumference() {
                return 2*3.14*r;
    }
    public double area() {
               return 3.14 * r * r;
    }
}
```

Data Abstraction

Declare the Circle class, have created a new data type – Data Abstraction Can define variables (objects) of that type:

```
Circle  aCircle;
Circle  bCircle;
```

aCircle, bCircle simply refers to a Circle object, not an object itself.

Creating objects of a class

```
aCircle = new Circle();
bCircle = new Circle() ;
bCircle = aCircle;
```

Accessing Object/Circle Data:

Similar to C syntax for accessing data defined in a structure.

```
ObjectName.VariableName
ObjectName.MethodName(parameter-list)
Circle aCircle = new Circle();
aCircle.x = 2.0 // initialize center and radius
aCircle.y = 2.0
aCircle.r = 1.0
```

Executing Methods in Object/Circle:

```
Circle aCircle = new Circle();
double area;                              sent ‘message’ to aCircle
aCircle.r = 1.0;
area = aCircle.area();
```

Using Circle Class:

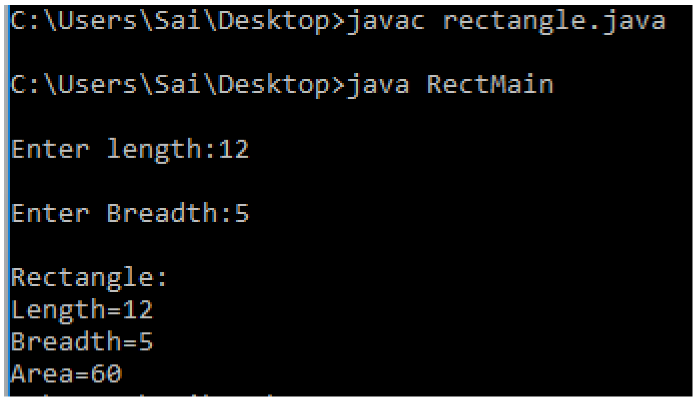
```
// Circle.java:  Contains both Circle class and its user class
//Add Circle class code here
class MyMain
{
      public static void main(String args[])
      {
            Circle aCircle;  // creating reference
            aCircle = new Circle(); // creating object
            aCircle.x = 10;  // assigning value to data field
```

```
                        aCircle.y = 20;
                        aCircle.r = 5;
                        double area = aCircle.area(); // invoking method
                        double circumf = aCircle.circumference();
                        System.out.println("Radius="+aCircle.r+" Area="+area);
                        System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
                }
        }
```

Program : A)<u>Program for calculation of area of rectangle:</u>

```
import java.util.*;
class  rectangle
{
  int length;
  int breadth;
  int area;
  rectangle()                                // Default cnstructor
  {
    Scanner scan=new Scanner(System.in);
    System.out.print("\nEnter length:");
    length=scan.nextInt();
    System.out.print("\nEnter Breadth:");
    breadth=scan.nextInt();
    this.Area();
  }
  void Area()                                // method to calculate area
  {
   this.area=this.length*this.breadth;
  }
  void show()                                //method to show data
  {
System.out.print("\nRectangle:\nLength="+this.length+"\nBreadth="+this.breadth+"\nArea=
"+this.area);  }
}
class RectMain
{
 public static void main(String args[])
 {
   rectangle rect=new rectangle();
   rect.show();
```

Output :



```
C:\Users\Sai\Desktop>javac rectangle.java

C:\Users\Sai\Desktop>java RectMain

Enter length:12

Enter Breadth:5

Rectangle:
Length=12
Breadth=5
Area=60
```

Conclusion: Thus we studied the Concept of Classes, Object and Method in Java Language.

Experiment No. 02

Title : Implement program for Constructor and Method overloading.

Aim : To understand and Implement the Constructor and Method overloading.

Objective: To study the importance of Constructor calling at the time of Object creation, and Types of it. We also study how to minimize the work by using Method overloading.

Theory : A constructor of a class is a special function which is automatically called when the object of the class is called. A constructor makes our work easier by initializing objects at their creation. The most important question is how ?

Special Features of Constructors –

These are some really important characteristics of Constructors.

■ Constructors can use any access modifier, including private. (A private constructor means only code within the class itself can instantiate an object of that type, so if the private constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
■ The constructor name must match the name of the class.
■ Constructors must not have a return type.
■ It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.
■ If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
■ The default constructor is ALWAYS a no-arg constructor.
■ If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself!
■ Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the superclass constructor (super()), although remember that this call can be inserted by the compiler.
■ If you do type in a constructor (as opposed to relying on the compiler-generated

default constructor), and you do not type in the call to super() or a call to this(), the compiler will insert a no-arg call to super() for you, as the very first statement in the constructor.

■ A call to super() can be either a no-arg call or can include arguments passed to the super constructor.

■ A no-arg constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-arg constructor. The default constructor is the one the compiler provides! While the default constructor is always a no-arg constructor, you're free to put in your own no-arg constructor.

■ You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs.

■ Only static variables and methods can be accessed as part of the call to super() or this(). (Example: super(Animal.NAME) is OK, because NAME is declared as a static variable.)

■ Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.

■ Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.

■ The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor as follows:

```
class Horse {
Horse() { } // constructor
void doStuff() {
Horse(); // calling the constructor - illegal!
}
}
```

Declaration of Constructors---

```
class any
        {
                public any() //Constructor
                {
                /* Body of Constructor
                Body continues ….
                */
                }
        }
```

Two Types of Constructors---
1.Default Constructor.
2.Parameterized Constructor

Default Constructor---A "*default constructor*" is a constructor with no parameters. You are already familiar with the concepts of parameters. The class constructed in the previous page contains a default constructor "*calc*".

Parameterized Constructor---A "*parameterized constructor*" is a constructor with parameters. The way of declaring and supplying parameters is same as that with methods.

Study this example—

```
class cons_param
{
int n,n2;
int sum,sub,mul,div;
public cons_param(int x,int y)
{
n=x;
n2=y;
}
public void opr()
{
sum=n+n2;
sub=n-n2;
mul=n*n2;
div=n/n2;
System.out.print("Sum is "+sum);
System.out.print("Difference is "+sub);
System.out.print("Product is "+mul);
System.out.print("Quotient is "+div);
}
}
class apply11
{
public static void main(String args[])
{
cons_param obj=new cons_param(6,17);
obj.opr();
}
}
```

Key Rule: The first line in a constructor must be a call to super() or a call to this().

If you have neither of those calls in your constructor, the compiler will insert the no-arg call to super(). In other words, if constructor A() has a call to this(), the compiler knows that constructor A() will not be the one to invoke super().

The preceding rule means a constructor can never have both a call to super() and a call to this(). Because each of those calls must be the first statement in a constructor, you can't legally use both in the same constructor. That also means the
compiler will not put a call to super() in any constructor that has a call to this().

Constructor Overloading---

Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:

```
class Cons {
Cons() { }
Cons(String s) { }
}
```

Lets see our next example of the apply series.

```
class overload
{
int n=0,n2=0;
public overload()
{
n=1;
n2=2;
System.out.println("n & n2 "+n+" "+n2);
}
public overload(int x)
{
n=x;
System.out.println("n & n2 "+n+" "+n2);
}
public overload(int x,int y)
{
n=x;
n2=y;
System.out.println("n & n2 "+n+" "+n2);
}
}
class apply12
{
public static void main(String args[])
{
overload obj=new overload();
overload obj1=new overload(5);
overload obj2=new overload(10,15);
}
}
```

Method Overloading:

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading.*

Method overloading is one of the ways that Java implements polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {

System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

Program :

A) Method Overloading:

```
import java.util.*;
```

```
class shapes
{ public String name;
  double perimeter;
  double area;
  public shapes()
  {  Scanner Scan=new Scanner(System.in);
    System.out.println("Enter name of shape:");
    this.name=Scan.next();
  }
  void getarea(double r)                    // Overloaded method getarea() with one double type
parameter
  {  this.perimeter=(2*3.142*r);
    this.area=(3.142*r*r);
    System.out.println("For Circle:\nCircumference="+perimeter+"\nArea="+area);
  }
  void getarea(int l,int b)                         // getarea() with two int parameters
  {  this.perimeter=2*(l+b);
     this.area=l*b;
     System.out.println("For Rectangle:\nPerimeter="+perimeter+"\nArea="+area);
   }
  void getarea(int s)                               // getarea() with one int type parameter
  {   this.perimeter=4*s;
      this.area=s*s;
      System.out.println("For Square:\nPerimeter="+perimeter+"\nArea="+area);
  }
  public static void main(String args[])
  {  int i;
     String str;
     Scanner sc=new Scanner(System.in);
    shapes shape[]=new shapes[3];
    for(i=0;i<3;i++)                                          // Looping construct- for
    {   shape[i]=new shapes();
       switch(shape[i].name)                              // Multi-way branching using switch
statement
  {
       case "circle": double radius;
                          System.out.println("Enter radius of circle:");
                          radius=sc.nextFloat();
                          shape[i].getarea(radius);
                          break;
       case "rectangle": int length,breadth ;
                              System.out.println("Enter length of rectangle:");
                              length=sc.nextInt();
                              System.out.println("Enter breadth of rectangle:");
                              breadth=sc.nextInt();
                              shape[i].getarea(length,breadth);
```

```
                                    break;
        case "square": int side;
                                System.out.println("Enter side of square:");
                                side=sc.nextInt();
                                shape[i].getarea(side);
                                break;
       default:  System.out.println("Invalid shape!");
      }}}}
```

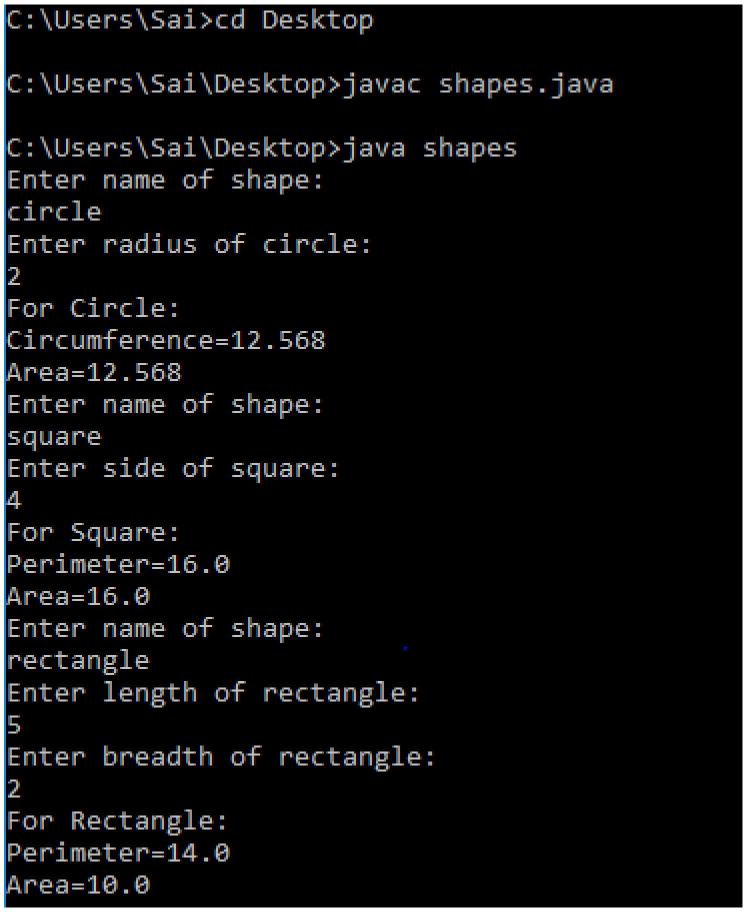
<u>B)Constructor Overloading:</u>

INPUT:

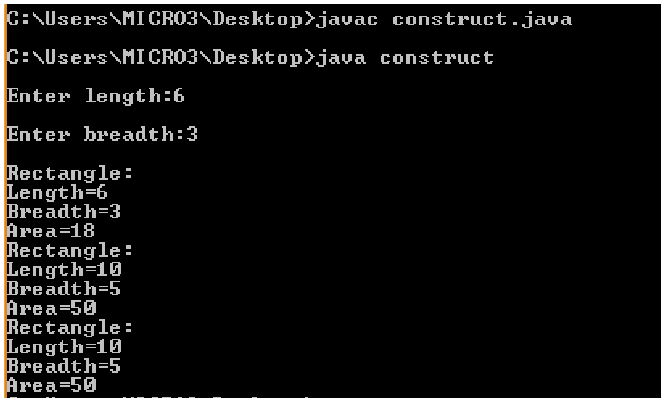
```
import java.util.*;
class  rectangle
{
  int length;
  int breadth;
  int area;
  rectangle()                                                    // Default Constructor
  {
    Scanner sc=new Scanner(System.in);
    System.out.print("\nEnter length:");
    length=sc.nextInt();
    System.out.print("\nEnter breadth:");
    breadth=sc.nextInt();
    area=length*breadth;
   }
   rectangle(int length, int breadth)                          // Parameterized constructor
  {
    this.length=length;
    this.breadth=breadth;
    area=this.length*this.breadth;
  }
  rectangle(rectangle r)                                       // Constructor with object as parameter
  {
    this.length=r.length;
    this.breadth=r.breadth;
    area=this.length*this.breadth;
  }
  void Display()
  {

System.out.print("\nRectangle:\nLength="+this.length+"\nBreadth="+this.breadth+"\nArea=
"+this.area);
  }
}
```

```
class construct
{
  public static void main(String args[])
  {
   rectangle rect1=new rectangle();
   rectangle rect2=new rectangle(10,5);
   rectangle rect3=new rectangle(rect2);
   rect1.Display();
   rect2.Display();
   rect3.Display();
  }
}
```

Output :



```
C:\Users\Sai>cd Desktop

C:\Users\Sai\Desktop>javac shapes.java

C:\Users\Sai\Desktop>java shapes
Enter name of shape:
circle
Enter radius of circle:
2
For Circle:
Circumference=12.568
Area=12.568
Enter name of shape:
square
Enter side of square:
4
For Square:
Perimeter=16.0
Area=16.0
Enter name of shape:
rectangle
Enter length of rectangle:
5
Enter breadth of rectangle:
2
For Rectangle:
Perimeter=14.0
Area=10.0
```



```
C:\Users\MICRO3\Desktop>javac construct.java

C:\Users\MICRO3\Desktop>java construct

Enter length:6

Enter breadth:3

Rectangle:
Length=6
Breadth=3
Area=18
Rectangle:
Length=10
Breadth=5
Area=50
Rectangle:
Length=10
Breadth=5
Area=50
```

Conclusion: This way we study the concept of Constructor, Method Overloading.

Experiment No. 03

Title : Implement the concept of Inheritance and Interface

Aim : To study Inheritance and Interface concept in Java.

Objective: Inheritance in java is nothing but the reusability of the existence code.

Theory : Central to Java and other object-oriented (OO) languages is the concept of *inheritance*, which allows code defined in one class to be reused in other classes. In Java, you can define a general (more abstract) superclass, and then extend it with more specific subclasses. The superclass knows nothing of the classes that inherit from it, but all of the subclasses that inherit from the superclass must explicitly declare the inheritance relationship. A subclass that inherits from a superclass is automatically given accessible instance variables and methods defined by the superclass, but is also free to override superclass methods to define more specific behavior.

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

When we talk about inheritance the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}
```

```
public class Dog extends Mammal{
}
```

Now based on the above example, In Object Oriented terms following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are sub classes of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now if we consider the IS-A relationship we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass. We can assure that Mammal is actually an Animal with the use of the instance operator.

Example:

```
public class Dog extends Mammal{
  public static void main(String args[]){

    Animal a = new Animal();
    Mammal m = new Mammal();
    Dog d = new Dog();

    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

This would produce following result:

```
true
```

```
true
true
```

Since we have a good understanding of the **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship.

The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

Example:

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example:

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
        private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

In Object Oriented feature the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Vann class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

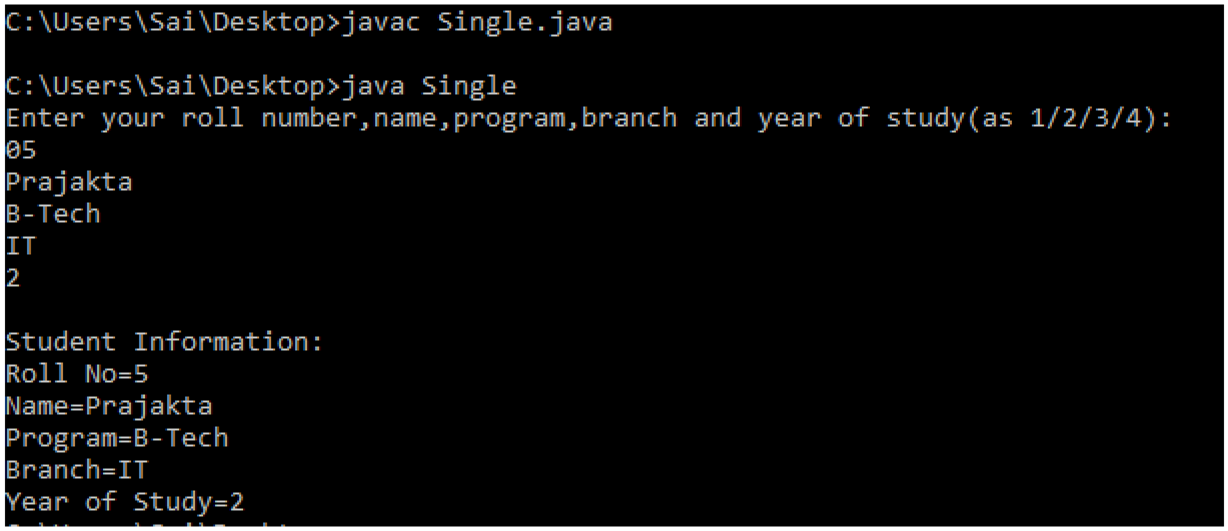
```
public class extends Animal, Mammal{}
```

However a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance

A)<u>Program for Single Inheritance:</u>

```
import java.util.*;
class course                                          // superclass
{
  String Branch;
  int Year;
  String Program;
  course(String branch, int year, String program)
  {
    Branch=branch;
    Year=year;
    Program=program;
  }
  final void show()
  {
   System.out.print("\nProgram="+Program+"\nBranch="+Branch+"\nYear of
Study="+Year);
  }
}
class student extends course                          //subclass
{
  int RollNo;
  String name;
  student(String b, int y, String p, int r, String n)
  {
   super(b,y,p);                                      // use of super() to call constructor
   RollNo=r;
   name=n;
  }
 void Display()
 {
  System.out.print("\nStudent Information:\nRoll No="+RollNo+"\nName="+name);
  super.show();                                    //use of super for other members
 }
}
class Single
{
 public static void main(String args[])
 {
   String strb,strp,strn;
   int inty,intr;
   Scanner scan=new Scanner(System.in);
   System.out.println("Enter your roll number,name,program,branch and year of study(as
1/2/3/4):");
   intr=scan.nextInt();
   strn=scan.next();
   strp=scan.next();
   strb=scan.next();
   inty=scan.nextInt();
   student stu=new student(strb,inty,strp,intr,strn);
   stu.Display();
 }
}
```

OUTPUT:



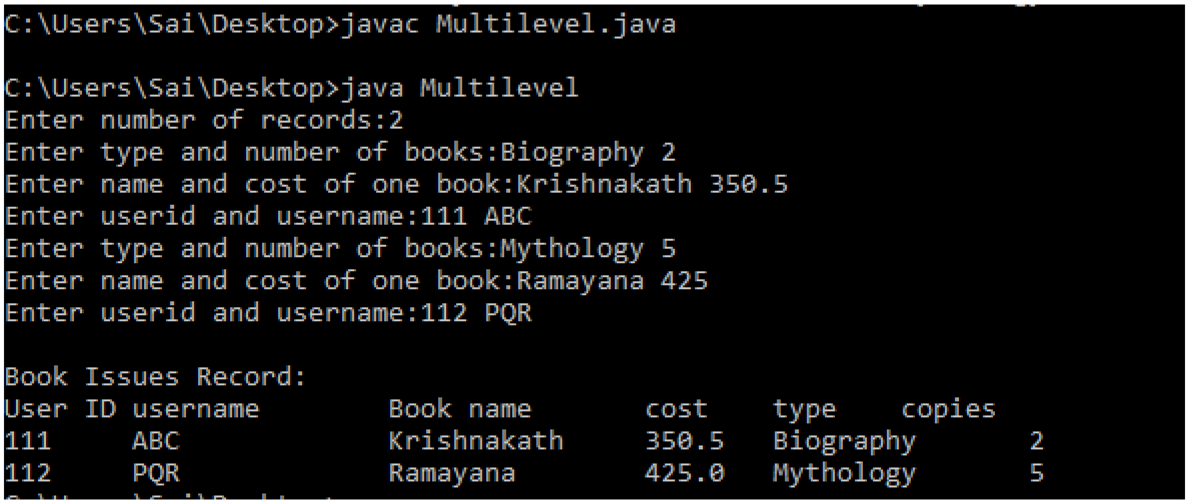
```
C:\Users\Sai\Desktop>javac Single.java

C:\Users\Sai\Desktop>java Single
Enter your roll number,name,program,branch and year of study(as 1/2/3/4):
05
Prajakta
B-Tech
IT
2

Student Information:
Roll No=5
Name=Prajakta
Program=B-Tech
Branch=IT
Year of Study=2
```

B)<u>Program for Multilevel Inheritance:</u>

INPUT:

```
import java.util.*;
class library
{
 protected String booktype;
 protected int nob;
 public library()
 {
  Scanner sc1=new Scanner(System.in);
  System.out.print("Enter type and number of books:");
  booktype=sc1.next();
  nob=sc1.nextInt();
 }
 void show()
 {
  System.out.print(booktype+"\t"+nob);
 }
}
class book extends library
{
 protected String Bname;
 protected float cost;
 public book()
 {
  super();                                          // super () to call default constructor
  Scanner sc2=new Scanner(System.in);
  System.out.print("Enter name and cost of one book:");
  Bname=sc2.next();
  cost=sc2.nextFloat();
 }
 void show()
 {
  System.out.print("\t"+Bname+"\t"+cost+"\t");
  super.show();
 }}
class reader extends book
{
 int userID;
 String username;
 reader()
 {
  super();
  Scanner sc3=new Scanner(System.in);
  System.out.print("Enter userid and username:");
  userID=sc3.nextInt();
  username=sc3.next();
 }
 void show()
 {
  System.out.print("\n"+userID+"\t"+username+"\t");
  super.show();
 }}
class Multilevel
{
  public static void main(String args[])
  {
    int i,n;
    Scanner sc=new Scanner(System.in);
    System.out.print("Enter number of records:");
    n=sc.nextInt();
    reader r[]=new reader[n];
    for(i=0;i<n;i++)
    { r[i]=new reader(); }
    System.out.print("\nBook Issues Record:\nUser ID\tusername\tBook
name\tcost\ttype\tcopies");
    for(i=0;i<n;i++)
    { r[i].show(); }
  }}
```

OUTPUT:



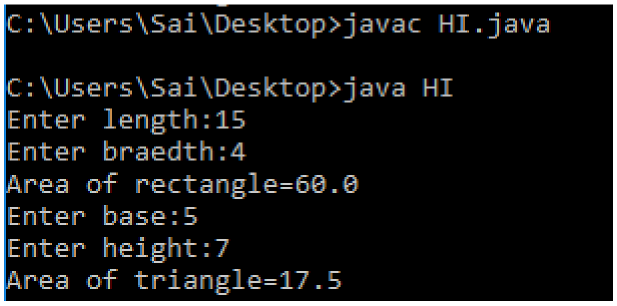
```
C:\Users\Sai\Desktop>javac Multilevel.java

C:\Users\Sai\Desktop>java Multilevel
Enter number of records:2
Enter type and number of books:Biography 2
Enter name and cost of one book:Krishnakath 350.5
Enter userid and username:111 ABC
Enter type and number of books:Mythology 5
Enter name and cost of one book:Ramayana 425
Enter userid and username:112 PQR

Book Issues Record:
User ID username        Book name       cost    type    copies
111     ABC             Krishnakath     350.5   Biography       2
112     PQR             Ramayana        425.0   Mythology       5
```

C)Program for Hierarchical Inheritance:

INPUT:

```
import java.util.*;
abstract class shape                                    // abstract class
{
 int dim1,dim2;
 double area;
 abstract void calcarea();                              // abstract method
 Scanner sc=new Scanner(System.in);
}
class rectangle extends shape
{
 rectangle()
 {
   System.out.print("Enter length:");
   dim1=sc.nextInt();
   System.out.print("Enter braedth:");
   dim2=sc.nextInt();
 }
 void calcarea()
 {
  area=dim1*dim2;
  System.out.print("Area of rectangle="+area);
 }
}
class triangle extends shape
{
 triangle()
 {
   System.out.print("\nEnter base:");
   dim1=sc.nextInt();
   System.out.print("Enter height:");
   dim2=sc.nextInt();
 }
 void calcarea()
 {
  area=0.5*dim1*dim2;
  System.out.print("Area of triangle="+area);
 }
}
class HI
{
 public static void main(String args[])
 {
   rectangle rect=new rectangle();
   rect.calcarea();
   triangle tri=new triangle();
   tri.calcarea();
 }
}
```

OUTPUT:



```
C:\Users\Sai\Desktop>javac HI.java

C:\Users\Sai\Desktop>java HI
Enter length:15
Enter braedth:4
Area of rectangle=60.0
Enter base:5
Enter height:7
Area of triangle=17.5
```

Interface:-

Interfaces are used to encode similarities which classes of various types share, but do not 4necessarily constitute a class relationship. For instance, a human and a parrot can both whistle, however it would not make sense to represent Humans and Parrots as subclasses of a Whistler class, rather they would most likely be subclasses of an Animal class (likely with intermediate classes), but both would implement the Whistler interface.

Another use of interfaces is being able to use an object without knowing its type of class, but rather only that it implements a certain interface. For instance, if one were annoyed by a whistling noise, one may not know whether it is a human or a parrot, all that could be determined is that a whistler is whistling. In a more practical example, a sorting algorithm may expect an object of type Comparable. Thus, it knows that the object's type can somehow be sorted, but it is irrelevant what the type of the object is. The call whistler.whistle() will call the implemented method whistle of object whistler no matter what class it has, provided it implements Whistler.

For example:

```
 interface Bounceable
 {
 public abstract void setBounce();/*Interface methods are by default public and abstract and
the methods in an interface ends  with a semicolon not with curly brace.*/
 }
```

Usage

Defining an interface

Interfaces are defined with the following syntax (compare to Java's class definition).

[visibility] interface **InterfaceName** [extends other interfaces] {
    constant declarations
    member type declarations

```
    abstract method declarations
}
```

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the abstract keyword is not required. Since the interface specifies a set of exposed behaviours, all methods are implicitly public.

Thus, a simple interface may be

```
public interface Predator {
    boolean chasePrey(Prey p);
    void eatPrey(Prey p);
}
```

The member type declarations in an interface are implicitly static and public, but otherwise they can be any type of class or interface.

Implementing an interface

The syntax for implementing an interface uses this formula:

```
... implements InterfaceName[, another interface, another, ...] ...
```

Classes may implement an interface. For example,

```
public class Cat implements Predator {

    public boolean chasePrey(Prey p) {
        // programming to chase prey p (specifically for a cat)
    }

    public void eatPrey (Prey p) {
        // programming to eat prey p (specifically for a cat)
    }
}
```

If a class implements an interface and is not abstract, and does not implement all its methods, it must be marked as abstract. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

```
public class Frog implements Predator, Prey { ... }
```

Interfaces are commonly used in the Java language for callbacks. Java does not allow the passing of methods (procedures) as arguments. Therefore, the practice is to define an interface and use it as the argument and use the method signature knowing that the signature will be later implemented.

Subinterfaces

Interfaces can extend several other interfaces, using the same formula are described above.

For example

```
public interface VenomousPredator extends Predator, Venomous {
       //interface body
}
```

is legal and defines a subinterface. Note how it allows multiple inheritance, unlike classes. Note also that Predator and Venomous may possibly define or inherit methods with the same signature, say kill(Prey prey). When a class implements VenomousPredator it will implement both methods simultaneously.

Examples

Some common Java interfaces are:

- Comparable has the method compareTo, which is used to describe two objects as equal, or to indicate one is greater than the other. Generics allow implementing classes to specify which class instances can be compared to them.
- Serializable is a marker interface with no methods or fields - it has an empty body. It is used to indicate that a class can be serialized. Its Javadoc describes how it should function, although nothing is programmatically enforced.

A)Simple Interface Program:

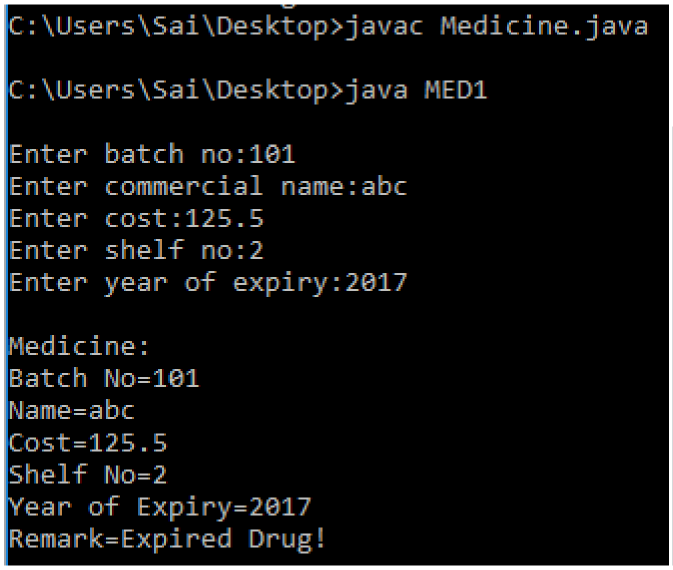
INPUT:

```
import java.util.Scanner;
public interface Medicine
{
  public void get();
  public void showMed();
  public void check(int y);
}
```

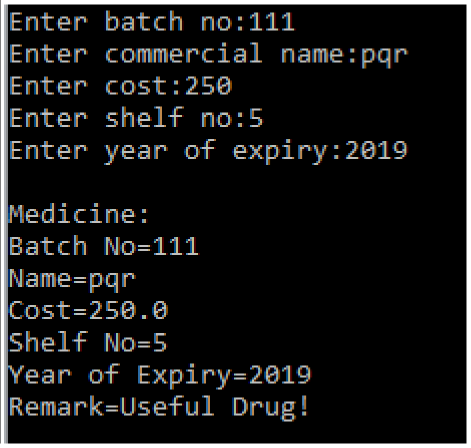
```
class medicines implements Medicine
{
 int BatchNo;
 String name;
 float cost;
 int shelfNo;
 int YOE;
 String remark;
 public void get()
 {
   Scanner sc1=new Scanner(System.in);
   System.out.print("\nEnter batch no:");
   BatchNo=sc1.nextInt();
   System.out.print("Enter commercial name:");
   name=sc1.next();
   System.out.print("Enter cost:");
   cost=sc1.nextFloat();
   System.out.print("Enter shelf no:");
   shelfNo=sc1.nextInt();
   System.out.print("Enter year of expiry:");
   YOE=sc1.nextInt();
 }
 public void check(int y)
 {
   if(y>=2018)
    this.remark="Useful Drug!";
   else
    this.remark="Expired Drug!";
 }
 public void showMed()
 {
  this.check(this.YOE);
  System.out.print("\nMedicine:\nBatch
No="+BatchNo+"\nName="+name+"\nCost="+cost+"\nShelf No="+shelfNo+"\nYear of
Expiry="+YOE+"\nRemark="+remark+"\n");
 }}
 class MED1
{
 public static void main(String args[])
 {
   medicines med1=new medicines();
   med1.get();
   med1.showMed();
   medicines med2=new medicines();
   med2.get();
   med2.showMed();
```

}}

OUTPUT:



```
C:\Users\Sai\Desktop>javac Medicine.java

C:\Users\Sai\Desktop>java MED1

Enter batch no:101
Enter commercial name:abc
Enter cost:125.5
Enter shelf no:2
Enter year of expiry:2017

Medicine:
Batch No=101
Name=abc
Cost=125.5
Shelf No=2
Year of Expiry=2017
Remark=Expired Drug!
```



```
Enter batch no:111
Enter commercial name:pqr
Enter cost:250
Enter shelf no:5
Enter year of expiry:2019

Medicine:
Batch No=111
Name=pqr
Cost=250.0
Shelf No=5
Year of Expiry=2019
Remark=Useful Drug!
```

B)Interface Program for Banking:

INPUT:

```
import java.util.*;
public interface Account                    // Interface
{    int interest_rate=6;                   // Final variable
 void CreateAcc();                          // interface methods – abstract methods
 void Credit();
 void Debit();
 void AddInterest();
 void ShowAccount(); }
class savingsAcc implements Account      // class savingsAcc implementing interface
{   String AccHolder;
  int AccNo;
 float amount;
float percentcharge=0.5f;
static int s_cntr,min_balance;
  static
  {  s_cntr=0;
 min_balance=500;   }
```

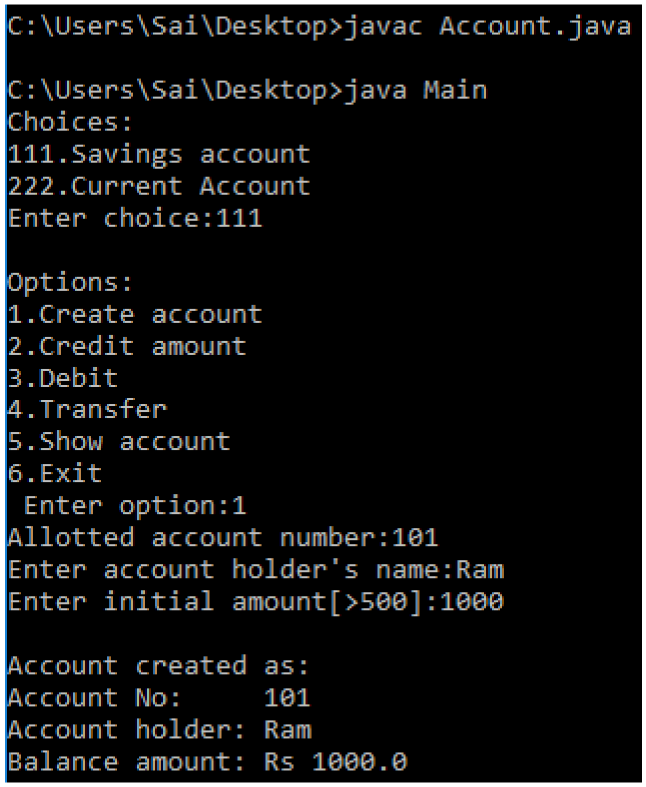
```
  savingsAcc(int a)
  {  AccNo=101+a;
System.out.print("Allotted account number:"+AccNo);
   CreateAcc();   }
  public void CreateAcc()
{   Scanner scan=new Scanner(System.in);
 System.out.print("\nEnter account holder's name:");
 AccHolder=scan.next();
 System.out.print("Enter initial amount[>"+min_balance+"]:");
  amount=scan.nextFloat();
  System.out.print("\nAccount created as:");
 this.ShowAccount();
savingsAcc.s_cntr=s_cntr+1;   }
 public void Credit()
 {   float amt;
Scanner Sc1=new Scanner(System.in);
System.out.print("\nEnter amount to be credited:");
amt=Sc1.nextFloat();
 this.amount=this.amount+amt;
System.out.print("\nAmount credited successfully!");
   ShowAccount();    }
 public void Debit()
 {  float amt1;
Scanner Sc2=new Scanner(System.in);
 System.out.print("Your available balance is "+(this.amount-min_balance)+"\nEnter amount
to be debitted:");
   amt1=Sc2.nextFloat();
   if(this.amount>=(amt1+min_balance))
   {this.amount=this.amount-amt1;
   System.out.print("\nAmount debitted successfully!");
   this.ShowAccount();    }
   else
   System.out.print("\nNo sufficient balance!");
 }
  public void Transfer(savingsAcc s[])
 {   int Tacc,no;
float charges;
float amt2;
Scanner Sc3=new Scanner(System.in);
 System.out.print("Enter destination account number:");
 no=Sc3.nextInt();
Tacc=no-101;
System.out.print("Enter amount to be transferred:");
amt2=Sc3.nextFloat();
   if(this.amount>=(amt2+min_balance))
   {   charges=amt2*percentcharge/100;
 this.amount=this.amount-(amt2+charges);
s[Tacc].amount=s[Tacc].amount+amt2;
 System.out.print("\nAmount transferred successfully!"+"\nSource account:");
  ShowAccount();
 System.out.print("\nCharges deducted: Rs "+charges);
 System.out.print("\nDestination account:");
     s[Tacc].ShowAccount();    }
  else
  System.out.print("\nNo sufficient balance!");
 }
 public void AddInterest()
 {  float interest=this.amount*interest_rate/100;
  this.amount=this.amount+interest;
   ShowAccount();  }
 public void ShowAccount()
 { System.out.print("\nAccount No:"+this.AccNo+"\nAccount holder's
name:"+this.AccHolder+"\nBalance amount: Rs "+this.amount);
 }}
class currentAcc implements Account       // class currentAcc implementing interface
{String enterprise;
int accno;
 float balance;
 float Draft_amount;
static int OverDraft;
 static int c_cntr,min_amount;
 static
 {  c_cntr=0;
OverDraft=10;
min_amount=5000;   }
 currentAcc(int x)
 {  accno=1001+x;
 System.out.print("\nAllotted account number is "+accno);
  CreateAcc();    }
 public void CreateAcc()
 {  Scanner scan1=new Scanner(System.in);
System.out.print("\nEnter name of enterprise:");
enterprise=scan1.next();
 System.out.print("Enter initial amount[>"+min_amount+"]:");
balance=scan1.nextFloat();
System.out.print("\nAccount created as:");
this.ShowAccount();
currentAcc.c_cntr=c_cntr+1;     }
 public void Credit()
 {  float amtc1;
Scanner scan2=new Scanner(System.in);
System.out.print("\nEnter amount to be credited:");
```

```
amtc1=scan2.nextFloat();
this.balance=this.balance+amtc1;
 System.out.print("\nAmount credited successfully!");
 ShowAccount();     }
  public void Debit()
 {  float amtc2;
Scanner scan3=new Scanner(System.in);
Draft_amount=this.balance*OverDraft/100;
System.out.print("\nYour available balance is "+this.balance+"\n Available amount
including overdraft:"+(balance+Draft_amount));
 System.out.print("\n Enter amount to be debitted:");
 amtc2=scan3.nextFloat();
 if((this.balance+Draft_amount)>=amtc2)
{this.balance=this.balance-amtc2;
System.out.print("\nAmount debitted successfully!");
 this.ShowAccount();  }
   else
 System.out.print("\nNo sufficient balance!");
  }
  public void Transfer(currentAcc c[])
  {   int Cacc,no1;
 float amtc3;
 Scanner scan4=new Scanner(System.in);
System.out.print("Enter destination account number:");
 no1=scan4.nextInt();
Cacc=no1-1001;
System.out.print("Enter amount to be transferred:");
 amtc3=scan4.nextFloat();
 Draft_amount=this.balance*OverDraft/100;
   if((this.balance+Draft_amount)>=amtc3)
   {  this.balance=this.balance-amtc3;
  c[Cacc].balance=c[Cacc].balance+amtc3;
 System.out.print("\nAmount transferred successfully!"+"\nSource account:");
ShowAccount();
 System.out.print("\nDestination account:");
c[Cacc].ShowAccount();    }
  else
  System.out.print("\nNo sufficient balance!");
 }
 public void AddInterest()
 {  System.out.print("Current accounts do not offer any interest!");    }
 public void ShowAccount()
 {      System.out.print("\nAccount No:"+this.accno+"\nAccount holder
enterprise:"+this.enterprise+"\nBalance amount: Rs "+this.balance);
 }}
class Main
```

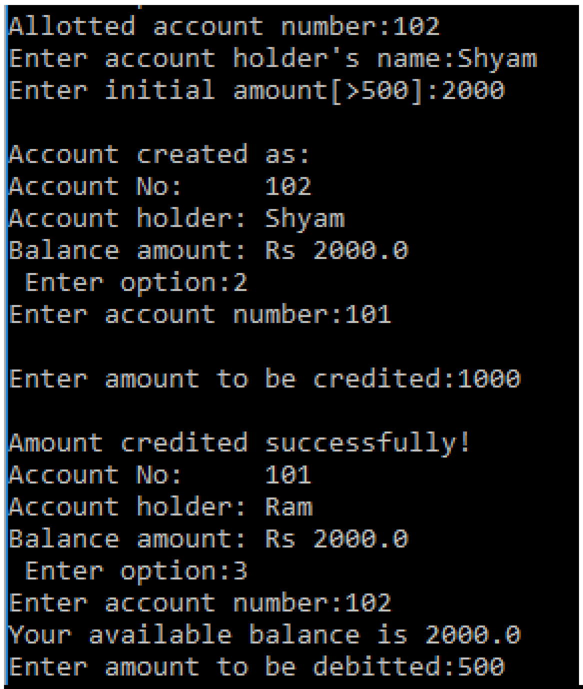
```
{  public static void main(String args[])
 {  savingsAcc sAcc[]=new savingsAcc[5];
currentAcc cAcc[]=new currentAcc[5];
int op,an1,an2;
 int choice;
 Scanner SCAN=new Scanner(System.in);
 System.out.print("Choices:\n111.Savings account\n222.Current Account\nEnter choice:");
choice=SCAN.nextInt();
System.out.print("\nOptions:\n1.Create account\n2.Credit
amount\n3.Debit\n4.Transfer\n5.Show account\n6.Exit ");
   do
   {System.out.print("\n Enter option:");
op=SCAN.nextInt();
     switch(op)
     {
       case 1:if(choice==111)   sAcc[savingsAcc.s_cntr]=new savingsAcc(savingsAcc.s_cntr);
                else                    cAcc[currentAcc.c_cntr]=new currentAcc(currentAcc.c_cntr);
            break;
       case 2:if(choice==111)
               {  System.out.print("Enter account number:");
            an1=SCAN.nextInt();
       sAcc[an1-101].Credit();   }
            else
             {  System.out.print("Enter account number:");
            an1=SCAN.nextInt();
cAcc[an1-1001].Credit();  }
            break;
       case 3:if(choice==111)
            {  System.out.print("Enter account number:");
            an1=SCAN.nextInt();
 sAcc[an1-101].Debit();    }
            else
             {   System.out.print("Enter account number:");
            an1=SCAN.nextInt();
        cAcc[an1-1001].Debit();   }
            break;
       case 4: if(choice==111)
                {   System.out.print("Enter account number of source account:");
            an1=SCAN.nextInt();
     sAcc[an1-101].Transfer(sAcc);  }
            else
                {   System.out.print("Enter account number of source account:");
            an1=SCAN.nextInt();
 cAcc[an1-1001].Transfer(cAcc);   }
            break;
       case 5:if(choice==111)
```

```
                {  System.out.print("Enter account number:");
            an1=SCAN.nextInt();
sAcc[an1-101].ShowAccount();   }
            else
                { System.out.print("Enter account number:");
            an1=SCAN.nextInt();
  cAcc[an1-1001].ShowAccount();    }
            break;
       case 6:System.out.print("\nExited!");
            break;
       default:System.out.print("\nIncorrect choice!");
     }}while(op<6);
}}
```

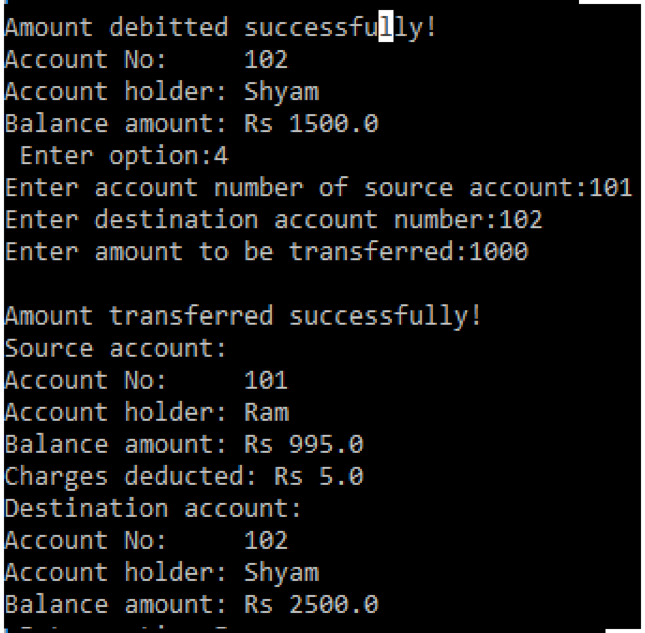
OUTPUT:



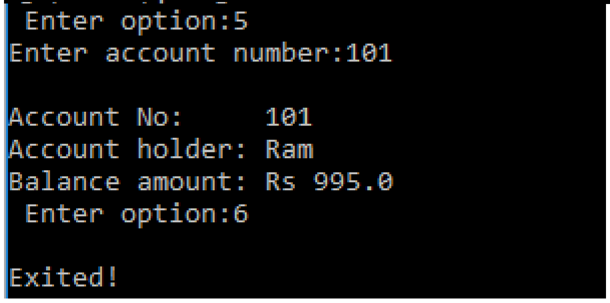
```
C:\Users\Sai\Desktop>javac Account.java

C:\Users\Sai\Desktop>java Main
Choices:
111.Savings account
222.Current Account
Enter choice:111

Options:
1.Create account
2.Credit amount
3.Debit
4.Transfer
5.Show account
6.Exit
 Enter option:1
Allotted account number:101
Enter account holder's name:Ram
Enter initial amount[>500]:1000

Account created as:
Account No:     101
Account holder: Ram
Balance amount: Rs 1000.0
```



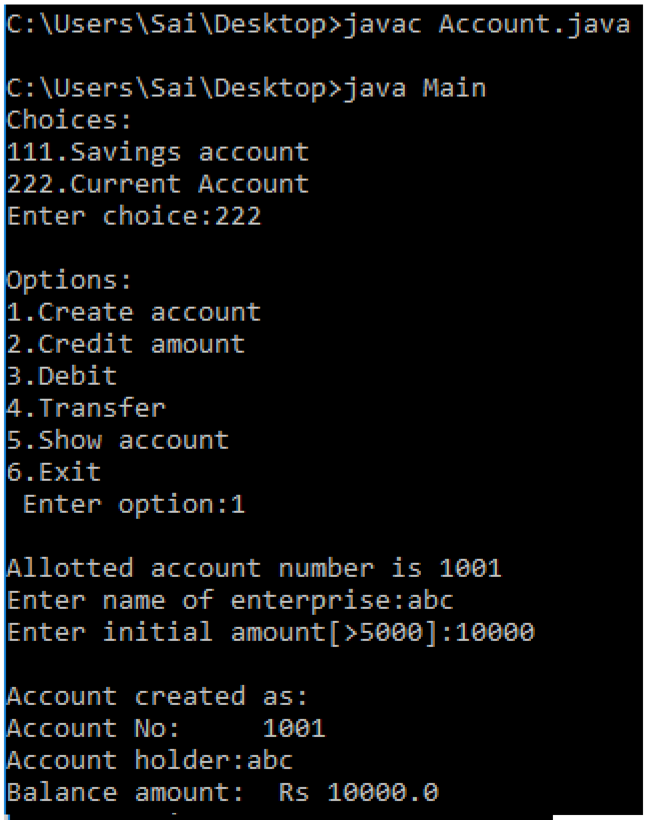
```
Allotted account number:102
Enter account holder's name:Shyam
Enter initial amount[>500]:2000

Account created as:
Account No:     102
Account holder: Shyam
Balance amount: Rs 2000.0
 Enter option:2
Enter account number:101

Enter amount to be credited:1000

Amount credited successfully!
Account No:     101
Account holder: Ram
Balance amount: Rs 2000.0
 Enter option:3
Enter account number:102
Your available balance is 2000.0
Enter amount to be debitted:500
```



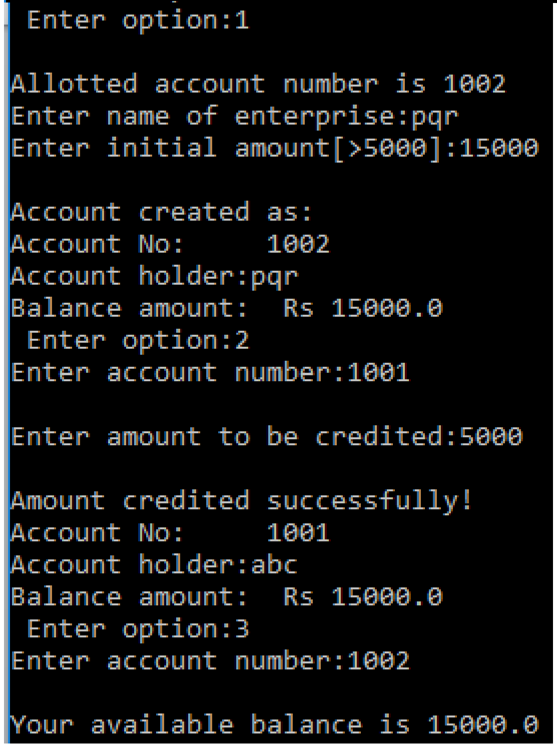
```
Amount debitted successfully!
Account No:     102
Account holder: Shyam
Balance amount: Rs 1500.0
 Enter option:4
Enter account number of source account:101
Enter destination account number:102
Enter amount to be transferred:1000

Amount transferred successfully!
Source account:
Account No:     101
Account holder: Ram
Balance amount: Rs 995.0
Charges deducted: Rs 5.0
Destination account:
Account No:     102
Account holder: Shyam
Balance amount: Rs 2500.0
```



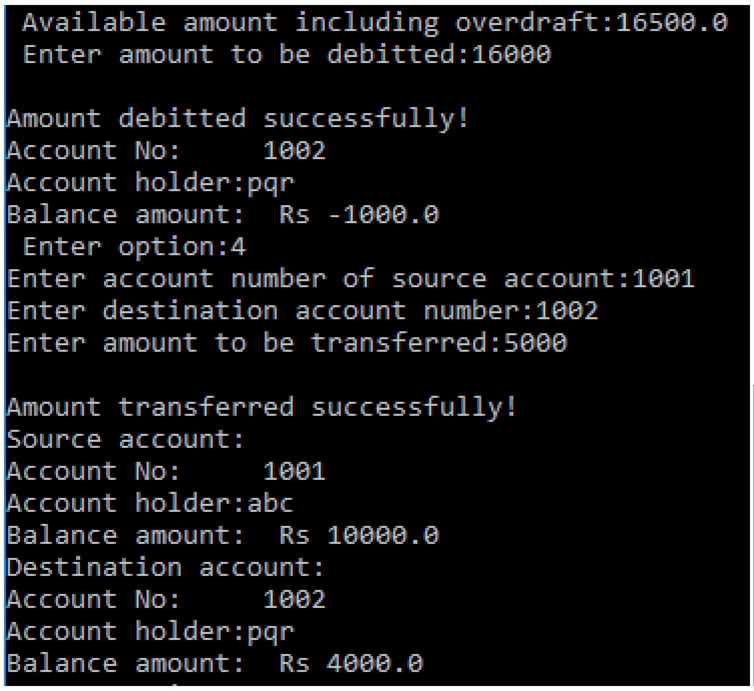
```
 Enter option:5
Enter account number:101

Account No:     101
Account holder: Ram
Balance amount: Rs 995.0
 Enter option:6

Exited!
```



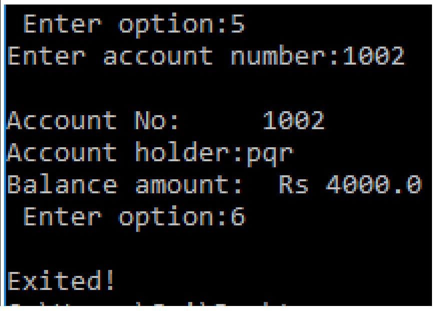
```
C:\Users\Sai\Desktop>javac Account.java

C:\Users\Sai\Desktop>java Main
Choices:
111.Savings account
222.Current Account
Enter choice:222

Options:
1.Create account
2.Credit amount
3.Debit
4.Transfer
5.Show account
6.Exit
 Enter option:1

Allotted account number is 1001
Enter name of enterprise:abc
Enter initial amount[>5000]:10000

Account created as:
Account No:     1001
Account holder:abc
Balance amount:  Rs 10000.0
```



```
 Enter option:1

Allotted account number is 1002
Enter name of enterprise:pqr
Enter initial amount[>5000]:15000

Account created as:
Account No:     1002
Account holder:pqr
Balance amount:  Rs 15000.0
 Enter option:2
Enter account number:1001

Enter amount to be credited:5000

Amount credited successfully!
Account No:     1001
Account holder:abc
Balance amount:  Rs 15000.0
 Enter option:3
Enter account number:1002

Your available balance is 15000.0
```



```
 Available amount including overdraft:16500.0
 Enter amount to be debitted:16000

Amount debitted successfully!
Account No:     1002
Account holder:pqr
Balance amount:  Rs -1000.0
 Enter option:4
Enter account number of source account:1001
Enter destination account number:1002
Enter amount to be transferred:5000

Amount transferred successfully!
Source account:
Account No:     1001
Account holder:abc
Balance amount:  Rs 10000.0
Destination account:
Account No:     1002
Account holder:pqr
Balance amount:  Rs 4000.0
```



```
 Enter option:5
Enter account number:1002

Account No:     1002
Account holder:pqr
Balance amount:  Rs 4000.0
 Enter option:6

Exited!
```

Conclusion: In This way we studied the concept of Inheritance with program of Multilevel and Hierarchical Inheritance and Interface.

Experiment No. 04

Title : Implementing the concept of Exception handling

Aim : To Study

1. How to monitor code for Exception
2. How to Catch exception
3. How to use throws and finally clauses
4. how to create our own exception class

Objective: Exception handling provides provision to recover from the failure condition

Theory :

- An exception is an abnormal condition that arises in a code sequence at run time.
- 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
- An exception can be caught to handle it or pass it on
- 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 via five keywords: **try, catch, throw, throws,** and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it
- System generated exceptions are automatically thrown by the Java run-time system
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause

■ Any code that absolutely must be executed before a method returns is put in a **finally** block

■ General form of an exception-handling block

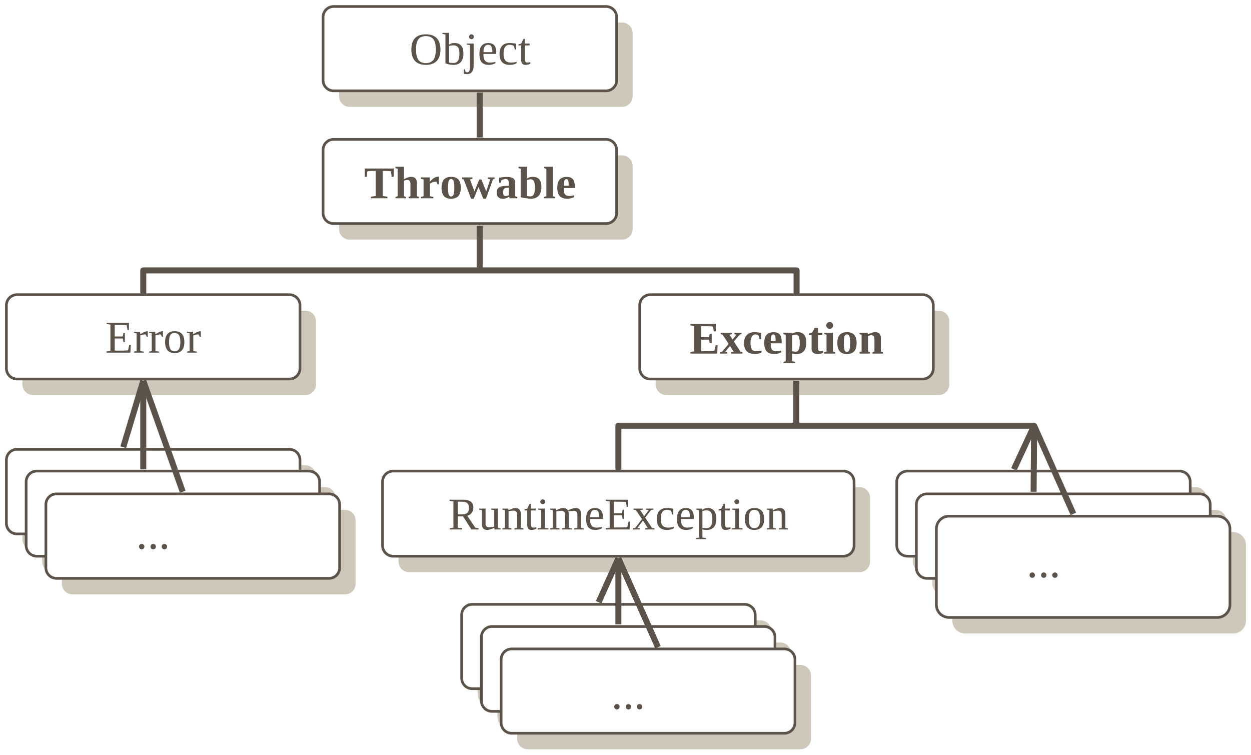
```
try{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb){
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb){
        // exception handler for ExceptionType2
}
//…
finally{
        // block of code to be executed before try block ends
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - Error (to handle exceptional conditions that are not expected to be caught under normal circumstances). i.e. stack overflow

Java Exception Hierarchy

Throwable	The base class for all exceptions, it is required for a class to be the rvalue to a **throw** statement.
Error	Any exception so severe it should be allowed to pass **uncaught** to the Java runtime.
Exception	Anything which should be handled by the invoker is of this type, and all but five exceptions are.



finally

- It is used to handle premature execution of a method (i.e. a method open a file upon entry and closes it upon exit)
- **finally** creates a block of code that will be executed after **try/catch** block has completed and before the code following the **try/catch** block
- **finally** clause will execute whether or not an exception is thrown

Creating your Own Exception Classes

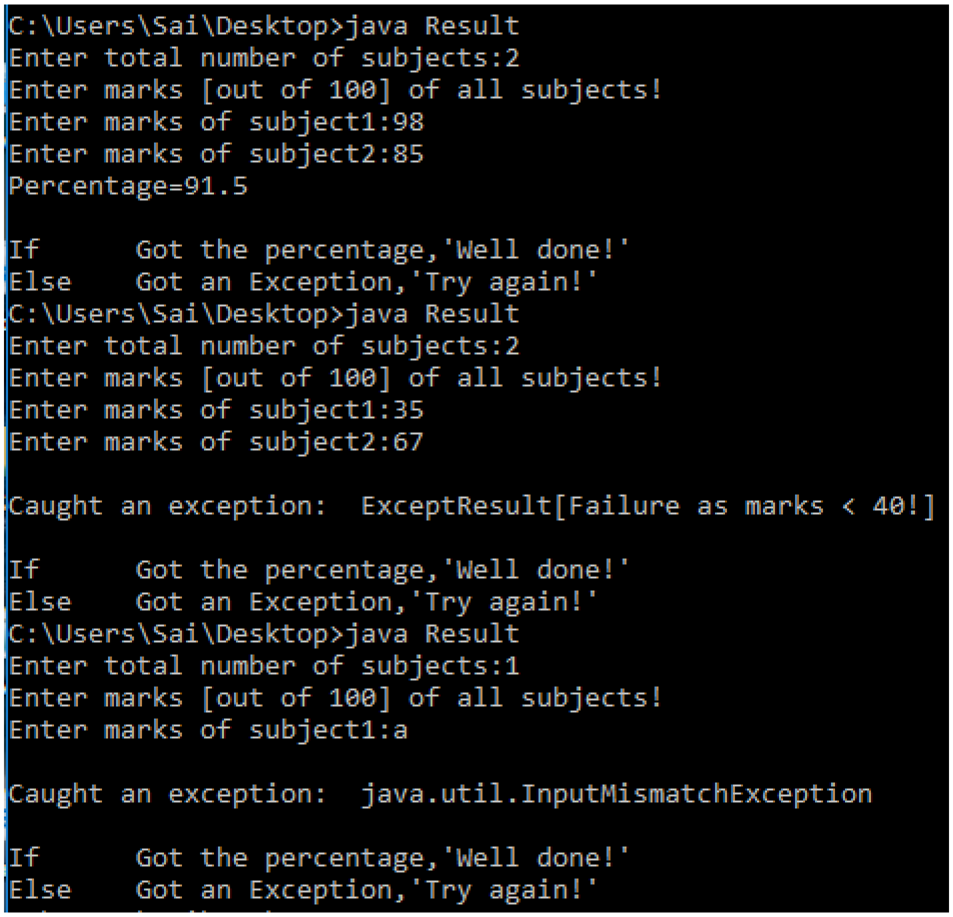
- You may not find a good existing exception class
- Can subclass Exception to create your own
- Give a default constructor and a constructor that takes a message

```
class MyFileException extends IOException
{
        public MyFileException ( ) { … }

        public MyFileException(String message)
        {
                super(message);
        }
}
```

Program : INPUT:

```
import java.util.*;
class ExceptResult extends Exception
{  String detail;
 ExceptResult(String str)
 {  detail=str;  }
 public String getMessage()
 {  return "ExceptResult["+this.detail+"]";
 }}
class Result
{ int n;
 int marks[];
 double percentage;
 void getMarks()
 {
   int flag=0;
   Scanner sc=new Scanner(System.in);
   System.out.print("Enter total number of subjects:");
   n=sc.nextInt();
   this.marks=new int[n];
   try
   {
    System.out.print("Enter marks [out of 100] of all subjects!\n");
    for(int i=0;i<n;i++)
    {
     System.out.print("Enter marks of subject"+(i+1)+":");
     marks[i]=sc.nextInt();
     if(marks[i]<40) {flag=1;}
    }
    if(flag==1) {throw new ExceptResult("Failure as marks < 40!");}
    else {this.calResult();}
   }
   catch(ExceptResult er)
   {
    System.out.print("\nCaught an exception:  "+er.getMessage());
   }
   catch(Exception e)
   {
    System.out.print("\nCaught an exception:  "+e);
   }
   finally
   {
    System.out.print("\n\nIf\tGot the percentage,'Well done!'\nElse\tGot an Exception,'Try
again!'");
   }}
 void calResult()
 {
  int sum=0;
  for(int i=0;i<this.n;i++)
  {  sum=sum+this.marks[i];   }
  this.percentage=((double)sum/this.n);
  System.out.print("Percentage="+this.percentage);
 }
 public static void main(String args[])
 {
  Result r1=new Result();
  r1.getMarks();
 }}
```

Output :



```
C:\Users\Sai\Desktop>java Result
Enter total number of subjects:2
Enter marks [out of 100] of all subjects!
Enter marks of subject1:98
Enter marks of subject2:85
Percentage=91.5

If      Got the percentage,'Well done!'
Else    Got an Exception,'Try again!'
C:\Users\Sai\Desktop>java Result
Enter total number of subjects:2
Enter marks [out of 100] of all subjects!
Enter marks of subject1:35
Enter marks of subject2:67

Caught an exception:  ExceptResult[Failure as marks < 40!]

If      Got the percentage,'Well done!'
Else    Got an Exception,'Try again!'
C:\Users\Sai\Desktop>java Result
Enter total number of subjects:1
Enter marks [out of 100] of all subjects!
Enter marks of subject1:a

Caught an exception:  java.util.InputMismatchException

If      Got the percentage,'Well done!'
Else    Got an Exception,'Try again!'
```

Conclusion: Thus we have studied Exception handling in different ways.

Experiment No. 05

Title : Implementing the concept of Multithreading

Aim : To study the concept of Multithreading how to handle multiple tasks simultaneously.

Objective: Multithreading is required in parallel application where simultaneous execution of the code is required.

Theory : A thread is the flow of execution of a single set of program statements. Multithreading consists of multiple sets of statements which can be run in parallel. With a single processor only one thread can run at a time but strategies are used to make it appear as if the threads are running simultaneously. Depending on the operating system scheduling method, either time slicing or interrupt methods will move the processing from one thread to another.

Serialization is the process of writing the state of an object to a byte stream. It can be used to save state variables or to communicate through network connections.

The Thread Class

The Thread Class allows multitasking (ie running several tasks at the same time) by instantiating (ie creating) many threaded objects, each with their own run time characteristics. Tasks that slow the processor can be isolated to prevent apparent loss of GUI response. One way to create threads is to extend the Thread class and override the run() method such as:

```
class HelloThread extends Thread
{
  public void run()
  {
    for int x=0;x<100; ++x)
      System.out.print(" Hello ");
  }
}
```

However if you need to inherit from another class as well, you can implement a Runnable interface instead and write the required run() method.

```
class HelloThread implements Runnable
```

```
{
  Thread t = new Thread(this);
  t.start();
  public void run()
  {
    for int x=0;x<100; ++x)
      System.out.print(" Hello ");
  }
}
```

Thread object methods are used on instantiated thread objects to control the thread appropriately. These methods include currentThread(), getName(), getPriority(), isAlive(),join(), run(), setName(string), setPriority(int), sleep(longInt) and start().

Some older methods such as stop(), suspend() and resume() have been deprecated as they sometimes caused system instability or hangup! A better way of stopping a thread is making the run() method into a while loop based on a logical that can be set to false as in:

```
public void run()
{
  while (okToRun==true)
  {
  // do the run time stuff here
  }
}
```

Inner classes are used to set up multiple threads in a utility.

Assigning Priority

Priority is thread ranking. Some threads can either run for a longer timeslice or run more often (depending on the operating system). Peers (or equals) get the same time/number of runs. Higher priority threads interrupt lower priority ones. Priority is set from constants MIN_PRIORITY (currently 1) to MAX_PRIORITY (currently 10) using the setPriority(int) method. NORM_PRIORITY is the midrange value (currently 5). These are defined in the Thread class.

Synchronization

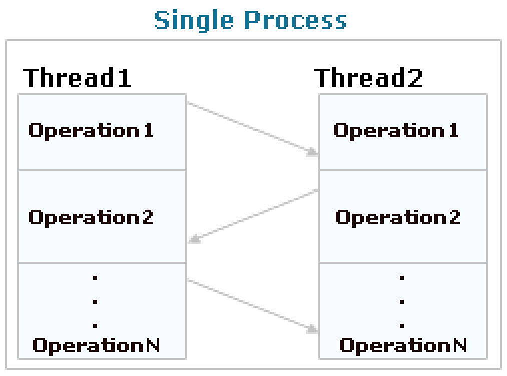
Thread synchronization is required when two or more threads need to share a resource. A monitor (aka semaphore) is an object that provides a mutually exclusive lock (mutex). Java provides the synchronized keyword as the key that locks/unlocks an object. It can be used as a class or method modifier or as a statement (very localized). Any long running method should not be synchonized as it would become a traffic bottleneck. To guarantee that a variable is threadsafe (ie. not shared between threads) it can be marked as volatile.

Multithreading:

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system

In the multithreading concept, several multiple lightweight processes are run in a single process/ task or program by a single processor. For Example, When you use a **word processor** you perform a many different tasks such as **printing, spell checking** and so on. Multithreaded software treats each process as a separate program.

In Java, the Java Virtual Machine **(JVM)** allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.
For example, look at the diagram shown as:



In this diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

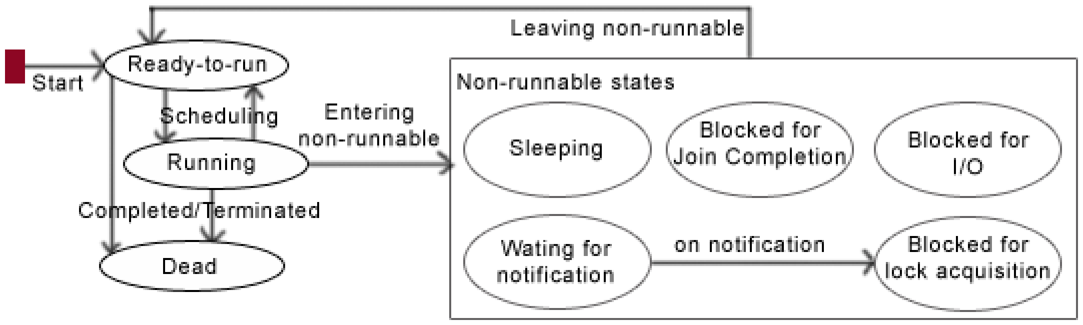
Advantages of multithreading over multitasking:

- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.
- Context switching between threads is usually less expensive than between

processes.

- Cost of communication between threads is relatively low.

Different states in multi- threads are-



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

- **Sleeping** – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method **sleep( )** to stop the running state of a thread.

  static void sleep(long millisecond) throws InterruptedException
- **Waiting for Notification** – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.
  final void wait(long timeout) throws InterruptedException
  final void wait(long timeout, int nanos) throws InterruptedException
  final void wait() throws InterruptedException

- **Blocked on I/O** – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.

- **Blocked for joint completion** – The thread can come on this state because of waiting the completion of another thread.

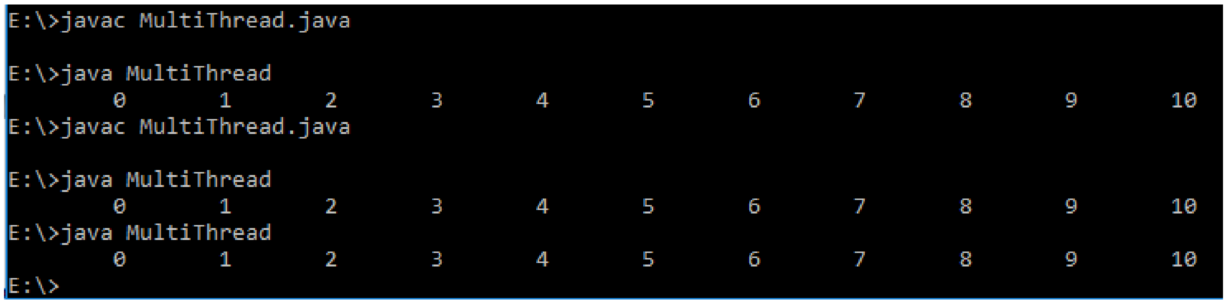
- **Blocked for lock acquisition** – The thread can come on this state because of waiting to acquire the lock of an object.

Program :

```
class ThreadEven extends Thread
{
 public void run()
 {
  for( ; MultiThread.counter<=10;)
  {
   if(MultiThread.counter%2==0)
   {
    System.out.print("\t"+MultiThread.counter);
    MultiThread.counter++;
   }
   else
   { Thread.yield(); }
  }
 }
}
class ThreadOdd extends Thread
{
 public void run()
 {
  for( ; MultiThread.counter<=10;)
  {
   if(MultiThread.counter%2!=0)
   {
    System.out.print("\t"+MultiThread.counter);
    MultiThread.counter++;
   }
   else
   { Thread.yield(); }
  }
 }
}
public class MultiThread
{
 static int counter =0;
 public static void main(String args[])
 {
  ThreadEven ET = new ThreadEven();
  ThreadOdd OT= new ThreadOdd();
  ET.start();
  OT.start();
 }
```

}

Output :



```
E:\>javac MultiThread.java

E:\>java MultiThread
        0       1       2       3       4       5       6       7       8       9       10
E:\>javac MultiThread.java

E:\>java MultiThread
        0       1       2       3       4       5       6       7       8       9       10
E:\>java MultiThread
        0       1       2       3       4       5       6       7       8       9       10
E:\>
```

Conclusion: Thus we studied how to do many work simultaneously.

Experiment No. 06

Title : Implement the concept of I/O Programming.

Aim : To Study

1. Byte Stream
2. Character stream
3. Buffered Stream

Input and Output from command line.

Objective: File IO is an important aspect of software development which can be used in storing inputs, outputs and intermediate results of the application on permanent storage devices in terms of FILEs.

Theory : I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

File I/O

- File Objects help you to write platform-independent code that examines and manipulates files.
- Random Access Files handle non-sequential file access.

Here are some basic points about I/O:
* Data in files on your system is called persistent data because it persists after the program runs.
* Files are created through streams in Java code.
* A stream is a linear, sequential flow of bytes of input or output data.
* Streams are written to the file system to create files.
* Streams can also be transferred over the Internet.

* Three streams are created for us automatically:
System.out - standard output stream
System.in - standard input stream
System.err - standard error

* Input/output on the local file system using applets is dependent on the browser's security manager. Typically, I/O is not done using applets. On the other hand, stand-alone applications have no security manager by default unless the developer has added that functionality.

Basic input and output classes
The java.io package contains a fairly large number of classes that deal with Java input and output. Most of the classes consist of:

- Byte streams that are subclasses of InputStream or OutputStream
- Character streams that are subclasses of Reader and Writer

The Reader and Writer classes read and write 16-bit Unicode characters. InputStream reads 8-bit bytes, while OutputStream writes 8-bit bytes. As their class name suggests, ObjectInputStream and ObjectOutputStream transmit entire objects.
ObjectInputStream reads objects; ObjectOutputStream writes objects.
Unicode is an international standard character encoding that is capable of representing most of the world's written languages. In Unicode, two bytes make a character.
Using the 16-bit Unicode character streams makes it easier to internationalize your code. As a result, the software is not dependent on one single encoding.

What to use?
There are a number of different questions to consider when dealing with the java.iopackage:
* What is your format: text or binary?
* Do you want random access capability?
* Are you dealing with objects or non-objects?
* What are your sources and sinks for data?
* Do you need to use filtering?

Text or binary
What's your format for storing or transmitting data? Will you be using text or binary data?
* If you use binary data, such as integers or doubles, then use the InputStream and OutputStream classes.
* If you are using text data, then the Reader and Writer classes are right.

Random access
Do you want random access to records? Random access allows you to go anywhere within a

file and be able to treat the file as if it were a collection of records.
The RandomAccessFile class permits random access. The data is stored in binary format.
Using random access files improves performance and efficiency.

java.io class overview
This section introduces the basic organization of the java.io classes, consisting of:
* Input and output streams
* Readers and writers
  * Data and object I/O streams

. **Files**
Introduction
This section examines the File class, an important non-stream class that represents a file or directory name in a system-independent way. The File class provides methods for:
* Listing directories
* Querying file attributes
* Renaming and deleting files

The File classes
The File class manipulates disk files and is used to construct FileInputStreams and FileOutputStreams. Some constructors for the File I/O classes take as a parameter an object of the File type. When we construct a File object, it represents that file on disk. When we call its methods, we manipulate the underlying disk file.
The methods for File objects are:
* Constructors
* Test methods
* Action methods
* List methods

Constructors
Constructors allow Java code to specify the initial values of an object. So when you're using constructors, initialization becomes part of the object creation step. Constructors for the File class are:
* File(String filename)
* File(String pathname, String filename)
* File(File directory, String filename)

Test Methods
Public methods in the File class perform tests on the specified file. For example:
* The exists() method asks if the file actually exists.
* The canRead() method asks if the file is readable.
* The canWrite() method asks if the file can be written to.
* The isFile() method asks if it is a file (as opposed to a directory).
* The isDirectory() method asks if it is a directory.

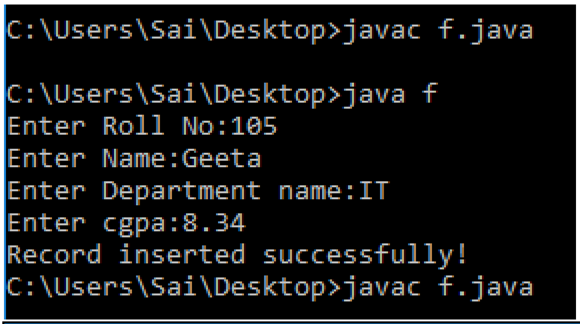
These methods are all of the boolean type, so they return a true or false.

Program : **INPUT**:

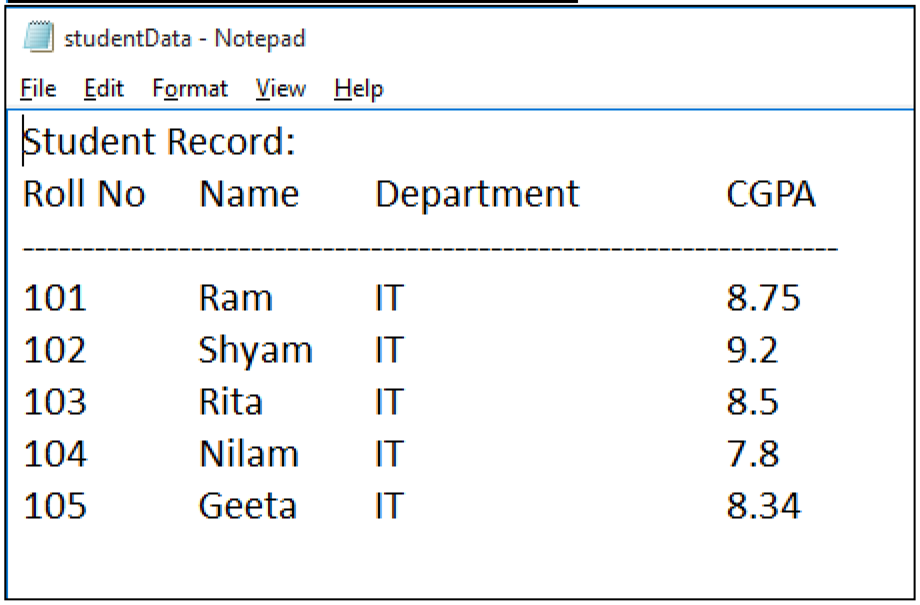
```
import java.io.*;
import java.util.*;

class student
{
 int RollNo;
 String name;
 String Dept;
 Double cgpa;
 student()
 {
  Scanner sc1=new Scanner(System.in);
  System.out.print("Enter Roll No:");
  RollNo=sc1.nextInt();
  System.out.print("Enter Name:");
  name=sc1.next();
  System.out.print("Enter Department name:");
  Dept=sc1.next();
  System.out.print("Enter cgpa:");
  cgpa=sc1.nextDouble();
 }
}
class f
{
 public static void main(String args[])throws Exception
 {
  Scanner sc=new Scanner(System.in);
  String s;
  File f1 = new File("studentData.txt");
  f1.createNewFile();
  OutputStream fos=new FileOutputStream(f1,true);
  PrintStream ps=new PrintStream(fos);

  student stu1=new student();

   ps.print(""+stu1.RollNo);
   ps.print("\t"+stu1.name);
   ps.print("\t"+stu1.Dept);
   ps.println("\t\t"+stu1.cgpa);
```

```
    System.out.print("Record inserted successfully!");
    fos.close();
    ps.close();
  }
 }
```

Output : **OUTPUT**:



```
C:\Users\Sai\Desktop>javac f.java

C:\Users\Sai\Desktop>java f
Enter Roll No:105
Enter Name:Geeta
Enter Department name:IT
Enter cgpa:8.34
Record inserted successfully!
C:\Users\Sai\Desktop>javac f.java
```



```
Student Record:
Roll No    Name    Department        CGPA
-----------------------------------------------------------------
101        Ram     IT                8.75
102        Shyam   IT                9.2
103        Rita    IT                8.5
104        Nilam   IT                7.8
105        Geeta   IT                8.34
```

Conclusion: Thus we studied different Input and output stream for byte stream and character stream.

Experiment No. 07

Title : Implement program for Applet with AWT controls.

Aim : To Study creating web application using applet

Objective: Implement a program to understand the concept of Applet

Theory : An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run. An applet is typically embedded inside a web-page and runs in the context of the browser. An applet must be a subclass of the java.applet.Applet class, which provides the standard interface between the applet and the browser environment.

Swing provides a special subclass of Applet, called javax.swing.JApplet, which should be used for all applets that use Swing components to construct their GUIs.

By calling certain methods, a browser manages an applet life cycle, if an applet is loaded in a web page.

Life Cycle of an Applet: Basically, there are four methods in the Applet class on which any applet is built.

- **init**: This method is intended for whatever initialization is needed for your applet. It is called after the param attributes of the applet tag.
- **start**: This method is automatically called after init method. It is also called whenever user returns to the page containing the applet after visiting other pages.
- **stop**: This method is automatically called whenever the user moves away from the page containing applets. You can use this method to stop an animation.
- **destroy**: This method is only called when the browser shuts down normally.

Thus, the applet can be initialized once and only once, started and stopped one or more times in its life, and destroyed once and only once.

For more information on Life Cycle of an Applet, please refer to The Life Cycle of an Applet section.

When to write Applets vs. Applications

In the early days of Java, one of the critical advantages that Java applets had over Java applications was that applets could be easily deployed over the web while Java applications required a more cumbersome installation process. Additionally, since applets are downloaded from the internet, by default they have to run in a restricted security environment, called the "sandbox", to ensure they don't perform any destructive operations on the user's computer, such as reading/writing to the filesystem.

However, the introduction of Java Web Starthas made it possible for Java applications to also be easily deployed over the web, as well as run in a secure environment. This means that the predominant difference between a Java applet and a Java application is that an applet runs in the context of a web browser, being typically embedded within an html page, while a Java application runs standalone, outside the browser. Thus, applets are particularly well suited for providing functions in a web page which require more interactivity or animation than HTML can provide, such as a graphical game, complex editing, or interactive data visualization. The end user is able to access the functionality without leaving the browser.

Loading Applets in a Web Page

In order to load an applet in a web page, you must specify the applet class with appropriate applet tags. A simple example is below:

```
<applet code=AppletWorld.class width="200" height="200">
</applet>
```

For development and testing purposes, you can run your applet using the lightweight appletviewer application that comes with the JDK. For example, if AppletWorld.html is the html file name, then you run the command as

```
appletviewer AppletWorld.html
```

Once you know your applet runs within the appletviewer, it is important to test your applet running in a web browser by loading the applet's web page into the browser window. The browser can retrieve the class files either from the internet or from the local working directory used during development. If you make changes to your applet's code while it is loaded in the browser, then you must recompile the applet and press the "Shift + Reload" button in the browser to load the new version.

Program :

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome",150,150);
}
}
Myapplet.html

<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Conclusion: Thus we studied to create an applet.

Experiment No. 08

Title : Program for creation of Frame and Window using Swing.

Aim : To study how to create Window using Swing component

Objective: Implement a program to understand the concept of Window using Swing

Theory : About the JFC and Swing

JFC is short form of Java Foundation Classes, which encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications. It is defined as containing the features shown in the table below.

Features of the Java Foundation Classes

Feature	**Description**
Swing GUI Components	Includes everything from buttons to split panes to tables. Many comp printing, and drag and drop, to name a few of the supported features.
Pluggable Look-and-Feel Support	The look and feel of Swing applications is pluggable, allowing a example, the same program can use either the Java or the Windows lo Java platform supports the GTK+ look and feel, which makes hundr available to Swing programs. Many more look-and-feel packages are a
Accessibility API	Enables assistive technologies, such as screen readers and Braille disp the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, te and applets. Java 2D includes extensive APIs for generating and se printing devices.
Internationalization	Allows developers to build applications that can interact with us languages and cultural conventions. With the input method fram applications that accept text in languages that use thousands of Japanese, Chinese, or Korean.

This trail concentrates on the Swing components. We help you choose the appropriate components for your GUI, tell you how to use them, and give you the background information you need to use them effectively. We also discuss other JFC features as they apply to Swing components.

Which Swing Packages Should I Use?

The Swing API is powerful, flexible — and immense. The Swing API has 18 public packages:

javax.accessibility	javax.swing.plaf	javax.swing.text
javax.swing	javax.swing.plaf.basic	javax.swing.text.html
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html.parser
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.rtf
javax.swing.event	javax.swing.plaf.synth	javax.swing.tree
javax.swing.filechooser	javax.swing.table	javax.swing.undo

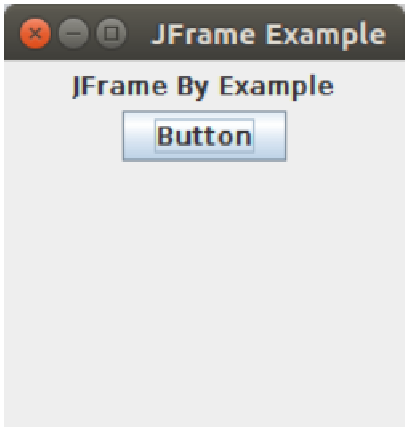
Fortunately, most programs use only a small subset of the API. This trail sorts out the API for you, giving you examples of common code and pointing you to methods and classes you're likely to need. Most of the code in this trail uses only one or two Swing packages:

- javax.swing
- javax.swing.event (not always required)

Program :

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JFrameExample {
    public static void main(String s[]) {
        JFrame frame = new JFrame("JFrame Example");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());
        JLabel label = new JLabel("JFrame By Example");
        JButton button = new JButton();
        button.setText("Button");
        panel.add(label);
        panel.add(button);
        frame.add(panel);
        frame.setSize(200, 300);
        frame.setLocationRelativeTo(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Output :



Conclusion: Thus we studied JFC and Swing

Experiment No. 09

Title : Program to demonstrate Event handling concept.

Aim : To study Graphical user interface design in Java

Objective: Event-based applications are highly used in GUI, which is essential part of the application development

Theory : Java, while only a few years old, is already being deployed in a wide variety of devices. Java exists for mainframes, midrange servers, PCs, and handheld devices. There is even a group currently working on a real-time Java for embedded control applications! Aside from the embedded environment, the vast majority of Java developers will be required to create some type of user interface for their application. For things like configuration tools, a command-line interface works fine (which Java supports, of course). However, for applications deployed to a wide range of users or for applications required to display data, graphical user interfaces work best.

Java supports GUI development through the Abstract Windowing Toolkit, or AWT. The AWT is the Java equivalent of the Microsoft Windows Common Control Library or a Motif widget toolkit. It includes support for simple graphics programming as well as a number of preconstructed components such as button, menu, list, and checkbox classes. The java.awt package is included with the Java 2 SDK and will be the focus of this discussion. Numerous third-party components are available (see links at bottom of this page) for additional GUI functionality.

Java AWT Basics

The AWT allows Java developers to quickly build Java applets and applications using a group of prebuilt user interface components. A number of Java IDE's are available which support the creation of user interfaces using the AWT by dragging-and-dropping components off a toolbar. It should be noted that these IDE's actually generate Java code on the fly based on what you do in the graphical design environment. This is in contrast to toolsets such as Microsoft Visual Basic which separate user interface design from the application code. The advantage of the Java approach is that you can edit your GUI either through a graphical IDE or by simply modifying the Java code and recompiling.

The three steps common to all Java GUI applications are:

1. Creation of a container
2. Layout of GUI components

3. Handling of events.

Creating A Container

This container object is actually derived from the java.awt.Container class and is one of (or inherited from) three primary classes: java.awt.Window, java.awt.Panel, java.awt.ScrollPane. The Window class represents a standalone window (either an application window in the form of a java.awt.Frame, or a dialog box in the form of a java.awt.Dialog).

The java.awt.Panel class is not a standalone window in and of itself; instead, it acts as a background container for all other components on a form. For instance, the java.awt.Applet class is a direct descendant of java.awt.Panel.

Laying Out GUI Components

GUI components can be arranged on a container using one of two methods. The first method involves the exact positioning (by pixel) of components on the container. The second method involves the use of what Java calls Layout Managers. If you think about it, virtually all GUIs arrange components based on the row-column metaphor. In other words, buttons, text boxes, and lists generally aren't located at random all over a form. Instead, they are usually neatly arranged in rows or columns with an OK or Cancel button arranged at the bottom or on the side. Layout Managers allow you to quickly add components to the manager and then have it arrange them automatically. The AWT provides six layout managers for your use:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.CardLayout
- java.awt.GridLayout
- java.awt.GridBagLayout
- java.awt.BoxLayout

The Container class contains the setLayout() method so that you can set the default LayoutManager to be used by your GUI. To actually add components to the container, we use the container's add() method:

```
Panel p = new java.awt.Panel();
Button b = new java.awt.Button("OK");
p.add(b);
```

Event Handling in Java 2

- Events are processed by listener objects which are registered by the GUI elements.
- In order to be used as a listener, a class must implement a listener interface.
- When an event occurs, the object on which the event occurred (source) sends the event to all registered listeners interested in that event by invoking the appropriate method on the listener object. An EventObject object is passed as an argument to the method. The Listener object can then take action based on that event or can ignore it.
- You do not have to specify separate listener classes -- the Applet itself can implement its own listeners.
- Or the button subclass could implement the listener interface
- There are several types of listener objects - each handles actions from several GUI element types. From Java Examples in a Nutshell:

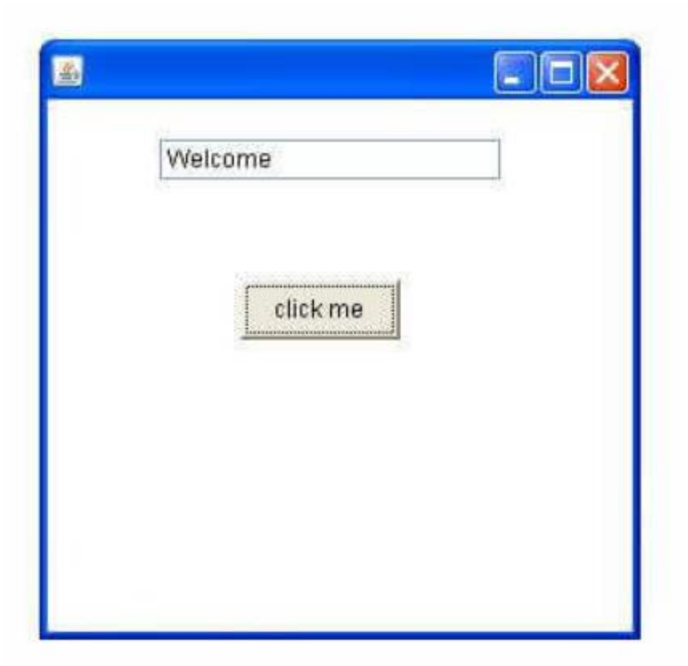
component	**Events generated**	**Meaning**
Button	ActionEvent	User clicked on button
Checkbox	ItemEvent	User selected or deselected item
CheckboxMenuItem	ItemEvent	User selected or deselected item
Choice	ItemEvent	User selected or deselected item
Component	ComponentEvent	Component moved hidden ,resized

- For a class to be a listener, it must implement one or more of the Listener interfaces: ActionListener, AdjustmentListener, ComponentListener,FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, TextListener, WindowListener. See here for the functions which are associated with each.
- A listener must implement all the methods in the listener interface. However, they can have empty bodies ({}) if not of interest.
- As listed above, the EventObject has subclasses (such as MouseEvent) that have their own methods. So, for example, a MouseEvent has getX() and getY() to get the coordinates of the mouse event (relative to the source component)).

Program :

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

Output :



Conclusion: Thus we studied how to handle event generated from AWT controls.

Experiment No. 10

Title : Implement a Client-Server Network programming.

Aim : To implement the concept of Client-Server Network programming.

Objective: Study the concept of Client-Server Network programming.

Theory : Java Networking

Network access is crucial to computer operations in the twenty first century. Java provides many built-in networking class objects through its .net, .nio and .rmi packages. java.net provides http connections and streams as well as protocol sockets. java.nio, java.nio.charset and java.nio.channels provide buffers, character sets and channels for multiplexed non-blocking applications that are more tolerant of dropped connections and time delays. java.rmi provides methods for remote method invocation.

Networking Basics

Internet addresses uniquely identify each computer on the network. They use a 4 byte dot IP notation such as 220.210.34.7 Domain Name Service allows us to use an easier to remember naming scheme. DNS servers translate the domain name such as amazon.com into its dot IP address.

Servers are any computer with resources (such as printers and disks) to be shared. Clients are entities that want to use these resources. Servers listen to their socket ports waiting for a client to connect with a service request. Servers are multithreaded to permit multiple services and multiple connects to the same service simultaneously. Proxy servers speak the client side of the protocol to another server. It acts as an agent of the client and can be set up to filter or cache data for it.

A network socket is a standard interface that uses the TCP/IP protocol to interconnect with a network. Sockets use IP address extensions or ports to connect to specific services such as FTP and telnet. Each port number (below 1024) has a reserved use.

Internet Streams

Internet streams allow you to access remote document data. The java.net package provides several objects for networking at the stream level:

- URL - builds a valid internet address. Methods are: getFile(), getHost(), getPort(), getProtocol(), openConnection() and toExternalForm()
- URLConnection - is a general purpose connection. Methods are: connect(), getContent(), getContentLength(), getContentType(), getDate(), getExpiration(), getInputStream() and getLastModified()
- HttpURLConnection - is an Http protocol specific connection. Methods in addition to those inherited from URLConnection are: getHeaderField(), getHeaderFieldDate(), getHeaderFieldKey(), getRequestMethod(), setRequestMethod(), [unfinished]

GetFile.java (found in jp7net.zip) demonstrates how to create a utility that displays both the header and the contents of an internet document. Seven main steps are involved:

1. Create an URL object that represents the resource's WWW address.
2. Create a HttpURLConnection object that opens a connection for the URL.
3. Use the getContent() method to create an InputStreamReader.
4. Use the InputStreamReader to create a BufferedReader object.
5. Use the getHeaderFieldKey(idx) and getHeaderField(idx) methods to retrieve header information.
6. Read the contents from the BufferedReader stream.
7. Write the contents to a Swing GUI textbox.

Warning: Both the access and display methods do not work for media files! If you are fetching image or audio files, use the technique from the packager project.

Socket Programming

TCP/IP sockets can be used to access protocols other than http. The java.net package provides several objects for networking at the TCP/IP socket level:

- Socket - TCP/IP client socket for telnet, smtp, nntp, whois, finger etc.
- ServerSocket - TCP/IP socket for server-side applications.

The following programs duplicate existing client utilities but demonstrate how to write TCP/IP applications.

Whois.java (found in jp7net.zip) accesses a directory service that provides information about a specific host. It makes a socket connection to internic.net (the registrar for commercial sites), sets a timeout and establishes a stream to access the data. Once compiled, test Whois with **java Whois amazon.com**.

Finger.java (found in jp7net.zip) accesses a directory service that provides information about a particular user based on his .plan and .project files. It makes a socket connection, sets a timeout and establishes a stream to access the data. Unfortunately most hosts have removed their finger software because of address harvesting. One remaining site for testing is hlr@well.com (aka hwl@well.sf.ca.us).

TimeServer.java (found in jp7net.zip) is a ServerSocket application that places the current time on its port 4415. Any client can access it. To test the application start the server with **java TimeServer**. A window should open with the message TimServer running... Do not close the window. On XP machines use the RUN dialog **telnet localhost 4415**. On older platforms start telnet in another window. At the connect dialog enter localhost in the Host Name field and 4415 in the Port field

Remote Method Invocation (RMI)

Remote Method Invocation allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This permits the creation of distributed applications. The java.rmi and java.rmi.server packages provide the required objects to handle remote method invocation. RMI is illustrated with a simple client/server application built in steps.

Step One: Compile Source Code

Note: Source code files for this example have been packed in jp7net.zip. Unpack them to a rmiProject directory.

AddServerIntf.java defines the remote interface that is provided by the server. It contains a method that takes two double arguments and returns a double sum. All remote interfaces must extend the Remote interface. Remote has no members but can throw a RemoteException.

AddServerImpl.java implements the remote interface. All remote objects must extend the UnicastRemoteObject.

AddServer.java contains the main program for the server. Its primary function is to update the RMI registry.

AddClient.java implements the client side of the application. It requires three command line arguments (IP address/name of the server and the two numbers to be added. The application forms an rmi URL.

Step Two: Generate Stubs and Skeleton

A stub is a Java object on the client machine which matches the interface on the server. A skeleton resides on the server. Java 2 does not require stubs. To generate stubs and skeletons, use the **rmic AddServerImpl** command to generate files called AddServerImpl_Skel.class and AddServerImpl_Stub.class. Be sure to have CLASSPATH pointed at the rmiProject directory.

Step Three: Install Files on Client and Server

Install the class files on their respective client or server machine. For testing purposes this may be the same one!

Step Four: Start the RMI Registry on the Server

First make sure that CLASSPATH includes the directory where you placed the files in step three. Then issue the **start rmiregistry** command. Note that a new window has been created. It must remain open until testing is complete.

Step Five: Start the Server

Issue the **java AddServer** command

Step Six: Start the Client

On your client machine issue the **java AddClient server1 8 9** command. If you are testing with the server on the same machine as the client use **java AddClient 127.0.0.1 8 9**. The address 127.0.0.1 is the loopback address for any self-test connection

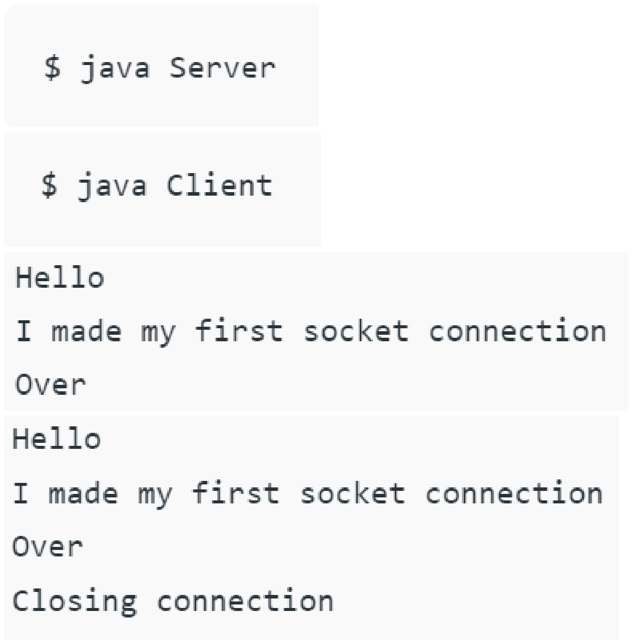
Program :

```
// A Java program for a Client
import java.net.*;
import java.io.*;

public class Client
{
        // initialize socket and input output streams
        private Socket socket            = null;
        private DataInputStream input = null;
        private DataOutputStream out          = null;

        // constructor to put ip address and port
        public Client(String address, int port)
        {
```

```
// establish a connection
try
{
        socket = new Socket(address, port);
        System.out.println("Connected");

        // takes input from terminal
        input = new DataInputStream(System.in);

        // sends output to the socket
        out = new DataOutputStream(socket.getOutputStream());
}
catch(UnknownHostException u)
{
        System.out.println(u);
}
catch(IOException i)
{
        System.out.println(i);
}

// string to read message from input
String line = "";

// keep reading until "Over" is input
while (!line.equals("Over"))
{
        try
        {
                line = input.readLine();
                out.writeUTF(line);
        }
        catch(IOException i)
        {
                System.out.println(i);
        }
}

// close the connection
try
{
        input.close();
        out.close();
        socket.close();
}
catch(IOException i)
```

```
                {
                        System.out.println(i);
                }
        }

        public static void main(String args[])
        {
                Client client = new Client("127.0.0.1", 5000);
        }
}
```

Output :



```
$ java Server
```

```
$ java Client
```

```
Hello
I made my first socket connection
Over
```

```
Hello
I made my first socket connection
Over
Closing connection
```

Conclusion: Thus we studied how to use the networking concept for socket programming and RMI using java.

Experiment No. 11

Title : Implement Program to demonstrate various methods of collection class

Aim : To Study the Collection class and its methods.

Objective: Implement a program to understand the concept of Collection class and its methods.

Theory :

Collections in Java :

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.
All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.
Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (Array List, Vector, Linked List, Priority Queue, Hash Set, Linked Hash Set, Tree Set etc).

What is Collection in java

Collection represents a single unit of objects i.e. a group.

What is framework in java

o provides readymade architecture.

o represents set of classes and interface.

o is optional.

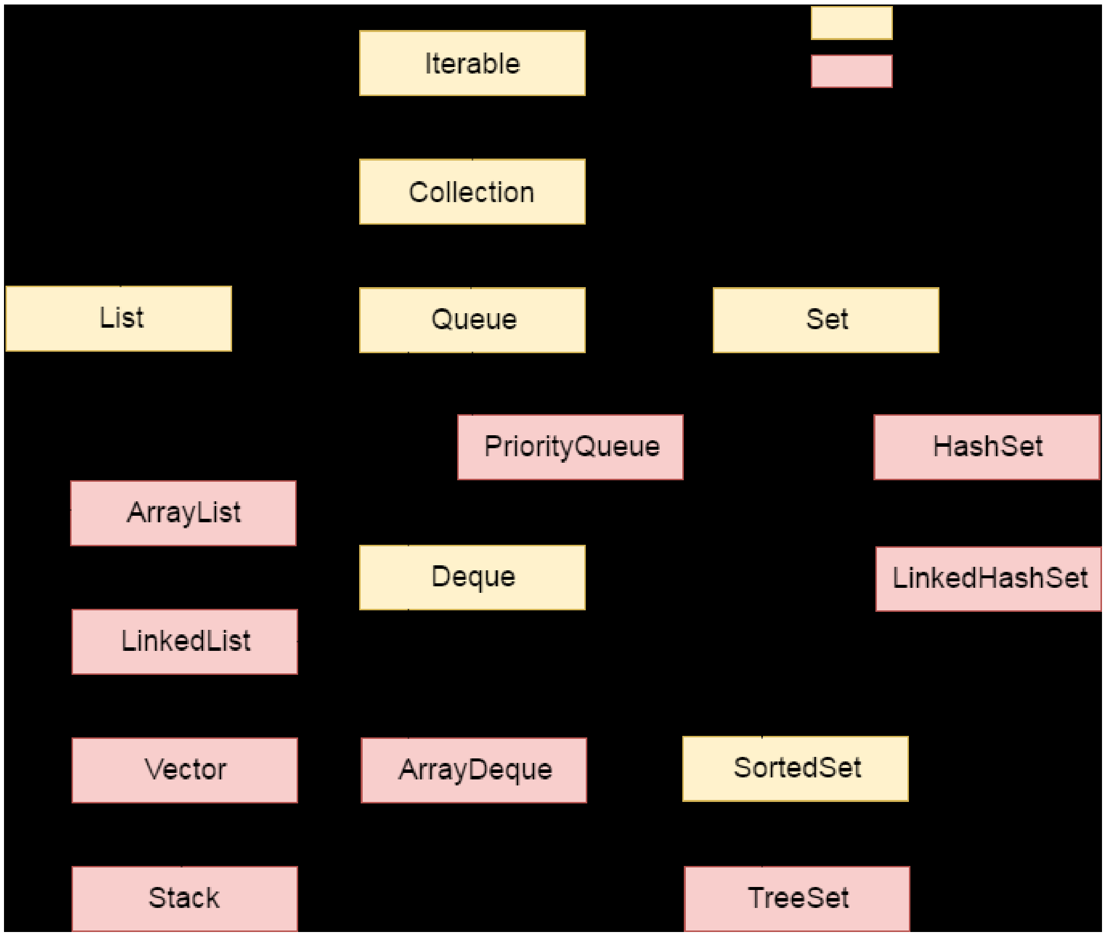
What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:
1. Interfaces and its implementations i.e. classes

2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

o Java ArrayList class can contain duplicate elements.

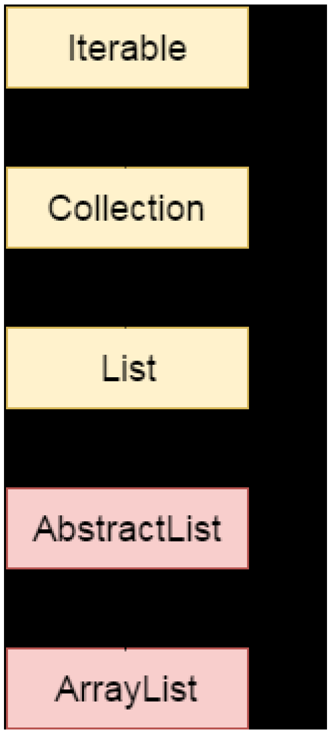
o Java ArrayList class maintains insertion order.

o Java ArrayList class is non synchronized.

o Java ArrayList allows random access because array works at the index basis.

o In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList class



Methods of Java ArrayList

 Void add()- It is used to insert the specified element at a specified position index in a list.

 Void clear() –It is used to remove all elements of the list.

Two ways to iterate the elements of collection in java

There are two ways to traverse collection elements:

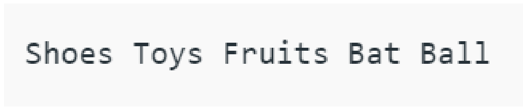
1. By Iterator interface.

2. By for-each loop.

Program :

```
// Java Program to Demonstrate Adding Elements
// Using addAll() method

// Importing required classes
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Main class
class GFG {

        // Main driver method
        public static void main(String[] args)
        {
                // Creating a list
                // Declaring object of string type
                List<String> items = new ArrayList<>();

                // Adding elements (items) to the list
                items.add("Shoes");
                items.add("Toys");

                // Add one or more elements
                Collections.addAll(items, "Fruits", "Bat", "Ball");

                // Printing the list contents
                for (int i = 0; i < items.size(); i++) {
                        System.out.print(items.get(i) + " ");
                }
        }
}
```

Output :



```
Shoes Toys Fruits Bat Ball
```

Conclusion: Thus we studied the Collection class and its various methods.

Experiment No. 12

Title : Implementing the concept of Database Programming

Aim : To study JDBC and ODBC connectivity in Java.

Objective: Database plays an important role in various applications. JDBC helps in communicate with any database through a java program which is an essential feature for any project development environment.

Theory :

JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

```
Connection con = DriverManager.getConnection
("jdbc:myDriver:wombat", "myLogin","myPassword");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
   int x = rs.getInt("a");
   String s = rs.getString("b");
   float f = rs.getFloat("c");
}
```

This short code fragment instantiates a DriverManager object to connect to a database driver and log into the database, instantiates a Statement object that carries your SQL language query to the database; instantiates a ResultSet object that retrieves the results of your query, and executes a simple while loop, which retrieves and displays those results. It's that simple.

JDBC Product Components

JDBC includes four components:

1. The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the Java™ Standard Edition (Java™ SE) and the Java™ Enterprise Edition (Java™ EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager —**

   The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

   The Standard Extension packages javax.naming and javax.sql let you use a DataSource object registered with a Java Naming and Directory Interface™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a DataSource object is recommended whenever possible.
3. **JDBC Test Suite —**

   The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
4. **JDBC-ODBC Bridge —**

   The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

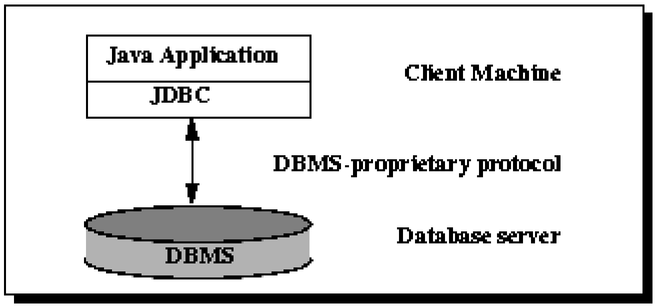
This Trail uses the first two of these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

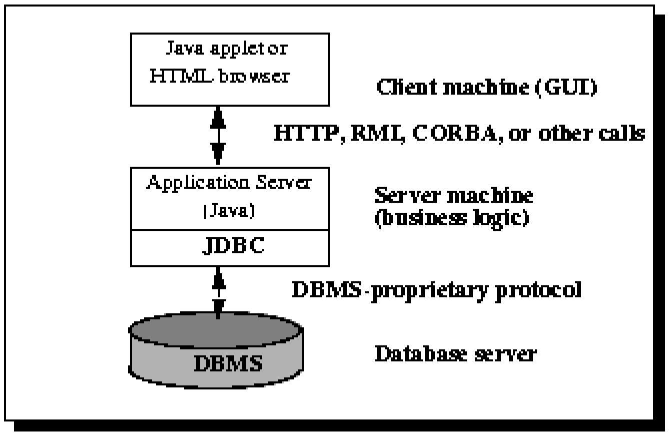
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.


Java applet or HTML browser
Client machine (GUI)
HTTP, RMI, CORBA, or other calls
Application Server (Java)
JDBC
Server machine (business logic)
DBMS-proprietary protocol
DBMS
Database server

Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

Table 1.2 illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Table 1.2: Employees

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Taylor gets a job at this company and the primary key is

First_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that Table 1.2 assumes that there can be only one car per employee.

SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME         LAST_NAME
----------        -----------
Axel               Washington
Florence            Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that SELECT * means "SELECT all columns."

```
SELECT *
FROM Employees
```

WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of WHERE clause involves a join, which is explained in the next section.

Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, shown in Table 1.3.

Table 1.3. Cars

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car_Number) in Employees would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the FROM clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
        Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

```
FIRST_NAME   LAST_NAME     MAKE       MODEL     YEAR
-----------  ------------  --------  ---------  -------
Axel         Washington    Honda     CivicDX    1996
Florence     Wojokowski    Toyota    Corolla    1999
```

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

- SELECT — used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
- INSERT — adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- DELETE — removes a specified row or set of rows from a table
- UPDATE — changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- CREATE TABLE — creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- DROP TABLE — deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as

specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

- ALTER TABLE — adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

```
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo)
                                        throws SQLException {
        Connection con = null;
        PreparedStatement pstmt = null;

        try {
          con = DriverManager.getConnection("jdbc:default:connection");

            pstmt = con.prepareStatement(
                        "UPDATE EMPLOYEES SET CAR_NUMBER = ? " +
                        "WHERE EMPLOYEE_NUMBER = ?");
            pstmt.setInt(1, carNo);
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
        }
        finally {
           if (pstmt != null) pstmt.close();
        }
    }
}
```

Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Establishing a Connection

First, you need to establish a connection with the DBMS you want to use. Typically, a JDBC™ application connects to a target data source using one of two mechanisms:

- DriverManager: This fully implemented class requires an application to load a specific driver, using a hardcoded URL. As part of its initialization, the DriverManager class attempts to load the driver classes referenced in the jdbc.drivers system property. This allows you to customize the JDBC Drivers used by your applications.
- DataSource: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. A DataSource object's properties are set so that it represents a particular data source.

Establishing a connection involves two steps: Loading the driver, and making the connection.

Loading the Driver

Loading the driver you want to use is very simple. It involves just one line of code in your program. To use the Java DB driver, add the following line of code:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

Your driver documentation provides the class name to use. In the example above, EmbeddedDriver is one of the drivers for Java DB.

Calling the Class.forName automatically creates an instance of a driver and registers it with the DriverManager, so you don't need to create an instance of the class. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

After you have loaded a driver, it can make a connection with a DBMS.

Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS.

Using the DriverManager Class

The DriverManager class works with the Driver interface to manage the set of drivers available to a JDBC client. When the client requests a connection and provides a URL, the DriverManager is responsible for finding a driver that recognizes the URL and for using it to connect to the corresponding data source. Connection URLs have the following form:

```
jdbc:derby:<dbName>[propertyList]
```

The dbName portion of the URL identifies a specific database. A database can be in one of many locations: in the current working directory, on the classpath, in a JAR file, in a specific Java DB database home directory, or in an absolute location on your file system.

If you are using a vendor-specific driver, such as Oracle, the documentation will tell you what subprotocol to use, that is, what to put after jdbc: in the JDBC URL. For example, if the driver developer has registered the name OracleDriver as the subprotocol, the first and second parts of the JDBC URL will be jdbc.driver.OracleDriver . The driver documentation will also give you guidelines for the rest of the JDBC URL. This last part of the JDBC URL supplies information for identifying the data source.

The getConnection method establishes a connection:

```
Connection conn = DriverManager.getConnection("jdbc:derby:COFFEES");
```

In place of " myLogin " you insert the name you use to log in to the DBMS; in place of " myPassword " you insert your password for the DBMS. So, if you log in to your DBMS with a login name of " Fernanda " and a password of " J8, " just these two lines of code will establish a connection:

```
String url = "jdbc:derby:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

If one of the drivers you loaded recognizes the JDBC URL supplied to the method DriverManager.getConnection, that driver establishes a connection to the DBMS specified in the JDBC URL. The DriverManager class, true to its name, manages all of the details of establishing the connection for you behind the scenes. Unless you are writing a driver, you probably won't use any of the methods in the interface Driver, and the only DriverManager method you really need to know is DriverManager.getConnection

The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS. In the previous example, con is an open connection, and you use it in the examples that follow.

Using a DataSource Object for a connection

Using a DataSource object increases application portability by making it possible for an application to use a logical name for a data source instead of having to supply information specific to a particular driver. The following example shows how to use a DataSource to establish a connection:

You can configure a DataSource using a tool or manually. For example, Here is an example of a DataSource lookup:

```
 InitialContext ic = new InitialContext()

 DataSource ds = ic.lookup("java:comp/env/jdbc/myDB");
 Connection con = ds.getConnection();
 DataSource ds = (DataSource) org.apache.derby.jdbc.ClientDataSource()
 ds.setPort(1527);
 ds.setHost("localhost");
 ds.setUser("APP")
 ds.setPassword("APP");

Connection con = ds.getConnection();
```

DataSource implementations must provide getter and setter methods for each property they support. These properties typically are initialized when the DataSource object is deployed.

```
VendorDataSource vds = new VendorDataSource();
```

```
vds.setServerName("my_database_server");
String name = vds.getServerName();
```

JDBC-ODBC Bridge Driver

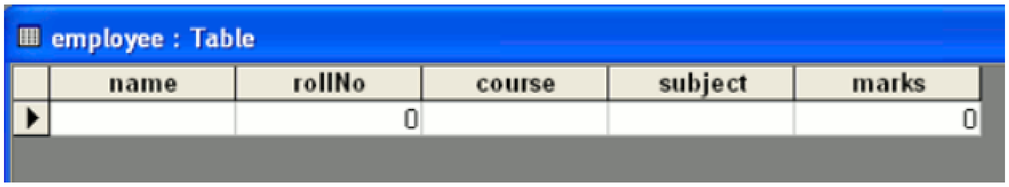
For normal use, you should obtain a commercial JDBC driver from a vendor such as your database vendor or your database middleware vendor. The JDBC-ODBC Bridge driver provided with JDBC is recommended only for development and testing, or when no other alternative is available.

Program :

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JdbcOdbcExample {

        public static void main(String args[])
        {
        Connection con = null;
        PreparedStatement stmt = null;
        try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        con = DriverManager.getConnection("jdbc:odbc:swing");
        String sql ="INSERT INTO employee (name, rollNo, course,

subject, marks) VALUES" +
                        "('Deepak', 10, 'MCA', 'Computer Science',

85)";
        stmt = con.prepareStatement(sql);
        int i = stmt.executeUpdate();
        if(i > 0 )
        {
                System.out.println("Record Added Into Table

Successfully.");
        }
        } catch (SQLException sqle) {
                System.out.println(sqle.getNextException());
        } catch (ClassNotFoundException e) {
                System.out.println(e.getException());
        } finally {
        try {
        if (stmt != null) {
        stmt.close();
        stmt = null;
```

```
            }
            if (con != null) {
            con.close();
            con = null;
            }
            } catch (Exception e) {
            System.out.println(e);
            }
            }
            }
    }
```

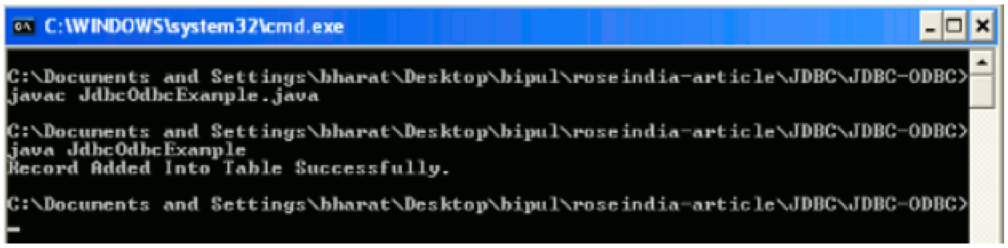
Output : Table Before Inserting the record into it.



employee : Table

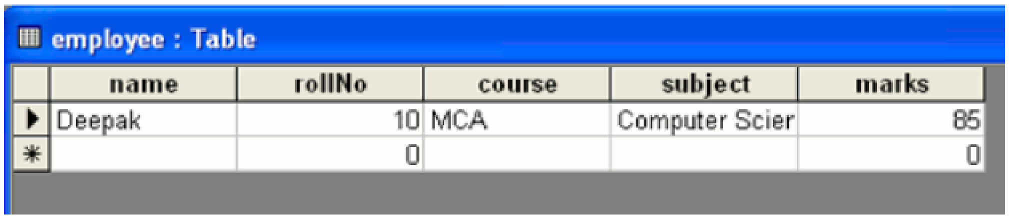
	name	rollNo	course	subject	marks
▶		0			0

When you will execute the above example you will get the output at console as follows :



```
C:\Documents and Settings\bharat\Desktop\bipul\roseindia-article\JDBC\JDBC-ODBC>
javac JdbcOdbcExample.java

C:\Documents and Settings\bharat\Desktop\bipul\roseindia-article\JDBC\JDBC-ODBC>
java JdbcOdbcExample
Record Added Into Table Successfully.

C:\Documents and Settings\bharat\Desktop\bipul\roseindia-article\JDBC\JDBC-ODBC>
_
```

And then the Table after inserting record will be seen as follows :



employee : Table

	name	rollNo	course	subject	marks
▶	Deepak	10	MCA	Computer Scier	85
✳		0			0

Conclusion: Thus we studied Database Connectivity in java.