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 :

Output :

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 :

Output :

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

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.

Program-
Output-

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

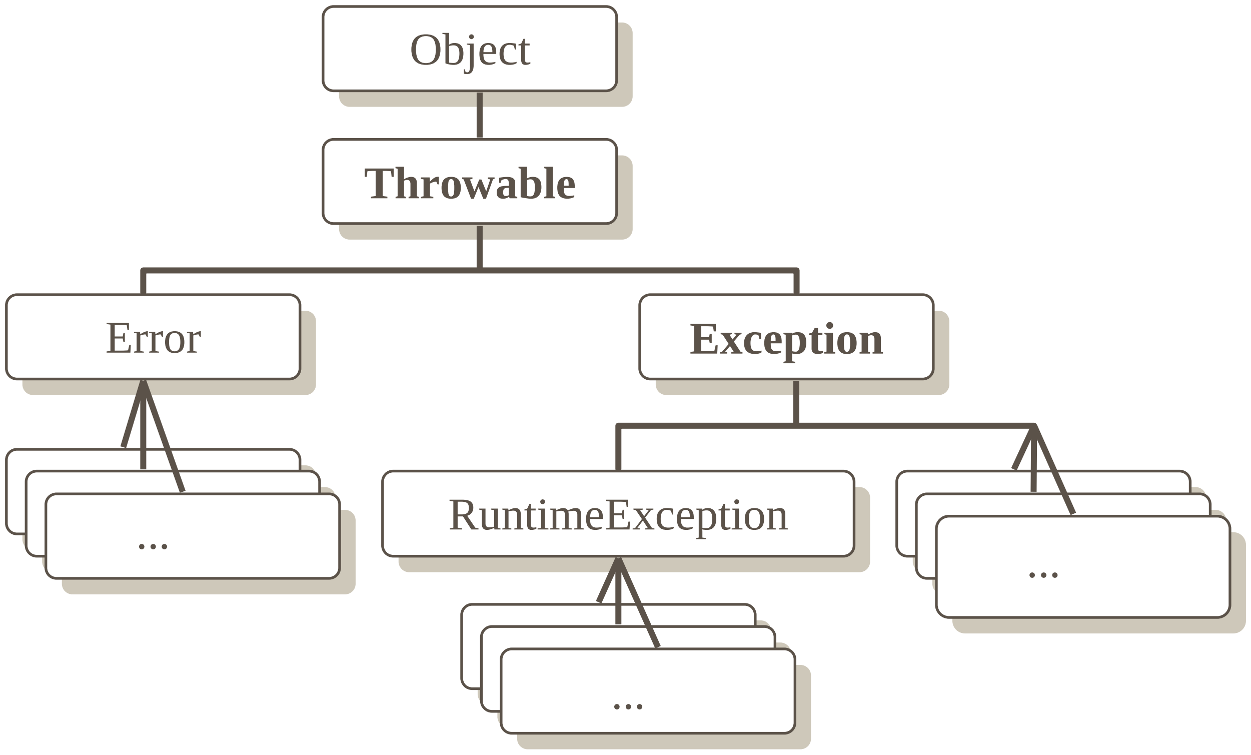
```
try{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb){
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb){
        // exception handler for ExceptionType2
}
//…
finally{
        // block of code to be executed before try block ends
}
```

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 :

Output :

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

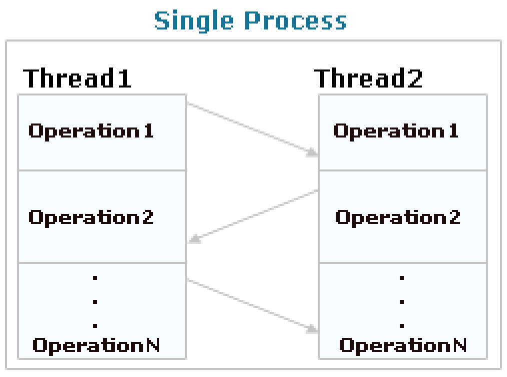
Thread synchronization is required when two or more threads need to share a resource. A monitor (aka semaphore) is an object that provides a mutually exclusive lock (mutex). Java provides the synchronized keyword as the key that locks/unlocks an object. It can be used as a class or method modifier or 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:

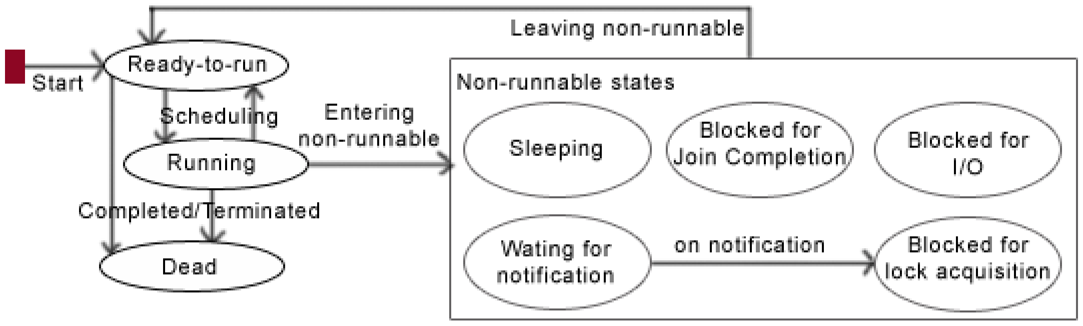


In this diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

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 :

Output :

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 :
Output :

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 :

Output

Conclusion: Thus we studied to create an applet.