

→ COURSE CONTENTS

Basics - Encapsulations

A simple java program - compilation and execution.

Object - definition and creation

Primitive data types

Type casting - necessity and usage

Functions - passing parameters, return data types e.t.c.

static keyword - meaning and purpose

Interpreting System.out.println statement.

Polymorphism - static and dynamic polymorphism.

Method overloading and overriding.

Constructors - 'this' and 'super' keywords.

String class & StringBuffer class

Arrays - declaration and usage.

Command-line arguments

Wrapper classes

Inheritance - Types, usage, the keyword 'final'

Superclass reference - sub-class object rule.

Interfaces - late binding.

Abstract classes

Packages

Access specifiers - public, private, protected & default

Inner classes

static and non-static blocks

Exception Handling - try, catch, throw, throws, finally keywords and their usage., userdefined exceptions.

Multithreading - All important concepts.

I/O streams - classification and usage.

files

Networking

Abstract Window Toolkit (AWT)

Event handling.

Applets

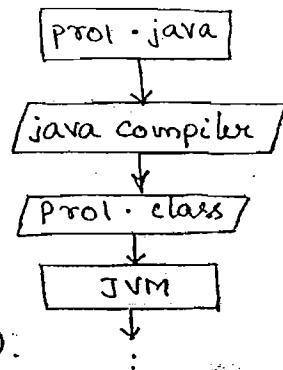
Util Package - collections framework , maps and all the utility classes.

(* Includes interview questions in each topic *)

26.06.

Execution of a Java Program

Static loading: A block of code would be loaded into the RAM before it is executed i.e. after being loaded into the RAM, it may or may not get executed).



Dynamic loading: A block of code would be loaded into the RAM only when it is required to be executed.

NOTE: static loading takes place in the execution of structured programming languages. Ex: C-language.

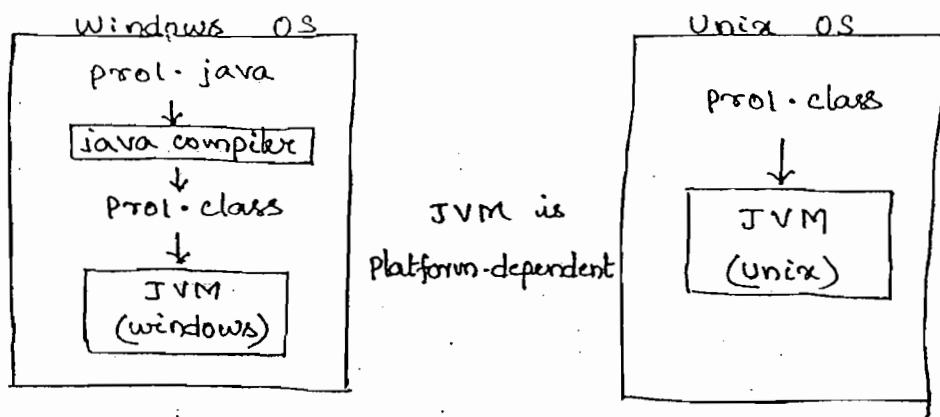
Java follows Dynamic loading.

- JVM would not convert all the statements of the .class file into its executable code at a time.
- Once the control comes out from the method, then it is deleted from the RAM and another method of .exe type will be loaded as required.
- Once the control comes out from the main(), the main() method would also be deleted from the RAM. This is why we are not able to view the .exe contents of a .class file.

27.06.2006

Functions of JVM:

- (i) It converts the required part of the byte code into its equivalent executable code.
- (ii) It loads the executable code into the RAM
- (iii) Executes this code through local operating system.
- (iv) Deletes the executable code from the RAM.



we know that JVM converts .class file into its equivalent executable code. Now, if a JVM is in windows environment it converts the .class file into its equivalent executable code that is understood by windows environment only.

Similarly, same is the case with Unix or other OS.

Thus JVM is platform dependent.

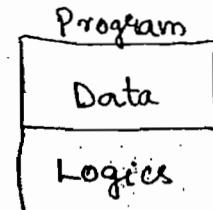
But the platform dependency of the JVM is not considered while saying Java is platform independent, because JVM is supplied free of cost through the internet by the Sun Micro Systems. (Hotels - tiffen - water : example)

platform Independency:

Compiled code of a program should be executed in any operating system, irrespective of the OS in which that code had been generated. This concept is known as platform independency.

- The birth of OOPS concept took place with encapsulation.
- Any program contains two parts :

Data part and Logic part



- out of data and logics the highest priority is given to data.

But in structured programming languages the data insecurity is high.

Thus in a process of securing data in structured prog. lang. the concept of encapsulation came into existence.

25.06.06

Note: In structured programming lang. programs, the global variables play a vital role.

But, because of these global variables, there is data insecurity in the structured programming lang. programs.

i.e. functions that are not related to some variables will have access to those variables and thus data may get corrupted. In this way data is unsecured.

"This is what people say in general about data insecurity

But this is not the actual reason. The actual concept is as follows."

Let us assume that, we have a 'c' program with two hundred functions. Assume that it is a project. Now if any upgradation is required, then the client i.e. the user of this program (s/w) comes to the IT company and asks the programmers to update it according to his requirement. Now we should note that, it is not guaranteed that, the programmers who developed this program will still be working with that company. Hence this project falls into the hands of new programmers.

(P.T.O)

Automatically, it takes a lot of time to study the project itself before upgrading it. It may not be surprising that the time required for writing the code to update upgrade the project may be very less when compared to the time required for studying the project.

Thus maintenance becomes a problem.

If the new programmer adds a new function to the existing code in the way of upgrading it, there is no guarantee that it will not affect the existing functions in the code. This is because of global variables.

In this way data insecurity is created.

To overcome this problem, programmers developed the concept of encapsulation.

For example, let us have a program with ten global variables and twenty functions.

It is sure that all the twenty functions will not use all the global variables.

Three of the global variables may be used only by two functions. But in a structured prog. lang. like 'C' it is not possible to restrict the access of ~~var~~ global variables by some limited no. of functions.

Every function will have access to all the global variables.

To avoid this problem, programmers have designed a way such that the variables and ^{the} functions which are associated or operate on those variables are enclosed in a block, and that block is called a class.

and that class is given a name, just as a function is given a name.

Now the variables inside the block cannot be called as local variables because they are not local to only one function and they cannot be called as global variables because they are confined to a block and not global.

Hence these variables are known as instance variables.

Ex:

struct. prog. lang. program.

```
#include<stdio.h>
int i,j,k,l,m,n;
fun1()
{
}
fun2()
{
}
:
fun10()
{
}
main()
{
}
```

Ex:

OOP. lang. program.

```
class Abc
{
    int i,j,k;
    fun1()
    {
    }
    fun2()
    {
    }
    :
    class XYZ
    {
        int l,m,n;
        fun3()
        {
        }
        fun4()
        {
    }
```

This block is called a class with name Abc.

Therefore a class is nothing but grouping data along with its functionalities.

NOTE 1: Encapsulation is the concept of binding data along with its corresponding functionalities.

Encapsulation came into existence in order to provide security for the data present inside the program.

NOTE 2: Any object oriented programming language file looks like a group of classes. Everything is encapsulated. Nothing is outside the class.

- Encapsulation is the backbone of oop languages.
- JAVA supports all the oop concepts (i.e. encapsulation, polymorphism, Inheritance) and hence it is known as object oriented programming language.
- C++ breaks the concept of encapsulation, because the main() method in a C++ program is declared outside a class. Hence it is not a pure OOP language. In fact, it is a poor oop language.

29-06-2006

A simple java program:

Rules and Regulations:

- The class name should start with a capital letter. This is not compulsory but it is a conventional rule.
- Java is case sensitive.
- All keywords in java are written in small letters

- As a convention, anything that starts with a capital letter is treated as the name of a class in java.
- Following is a simple program that prints "Hello world".

```
class Hello  
{ public static void main(String args[])  
{  
    System.out.println("Hello World");  
}  
}
```

In the above program, Hello is the class name. we have the main method which has only one statement in it.

NOTE: No method or function should be written in a program without mentioning its return data type.

Even in this program, main is a method and it returns nothing. Hence the keyword void is present before it. void indicates that the main() method does not return anything.

In the above program public, static, void are keywords. main() is a method. String is a class.

println is a method and System is a class.

Let us save the program in the name Hello.java.

Now, we compile the program with the following command in the dos prompt.

```
D:\> javac Hello.java
```

- After the execution of this command, the .class file i.e. Hello.class is created.
- In other words, it is the DOS command given to the Java compiler to convert contents of the mentioned source into its equivalent byte code.
 - Now the following command is given to the JVM to execute the .class file:

D:\> java Hello.

- On the process of executing the .class file one of the responsibilities of the JVM is to convert the .class file into its equivalent executable code.

NOTE: We always use the Editplus editor for developing Java programs. The advantage of Editplus editor is that, all the keywords are displayed in blue colour. All the predefined classes are displayed in red colour. Other part of the program is displayed in black. These colour differences help in identifying some of the syntactical errors / grammatical errors.

Program:

```
class A
{
    int x=10;
    public void main(String args[])
    {
        int y=20;
        y=y+1;
        x=x+1;
    }
}
```

D:\> javac A.java

A.class

D:\> java A

y

As soon as the program is compiled the A.class file is created and after the "D:\> java A" command is given, then the necessary part i.e. main() method of the .class file is converted into its equivalent executable code and is loaded into RAM from the harddisk and made available for to the CPU for execution.

Now, till $y = y + 1$, we do not have any problem. But when the control comes to the statement $x = x + 1$; it generates a runtime error because 'x' is not a variable which is declared in main. But we should note that `int x = 10;` is declared inside the class but presently it is not available to the CPU as it is not in the RAM.

Now, by using some procedure, if we are able to transfer the contents of harddisk to the RAM, then x will be present in the RAM and it will be available to the main and thereby to the CPU for execution.

The procedure of loading the contents of harddisk to the RAM dynamically at runtime is done by creating an object. (This is the need for creating an object).

(P.T.O)

30.06.2006

Till now we have seen the need for the creation of an object. Now let us discuss how the object is created.

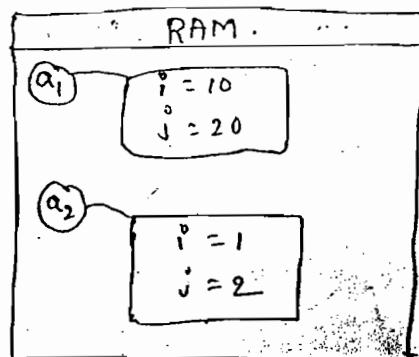
- This is done using the 'new' operator. 'new' is a keyword in Java.
- An important responsibility of the new operator is to transfer the contents of .class file from the harddisk into the RAM.

NOTE: In a given code, all the elements that have the keyword 'static' before them are known as static elements and others are known as non-static elements.

Ex: class A

```

{ int i=1, j=2;
  public static void main(String args[])
  {
    A a1 = new A();
    a1.i = 10;
    a1.j = 20;
    A a2 = new A();
    a2.i = 1;
    a2.j = 2;
  }
}
  
```



consider the above program. whenever this program is compiled and executed, first the main() method is transferred from the harddisk to the RAM.

The first statement inside the main method is

A a₁ = new A();

As soon as the keyword 'new' is encountered by the JVM it understands that it has to load all the non-static elements of the .class file from harddisk to the RAM.

Inside the RAM some memory space would be allocated for these non-static elements and the address of this memory location would be assigned to the variable a₁, where, a₁ is a variable of type class A.

For ex: when we have a statement a₁.i, then it gets access to the content of a₁ which is nothing.. but the address of the memory location where the variable 'i' is stored. Now it can get access to the variable 'i' in that memory location whose address is in a₁.

Ex: a₁.i = 10 ; a₁.j = 20 e.t.c.

Definition: The memory in the RAM which is reserved for the non-static elements of the .class file is known as OBJECT.

- The concept of reserving memory on the RAM (i.e. at run time) for the contents of harddisk is known as instance.
- Note that, in case of object, the memory is reserved for the contents of .class file (in fact they are also contents of harddisk) in the RAM.
- Hence object is called as instance of class.
- Now we can say that a_1 is a variable which contains the address of the object.
- a_1 is known as handle (or) pointer of type A.

NOTE: When we write $a_1.i=10$; $a_1.j=20$, then the values of $a_1.i$ and $a_1.j$ change to 10 & 20 respectively only in the RAM, but the values of i and j are not affected in the .class file i.e. A.class which is in the harddisk.

NOTE: Since a_1 points to an object of class A, many people call a_1 as an object. But strictly speaking a_1 is not an object, it is the reference to an object (pointing finger \rightarrow TV; Example).

- Once the control comes out from the main(), then the main method is deleted from the RAM. Since variables a_1 and a_2 are created in main, automatically they will also be deleted when main is deleted.

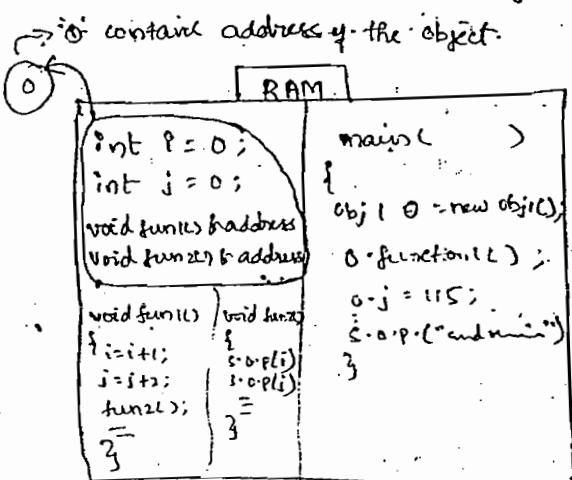
so, when a_1 and a_2 are deleted, the addresses to the objects pointed out by a_1 and a_2 are also lost. Now these objects, which have no variable to point them, are treated as garbage. In java, it is the duty of the garbage collector to clear this garbage and free the memory in the RAM.

01.07.2006

```

class obj1
{
    int i=0, j=0;
    void function1()
    {
        i = i+1;
        j = j+1;
        function2();
        System.out.println("Inside function1()");
    }
    void function2()
    {
        System.out.println(i);
        System.out.println(j);
        System.out.println("Inside function2()");
    }
}
public static void main(String args[])
{
    obj1 o = new obj1();
    o.function1();
    o.j = 115;
    System.out.println("End of main");
} //end main
} // end class

```



on the above program, as soon as it is compiled and executed, first the main method is loaded into the RAM. Inside the main method, when the statement `obj2 o = new Obj1();` is encountered, the JVM loads all the non-static elements of the class `Obj1` into the RAM from the harddisk and allocates a memory for these non-static elements and then assigns the address of this memory to the variable `o`. Hence the object of class `Obj1` is said to be created.

The non-static members are `int i=0, j=0;` and `void fun1() { i = 3; }` and `void fun2() { j = 3; }`. But here we have to note that JVM loads only the signatures and addresses of `fun1` and `fun2` into the RAM, instead of loading the entire body of the function.

NOTE: we have to keep in mind that, JVM follows the concept of dynamic loading.

Thus, when `fun1()` and `fun2()` are specifically called, then the body of those functions will be loaded to the RAM from the harddisk and once when the control comes out of the function, the body of the function will be deleted from the RAM. But, the address of the function (i.e. the address of its location in the harddisk) will be present in the object.

The above discussion, is illustrated in the previous page, inside the box, by the side of the program.

NOTE-1: Suppose if $i = i + 1$ is a statement in `fun1()`, then the control finds out whether `i` is defined in `fun1()` or not. In this case, since it is not defined in `fun1()`, the control goes to the object which is available to `function1()` using the address of the object present in the variable `O`, and then searches for `i`. Now, it finds `i` here. Suppose, if it does not find the variable `i` declared here, then it generates a run time error.

- From this, it is clear that, in order to execute `fun1()`, we need to create an object of the class which holds `fun1()`.
- Hence, without creating an object of a class, we can not execute the functions of that class.

NOTE-2: Signature and address of a function is also data. Hence object is said to contain only data and nothing other than that.

NOTE-3: The data present inside the object is always tentative as and when we execute any functionalities acting on that data.

State of the object: The data present inside the object at that instant of time is called state of the object.

Behaviour of the object:

The functionalities that are applicable to the object is known as behaviour of the object.

variables i and j would be transferred to the ... instance of the class (object) i.e. they are accommodated inside the instance. and hence they are known as instance variables.

NOTE: Any non-static variable, will be accommodated inside the instance and hence it is known as instance variable.

Local variables are those that are defined inside a function and their scope lies only within that function.

Ex:

```
class obj2
{
    int i=0, j=0;
}

void func()
{
    int i=0;
    i=25;
    j=35;
}
```

```
s.o.p("Inside func");
s.o.p(i); s.o.p(j);
s.o.p("End of func");
}
```

```
public static void main(String args[])
{
    obj2 o = new obj2();
    o.func();
    s.o.p(o.i);
    s.o.p(o.j);
}
```

obj2.java

D:\> javac obj2.java

D:\> java obj2

Inside func()

25 → i

35 → j

End of func()

0 → i

35 → j

Interpretation of the o/p of the above program is as follows:

As the program is compiled and executed, object O will be created and after that fun() is called using object O. Inside fun() we have int i=0;
 $i=25$, $j=35$.

Here, i is declared once again, so any value of i will be local to this function. So now $i=25$.

The next statement $j=35$, but j is not a variable of fun(), infact it is a class variable. So $j=35$ both inside the function and outside the function.

So after the execution of s.o.p statements we have $i=25$ and $j=35$ and then control returns to main.

Now, in main, the next statement is

s.o.p(O.i) & s.o.p(O.j).

Now i and j refer to the class variables.

Now the value of i is 0 and j is 35 because i was redeclared in fun() and any change in the value of i is confined only to fun(). Once the control comes out of it, again the value of i will be 0. But j is affected as it is accessed through the object.

Hence, the value of i is 0 & j is 35.

NOTE: Local variables have the highest priority at the time of execution. i.e. if the control encounters a variable assignment, then it checks whether that variable is defined inside that function or not. If yes, it is a local variable and it has its priority. If no, then the control goes to the object on which the function is acting to find the variable defined. Even if the variable is not defined inside the object, then it generates a runtime error.

→ // comment lines: This is literally asking the compiler not to convert that statement (having the comment lines) into its equivalent executable code.

03-07-2006

compile the following two programs within the same path.

x.java

```
class X
{
    int i=0;
    void funX()
    {
        i = i+1;
        i = i*i;
    }
    public static void main(String args[])
    {
        X x1 = new X();
        x1.i = 2;
        x1.funX();
        System.out.println(x1.i);
    }
}
```

obj2.java

```
class obj2
{
    int j=1;
    void fun1()
    {
        j = j+1;
        X x1 = new X();
        x1.i = 15;
        System.out.println(x1.i);
    }
}
public class obj2
{
    obj2 o = new obj2();
    o.fun1();
    X x1 = new X();
    System.out.println(x1.i); // here x1.i=0.
}
```

Before discussing in detail about the above two programs, once again we have to recall that whenever the 'new' operator is encountered, immediately, JVM loads all the non-static members of the class into the RAM and allocates some memory and the address of that object is assigned to a variable of that class type.

The above two programs get executed perfectly. If we are keen, we can observe that object of X.class file is created inside Obj2.class file. Here, one precaution that we should take is that the path name of execution of the two programs should be same.

Now, we can conclude that, object of any .class file can be created at anywhere (i.e. inside any other program) provided the path should be same for both the programs.

→ Consider the following program:

```
class obj3
{
    int i=1, j=2;
    void fun1()
    {
        S.O.P(i);
        S.O.P(j);
    }
}
```

Let us save this program as obj3.java. Now when we compile this program, it gets compiled and even the .class file obj3.class will be created. If we try to execute this .class file, then it generates a runtime error, because it does not have a main method.

NOTE: Every .java file (or .class file) need not ~~have~~ a main method.

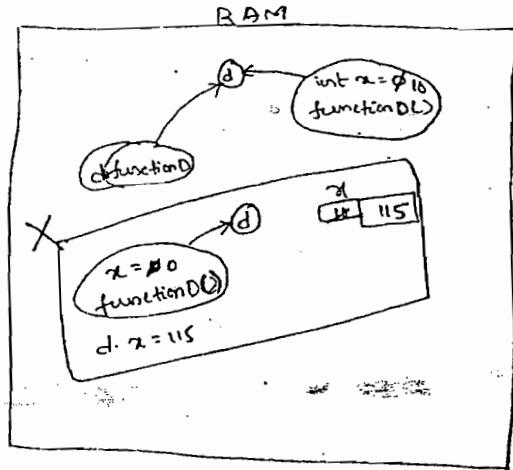
- The main method is required inside a .class file only to execute that particular .class file. This is because execution of a .class file starts from the main method that it contains.
- Another important point to be noted is that, there is no compulsion that every class should have some variables and functions (or) at least one function in it.
- A class can be completely empty without even a single statement in it.

Ex: class xyz
 {
 // no statements inside.
 }

Now, if we create an object of class xyz, then an object will be created. Since the class is empty, the object will also be empty. But still, remember, the object will have some properties. (These will be discussed later).

Ex:

```
class Demo1  
{    int x = 0;  
    void functionD()  
    {  
        x = 10;
```



```
    Demo1 d = new Demo1();  
    d.x = 115;  
    S.o.p(d.x); // here d.x = 115  
    S.o.p(x); // here x = 10  
    S.o.p("End of functionD()");  
}
```

```
P.S.V.main(String args[])
```

```
{  
    Demo1 d = new Demo1();  
    d.functionD();  
    S.o.p(d.x); // here x = 10  
}
```

}

In the above program, we get an output 10, for the `println` statement in the `main` method. This can be understood, if we clearly interpret the concept of objects once again. As illustrated roughly in the box above.

04.01

As soon as the Demol. class file comes for execution, the main method is loaded into the RAM, and it gets executed. The first statement in the main method creates an object of the Demol. class file and loads all the non-static members of the class (i.e. int $x=0$, signature and address of funD()) in to the object, and the address of the object is assigned to the variable d.

The second statement is $d.funD()$, which is a call to functionD(). Now the functionD() will be loaded to the RAM. The first statement inside the function is $x=10$. As the variable x is not defined inside the function, automatically the variable x inside the object on which the function is acting gets affected and now its value is $x=10$.

In the second statement inside the function, we are again creating an object of the Demol. class file. So once again all the non-static members of the class are loaded into the RAM and the object is created whose address is assigned to variable d.

The next statement $d.x = 115$ assigns the value 115 to the variable x inside the object that was just created. Now $s.o.p(d.x)$ prints 115. But $s.o.p(x)$ prints 10.

After this the control comes out from the function and reaches the main method. In this process, functionD() is deleted from RAM and all the local variables of funD() lose their life.

Now, in the main method, $s.o.p(d.x)$ prints 10, as that is the value of x inside the object.

04.07.06

consider the following program.

DemoC.java

```
class DemoA
{
    int i = 0;

    void functionA()
    {
        System.out.println("Inside funA() of class DemoA");
    }
}

class DemoB
{
    int j = 0;

    void functionB()
    {
        System.out.println("Inside funB() of class DemoB");
    }
}

class DemoC
{
    void func()
    {
        System.out.println("Inside func() of class DemoC");
    }
}

public static void main (String args[])
{
    DemoA da = new DemoA();
    DemoB db = new DemoB();

    da.functionA();
    db.functionB();
}
```

Saved the above program as DemoC.java
In the above program, we have three classes DemoA, DemoB and DemoC.

When we compile the programs using javac DemoC.java, the compiler converts all the contents of the source file into its equivalent byte code. So, when we compile the source file, its equivalent .class file is created.

Till now we have considered programs that had only single class along with the main method.

But in this program, we have three classes. Now what does the compiler do?

NOTE: The java compiler converts all the contents of source file into their equivalent byte code, class wise.

In the above program, the compiler generates three class files as there are three classes in the source file DemoC.java

D:\> javac DemoC.java
↓
DemoC.class, DemoA.class, DemoB.class.

NOTE (i) A .class file will always be an equivalent of only one class in the source file.

(ii) The name of the .class file would always be the name of the class that particular .class file is representing.

To be more clear, let us assume that we save the file Democ.java as pro1.java.

Now when we compile pro1.java, we do not get any pro1.class file and in fact we get the .class files according to the classes present in pro1.java.

Suppose, if we want to execute pro1.java, now we have to use any of the .class files (that contains main method) that have been generated after compiling pro1.java.

NOTE: If we save the source file with the name of the class that has the main method, then we can compile and execute the source file with the same name (as that of the class).

This is the reason why people prefer to save the source file with the name of the class.

- Actually we can save the file with any name we like. But to execute the .class file, we have to supply that .class file ^{to the JVM} which has the main method in it. (Hence the discussion in the NOTE above).

(P.T.O)

consider the following program.

Prog.java

```
class DemoX
{
    int i = 0;
    public static void main (String args[])
    {
        DemoY y = new DemoY ();
        System.out.println (y.i);
        y.functionY ();
    }
}

class DemoY
{
    int j = 10;
    void functionY ()
    {
        System.out.println ("Inside functionY ()");
    }
    public static void main (String args[])
    {
        DemoZ z = new DemoZ ();
        z.functionZ ();
    }
}

class DemoZ
{
    void functionZ ()
    {
        System.out.println ("Inside functionZ ()");
    }
    public static void main (String args[])
    {
        DemoX d = new DemoX ();
        System.out.println (d.i);
    }
}
```

In the program `pro2.java` we have three classes and each class has a `main()` method in it. Now the question is which `main()` method will be / should be executed.

Till now we have seen programs with some classes and only one `main()` method in one of those classes.

We know that, when we compile a `.java` file, the compiler creates `.class` files for all the classes in the `.java` file. But we are supposed to execute only one `.class` file at a time. ~~for~~ i.e. for example, in this program `pro2.java`,

D:\> `javac pro2.java` \leftarrow `Demox.class`, `Demoy.class` and `Demoz.class`.

Now D:\> `java Demox`. (or) D:\> `java Demoy` (or)

D:\> `java Demoz` is possible but .

X. D:\> `java Demox, Demoy, Demoz` is not possible.

Hence, at any instance of time JVM can execute only one `.class` file. It cannot execute two or more `.class` files simultaneously.

In this program (or) till now, we have seen programs with only one main method in one class.

NOTE: We can define more than one main method inside the same class with different arguments. i.e.

`p.s.v.m (String args[])`, `p.s.v.m (Int args[])` etc.

This is the concept of polymorphism to be discussed later.

consider the following programs Obj6.java

```
class Obj6
{
    int i = 1;
    void function()
    {
        i = i + 1;
        System.out.println(i);
        System.out.println("End of fun()");

    }
    public static void main (String args[])
    {
        Obj6 o1 = new Obj6();
        int x1 = 10;
        o1.i = 25;
        Obj6 o2 = new Obj6();
        System.out.println(o1.i); // i=1
        o1.i = 16;
        Obj6 o2 = o1;
        System.out.println(o2.i); // o2.i=16
    }
}
```

In the above program we have changed the address stored in the reference and also we have created two references to the same object.

Let us trace the program.

Let us assume that as soon as the main method is loaded into the RAM, the first statement in the main is for creation of object. So an object is created and its address is assigned to o1. Now we have `int x1 = 10;` so a variable x1 is created and it is assigned a value 10. Now we have `.o1.i = 25`. So, with this, the value of i inside the object changes to 25. Next we are creating another object for the same objs. class file and its address is assigned to o1 again. When this is done, the address which o1 held before is overwritten and now o1 points to the newly created object. The next statement is `s.o.p(o1.i)` and this obviously prints 1 since the value of i inside the new object is 1. After this we have `obj o2 = o1`. This means pointer variable o2 is assigned the address which o1 is holding. Now we have two references to o1 and o2 to the same object. This is supported by java.

NOTE: A single object can have multiple references. But one reference can always point only one object.

(P.T.O)

→ Data types in Java

Java supports 8 primitive data types under four categories:

(i) Integers

| | |
|-------|------------------------|
| byte | → 8 bits (or) 1 byte |
| short | → 16 bits (or) 2 bytes |
| int | → 32 bits (or) 4 bytes |
| long | → 64 bits (or) 8 bytes |

(ii) Real numbers

| | |
|--------|------------------------|
| float | → 32 bits (or) 4 bytes |
| double | → 64 bits (or) 8 bytes |

(iii) character — char → 16 bits (or) 2 bytes.

(iv) Boolean values — Boolean → 1 bit (true or false)

true/false are known as Tuples.

NOTE: Tuples are predefined constants which represent only one value.

05-07-2006

we create objects of a class to call the functions of that particular class.

Sometimes we need to call a function of a class only once in the entire program. In such cases if we create the object and assign the address of that object to a variable, this leads to unnecessary wastage of memory. Because the variable holding the address of the object will occupy some memory and the object itself occupies some memory, even though during

the entire execution of the program even though its use is limited.

So, to avoid this unnecessary wastage of memory, we create the object, call the function and after this we delete the object immediately. This allows for better memory management.

consider the following program:-

```
class DemoA
{
    void functionA()
    {
        System.out.println("Inside functionA()");
    }
}
public static void main (String args[])
{
    new A().functionA(); → ④
}
```

④ → when this statement is executed an object of class DemoA is created and the function inside the object is executed. once the control comes out from this object (function) it is deleted from the RAM (as its address is not maintained) and this object is collected by the garbage collector.

In a similar way, we can access a variable.

Ex: int i = new D().a;

NOTE: whenever we need to refer to an object more than once, then only we assign the address of the object to a variable. otherwise we follow the above process.

consider the following program:-

class DemoB

DemoB.java

```
{ byte b; short s; int i; long l;  
float f; double d; char c; boolean flag;  
public static void main (String args[]){  
    DemoB d = new DemoB();  
    s.o.p(d.b); s.o.p(d.s);  
    s.o.p(d.i); s.o.p(d.l);  
    s.o.p(d.f); s.o.p(d.d);  
    s.o.p(d.c); s.o.p(d.flag);  
}  
}
```

when we compile and execute this program we get the following output.

b → 0 s → 0 i → 0 l → 0
f → 0.0 d → 0.0 c → flag → false.

These values are the default values.

NOTE: After loading all the non-static elements into the object, then the JVM initializes all the uninitialized variables with their value default values.

Default values of variables of primitive data types are shown in the o/p of the above program.

NOTE (i) The default value of char is ' ' (space).

(ii) character has to be represented with '' (quotes)

(iii) Atmost we can assign only one character to a char variable.

Ex: char c = 's', a = 'a' c.t.c.

Another example program:-

```
class DemoB
{
    byte b; short s; etc.
    int j;

    public static void main (String args[])
    {
        DemoB d = new DemoB();
        int j;
        s = o.p(i); // Here we get an error.
        j = 10;
        s = o.p(j); // Here we do not get any error.
                      because already j is initialized to 10.
    }
}
```

In the above program, we have a variable *j* of type integer inside the main method.

If we compile this program, we get an error as those shown in comment lines. This is because of the following reason.

Reason: Here *j* is a local variable. JVM will not initialize it to its default value since it is not an instance variable.

NOTE: Uninitialized local variables cannot be used in java.

So, if we want to use a variable as a local variable in the program first it must be initialized to some value before using it (or) operating on it, in the program.

- When we assign any non-decimal number to a variable, the JVM accepts it as an integer (32 bits)
- Any number with a decimal point, would be accepted by the JVM as double (64 bits)

Type casting: The procedure of converting one datatype into its equivalent another datatype is known as type-casting. (Ex: 500 Rs \rightarrow 1 single note; 500 one rupee coins)

Type casting

- Implicit type casting
- Explicit type casting.

- Implicit type casting is storing a smaller value into its equivalent bigger location.
- Explicit type casting is storing a bigger value into its equivalent smaller location.

NOTE: The above two definitions are not always true.
There are some violations.

Consider the following program:-

```
class DemoC
{
    p. s. v. m( string args[])
    {
        int i = 6;
        float j = 7;
        s. o. p(j); // j = 7.0
    }
    float k = 7.0;
    // s. o. p(k); // Here we get a compilation error.
    float k = (float) 7.0; // explicit type casting.
    long l = 8; // implicit type casting.
```

~~O/P~~

7.0
8
8.0

NOTE

s.o.p(l);

int x = l;

byte b = 6; // Here we do not get any error. In fact implicit type casting takes place. (This type of implicit type casting is supported by only some versions of JVM.)

b = x; // error.

b = (byte)x; // In these cases explicit typecasting is highly recommended.

float c = 7.8f; // This is another method of typecasting.

char ch = (char)x; // This is possible.

s.o.p(ch);

char ch1 = '8';

s.o.p(ch1);

int z = (int)ch; // This is also possible.

s.o.p(z);

}

}

NOTE: We can typecast certain primitive data types into some certain primitive data types only.

Typecasting any primitive datatype into any other primitive datatype is not possible.

(Same is the concept with the objects of classes).

(P.T.O)

06.07.2009

consider the following program:-

A.java

```

class A
{
    int i, j;
    void functionA()
    {
        s.o.p(i);
        s.o.p(j);
    }
}
P.S.V. main (String args[])
{
    A a1 = new A();
    int x = 10;
    s.o.p(x);
    s.o.p(a1); // address of object will be displayed.
}

```

OP
10
A@cac268

consider the following program:-

Odemo1.java

```

class Odemo1
{
    int i = 10;
    A a1 = new A();
    A a2;
    // s.o.p(a2); NOTE: Executable statements should NOT be given as class instances.
}
P.S.V. main (String args[])
{
    Odemo1 d1 = new Odemo1();
    s.o.p(d1.a2);
    s.o.p(d1.i); // d1.i = 10
    int x = d1.i;
    s.o.p(d1.a1); // prints address of object of class A
}

```

OP
null
10
10
0
0
10
0

```
d1.i = 25;  
d1.a1.i = 10;  
d1.a1.functionA();  
System.out.println();
```

/* println() is a non-static method present in
out object. But note that System is a class.
In the statement d1.a1.functionA(), functionA()
is a non-static method present in a1 object of
class A; and d1 is also an object of a class */

```
s.o.p(d1.a2);
```

* d1.a2.functionA(); // we donot get a compilation error
but we get a runtime error.

```
// A a1;
```

```
A a1 = null;
```

a1.functionA(); */ // Here we get a compilation error
because a1 is a local variable
and JVM will not initialize it.

```
d1.a2 = d1.a1;
```

```
s.o.p(d1.a2.i); // d1.a2.i = 10
```

```
d1.j = d1.i;
```

```
d1.a2 = new A();
```

```
s.o.p(d1.a2.i); // d1.a2.i = 0
```

```
}
```

```
}
```

* Conclusion: we can define object of any class
as an instance variable of any class..

So we can have something as shown above
x1, y1, z1, function2() and this chain can be
continued.

Question: with what values will uninitialized reference
variables be initialized?

Ans: Default initialization of all uninitialized reference
variables is null.

NOTE: we can not initialize a variable with
zero to assign it null (nothing).

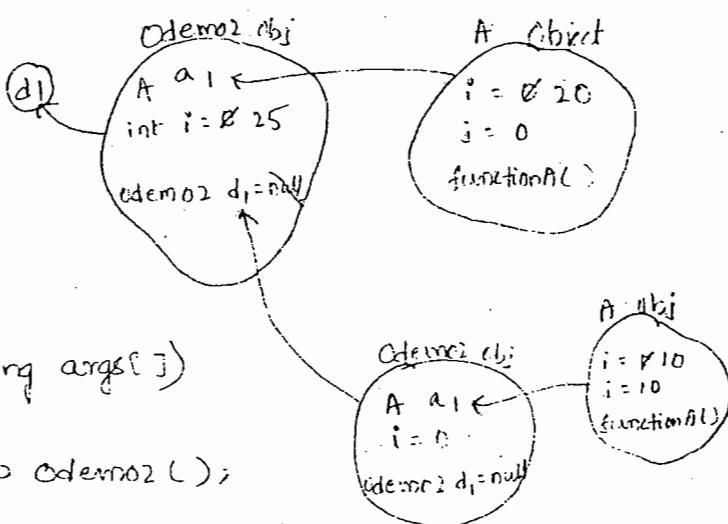
Consider the following program:-

Odemo2.java

```
class Odemo2
{
    A a1 = new A();
    int i;
    Odemo2 d1;

    P.S.V. main (String args[])
    {
        Odemo2 d1 = new Odemo2();
        d1.i = 25;
        S.O.P (d1.d1); // It prints null.

        d1.d1 = new Odemo2();
        d1.d1.a1.i = 10;
        d1.d1.i = 20;
        S.O.P (d1);
        S.O.P (d1.d1);
        S.O.P (d1.d1.d1); // Here we get null.
    }
}
```



~~o/p~~
null
odemo2@1a16e69
odemo2@1cdel00
null

NOTE

Conclusion: we can define object of a class as an instance in the same class, but it should not be initialized with the address of the object of the same class. This is illustrated in the following program.

```
class Odemo2
```

```
{ A a1 = new A();
```

```
Odemo2 d1 = new Odemo2();
```

```
int i;
```

```
main()
```

```
{ Odemo2 d1 = new Odemo2();
```

```
d1 = new Odemo2();
```

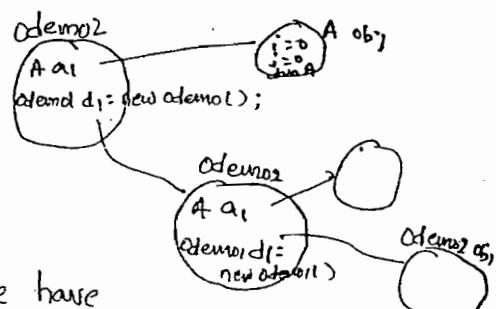
/* Instead of this statement we have

to just define but not initialize.

```
Odemo2 d1; */
```

NOTE: with this statement

it goes into an infinite loop creating objects of class Odemo2 as a chain.



Goes into infinite loop

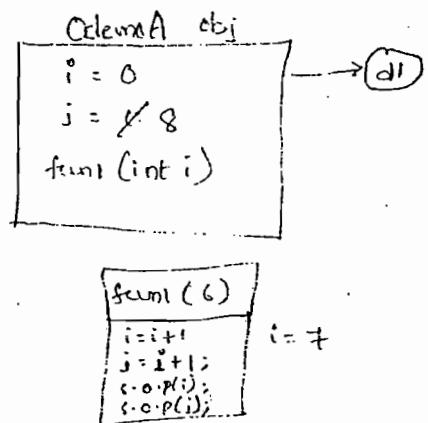
→ Passing arguments to a function:

NOTE: when a function is defined with parameters, then we should not / can not call that function without passing the corresponding arguments to the corresponding parameters of the function.

(P.T.O)

consider the following function program:- [OdemoA.java]

```
class OdemoA  
{  
    int i, j;  
  
    void function1(int i)  
    {  
        i = i + 1;  
        j = i + 1;  
        s.o.p(i); // i=7  
        s.o.p(j); // j=8  
    }  
}
```



```
class Test  
{  
    public static void main(String args[])  
    {  
        OdemoA d1 = new OdemoA();  
  
        d1.function1(6);  
  
        s.o.p(d1.i); // i=0  
  
        s.o.p(d1.j); // j=8  
    }  
}
```

Explanation: The value 6 is passed as an argument to function1(). There int i is the parameter and it is a local variable to the function1(). Now when the

statement `i = i + 1` is executed the value of `i` is changed to `7` only inside the function. This will not effect the instance variable `i` inside the object. This is because, as already mentioned, `i` is a local variable to `function1()` and remember local variables has have the highest priority.

- As we can define functions to accept primitive data types as arguments, we can define functions to accept object of a class as an argument.

This is illustrated in the following program.

```
class OdemoA
{
    int i, j;

    void function1(int i) // call by value
    {
        s.o.p(i);
        s.o.p("Inside function1()");
    }

    void function2(A a1) // call by reference/address
    {
        if (a1 != null) // usually this is the precaution
            taken, when we pass the object of
            a class as an argument to a
            function.
        {
            a1.i = 203;
            a1.j = 115;
        }
        s.o.p(a1);
        s.o.p("end of fun2()");
    }
}
```

```

public static void main( String args[] )
{
    CdemoA d1 = new CdemoA();
    A a2 = new A();
    d1.function2(a2);
    {
        d1.function2(a2new A());
    }
    // d1.function2(null); // we can pass null as an
                           // argument to a function.
    s.o.p(a2.i);
    s.o.p("End main");
}
}

```

K@15colea
 203
 end of function2
 end of main

Explanation: when the statement `d1.function2(a2);` is executed encountered, `function2(A a1)` will be executed.

The value of argument `a2` will be accepted by the parameter `a1` of `function2()`. variable `a1` is a local variable of `function2()` which holds the address of the object of class `A`.

once, the control comes out from the `function2()`, it will be deleted from the RAM and along with it variable `a1` is also lost. Since `a1` contains address of an object and when `a1` is lost, automatically the object becomes un-referred object and becomes garbage.

But in this case, even though variable a_1 is lost, the object is not lost because the object is still being pointed out by variable a_2 inside the main method, which is still existing in the RAM.

If we use `di.function2(new A())`, then in this case, as soon as the control comes out from the function, the variable is lost and along with it the object is lost, as the address of the object ~~is~~ only is assigned only to the variable a_1 , which is local to the `function2()`.

NOTE:

→ Defining functions with a return data type:

NOTE: Whenever we define a function with a return data type, then we should not close the function without returning the value of the return datatype. So a blind rule we follow is to declare a variable of the return datatype as the first statement inside the function and assign the value to be returned to that variable and then return that variable.

The following program illustrates the above concept.

control (P.T.O)

```
class OdemoB
{
    int i, j;

    int function1 (int i)
    {
        int j = 0;
        j = i * i;
        return (j);
    }
}

class Test
{
    public static void main (String args[])
    {
        OdemoB d1 = new OdemoB();

        d1.function1 (9);

        int x = d1.function1 (4);

        s.o.p (x); // x=16  

        (or)  

        s.o.p (d1.function1 (4)); // 16 is printed  

        s.o.p (x+6); // 22 is printed  

        (or)  

        s.o.p (d1.function1 (4) + 6); // 22 is printed

    } // end main()
}

} // end class.
```

Explanation :

In the statement `dl.function1(a)`, we are not handling the value of the return data type i.e. `dl.function1(a)` holds the value 81 and this value is not assigned to any variable in the main.

NOTE: In java, it is not mandatory to handle such values, but if not handled properly, ^{then} that will go to the garbage collector.

In the statement `int a = dl.function1(u)`, we are handling the value of the return data type i.e. the value returned by the function is assigned to the variable `a`.

The last statement in the program illustrates that we can even pass expressions as arguments to the functions.

**
* *

NOTE: As we can design functions to return a primitive datatype, we can design functions to return object of a class.

10.07.20

* consider the following program. In this program, the return datatype of a function is an object of a class*/

class Odemo2

{

 A getA()

{

 A a1 = null;
 a1 = new A();
 return (a1);

}

// Naming conventions are very important
as far as industry requirements are
concerned. Naming convention is
defining variables with names that
are user friendly or user understandable.//

public static void main (String args [])

{

 Odemo2 d = new Odemo2();

 A x = d.getA(); // Handling (= assigning) the return
 datatype of the method getA()
 to the variable x of type class A.
(or)

 d.getA(); // Return datatype is not handled.
 s.o.p(d.getA());
 s.o.p(x);
 x.funA();

(or)

 d.getA().funA(); // Since here we require object of class A
 only once, we do not handle the return
 datatype of method getA().

int i = d.getA(); // error - Because address of an object
 cannot be assigned to an integer variable.

int i = d.getA().j; // This is possible. Here i = j

s.o.p(i); // i=0, ∵ i is a non-static variable initialized
 by the JVM.

}

{ } .. .

o/p
K@15601ea
K@197d257
10
20

Question: what do you mean by the following statements?

(i) int i = a1.funA().funB().func();

The return datatype of func() is an integer and is assigned to the variable i.

func() is accessed using the object returned by funB() and funB() is accessed using the object returned by funA() and funA() is contained in a1 object.

By we can have a statement, as follows:

A bc x = a1.funA().funB().func();

(ii) a1.funA().b1.funB().c1.func();

Here funB() returns an object of some class where c1 is an instance variable of that class, but c1 is also an object of some other class which has func() in it.

(iii) int x = a1.funA() * a1.funC();

Not only * (multiplication) but any other arithmetic symbol can be used. This statement indicates that basic arithmetic operations can be performed on the ^{similar} _{return} datatypes of the functions and ^{the result} can be assigned to a variable of similar datatype.

Here variable x is of type integer. Obviously, the return data types of a1.funA() and a1.funC() should also be of type integer.

→ The static Keyword:

if the keyword 'static' is present before a variable/method then it is called a static variable/method.

NOTE: we can even define the main method without the keyword static.

/* The following program illustrates the use of static keyword */

class Sdemol

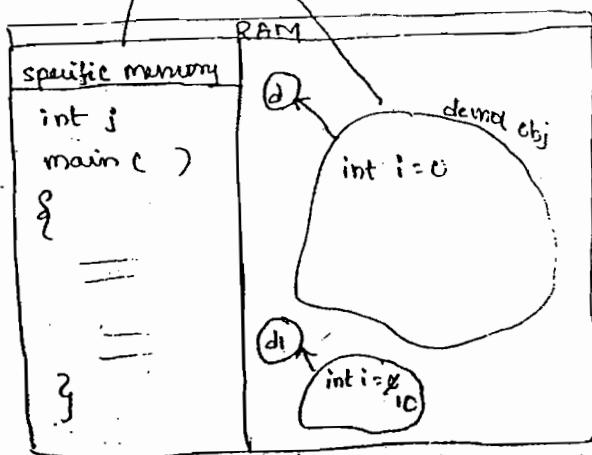
Sdemol.java

→ Sdemol.class

```

{
    int i;
    static int j;
    public static void main( String args[] )
    {
        s.o.p(i); // j is in RAM and available to main()
        s.o.p(i); // error - i is not in RAM and not
                    (available to main())
        Sdemol d = new Sdemol();
        j = 25;
        s.o.p(i);
        Sdemol d1 = new Sdemol();
        d1.i = 10;
        s.o.p(d1.i); // i = 10
    }
}

```



(O.T. 4)

Explanation: Till now we have learnt that as soon as we ask the JVM to execute a .class file, the JVM (i) loads the main method of the .class file into the RAM and then starts executing it. But this is not the actual concept. The underlying concept is as follows:

When ever we ask JVM to execute a .class file, JVM opens the .class file and searches for the static elements in the .class file and then loads all the static elements of the .class file into one specific memory location inside the RAM. Now JVM searches for the main() method among these static elements and then starts executing it.

NOTE: The memory space reserved in the RAM for the static contents of the .class file is known as context of the class.

We already know that, whenever new operator is encountered, JVM creates an object and loads all the non-static members of the class into that object. This is also known as instance of the class.

Even though, allocation of memory to the contents is the key criteria in the above two processes, there are certain important differences between these two.

contd... (P.T.O)

Differences between "context of a class" & "instance of a class"

- (i) During execution, any no. of objects can be created for the same .class file, but the context of the class can be created only once for a given .class file.
- (ii) As long as the address of the object (instance of class) is persistent in RAM, the object is persistent in RAM. As long as the main() method is under execution, the context of the class exists in the RAM. Once control encounters closing } of main(), the context of the class is deleted from the RAM.
- (iii) Garbage collection of objects will not have any effect on context of the class, but deletion of the context of the class from the RAM deletes everything related to that .class file from the RAM.
- (iv) Context of a class will be defined before or created before creating object of that class. But object of the class would be created only after the context of the class is created.

11.07.2006

We know that we can get access to the static variables or static functions from main() (since it is also a static method) without creating object of that class.

This facility helps us a lot to while we need to call a function more than once.

class
located
class
file.
of class)
un.RAM.
ution,
ree
ontext
effect
text
ted
re
ables
a

The following program illustrates this concept.

```
class Sdemol
{
    int i;
    static int j;
    static void function()
    {
        System.out.println(j);
        j = j + 1;
    }
    Sdemol s1 = new Sdemol();
    s1.i = 15;
    System.out.println(s1.i);
}
public static void main (String args[])
{
    function();
    function();
    Sdemol s1 = new Sdemol();
    s1.i = 150;
    System.out.println(s1.i);
}
```

sdemol.java → **Sdemol.class**

RAM

control & class

int j=0;

function();

{

=

main()

=

}

s1

i=15

s1

i=15

s1

i=150

Since function() is a static function, it will not be deleted from the RAM as soon as once the control comes out from the function. It will be deleted only when the control completes the execution of main() method. So we can call this function as many no. of times as required.

of `function()` is not a static function, then to get access to the function, we need to create object of the class. Now we can call the function through the object. But remember

But remember that, in this case whenever the control comes out from the function, it is deleted from the RAM and again when we call it, JVM has to load it to the RAM and then it gets executed. This results in poor memory management and in fact time consuming.

This is the disadvantage that results in, when we call a non-static function many times.

∴ whenever we require to call a function many times, we mention that function as static and by doing so, it will be loaded into the RAM only once but available for execution many times. It gets deleted from the RAM only when execution of `main()` method is completed.

In the above program, variable `i` and `function()` are present in the class, but to access variable `i` from `function()`, we have to create object of the class.

NOTE: In order to refer non-static members of the class from the static functionalities, we have to create object of the class.

(Q.T.Q.)

ers to
object
through

the
selected

JVM
sets
gement

In we

tatic
The
Anytime

tion

l(c)
able i
the class.

of
new

**

* NOTE: Till now we learnt that whenever JVM encounters 'new' operator it reserves memory space in the memory (RAM) and thus creates an object of that class and the address of this object will be assigned to a variable of type that particular class. This is what we know.

But in the background, something very interesting happens. Actually, the variable which holds the address of the object i.e. the reference, will have two parts.

The one part holds the address of the object and the other part holds the address of the context.

- The basic difference between the address of the object and the address of the context is as follows:

The part of the reference which is reserved for the address of the object is under the control of the program i.e. through the program, the programmer can assign addresses or null to that part.

The part of the reference which is reserved for the address of the context is not under the control of the program. The address of the context will be assigned to this part of the reference by the JVM only once and it can not be modified during the execution of the program.

NOTE: For a given reference, even though the address of the object is null, but still the other part of the reference contains address of the context.

Consider the following program:-

```
class Sdemo2  
{  
    int i;  
    static int j;
```

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

```
{  
    Sdemo2 s1 = null;  
    // s.o.p (s1.i); // error.  
    // s.o.p (s1.j); // j=0
```

```
Sdemo2 s1 = new Sdemo2();
```

```
s.o.p (s1); // Here we can see only the address of  
            the object. Address of context is hidden
```

```
s1.i = 25;
```

```
s1.j = 35; // highest priority will be given to  
            (os)  
            j = 35; address of the context.
```

```
s.o.p (s1.i); // i=25
```

```
Sdemo2 s2 = new Sdemo2();
```

```
Sdemo2 s3 = new Sdemo2();
```

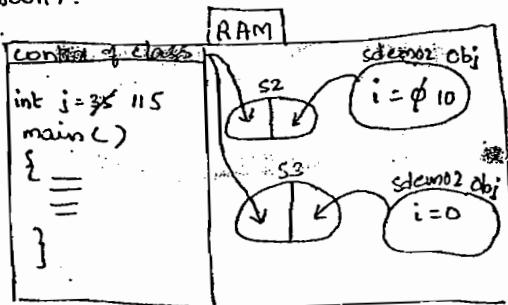
```
s2.i = 10;
```

```
s.o.p (s3.i); // i=0
```

```
s2.j = 115
```

```
s.o.p (s3.j); // j=115
```

```
}
```



Comments: Even though it is not required to refer static members of a class with object but we generally do it.

- object of any class would be associated with the context of the corresponding class and so, it is possible to refer any static elements with the object.
- all the static variables would be common to all the references of different objects of the same class.
- we cannot declare static int i; and int j; simultaneously because, we cannot declare two variables with same name within the same class.

// Another simple program.

```
class Sdemo2
{
    int i; static int j;
    public static void main()
    {
        Sdemo2 s1 = null;
        s1.i = 35;
        s.o.p(s1.i); // s1.i=35
        Sdemo2 s2 = new Sdemo2();
        s2.i = 10;
        Sdemo2 s3 = new Sdemo2();
        s.o.p(s3.i);
        s2.j = 203;
        s.o.p(s2.j);
    }
}
```

Question: How static functions are available to non-static functions.

Ans: This is illustrated in the following program.

```
class Sdemo3
{
    int x; static int y;
    void function1()
    {
        s.o.p ("Inside non static method - function1()");
        x = 10;
        y = 20;
        function2();
    }
    static void function2()
    {
        s.o.p ("Inside static function - function2()");
        s.o.p (y);
    }
    public static void main (String args[])
    {
        Sdemo3 s1 = new Sdemo3();
        function2();
        s1.function1(); // s1 has address of content and
                        // address of object.
    }
    // content of s1 would be available to function1();
    // so we are able to refer static variables/functions
    // from non-static functions.
```

.. static functions and static variables would be available to non-static functions of the same class directly but the converse is not true.

So to access non-static variables/functions from static functions, we need objects.

12.07.2006

NOTE: In the above program, function1() is a non-static function and function2() is a static function. When we say d.function1(), contents of the object pointed by 'd' are available to function1(). But when we use d.function2(), contents of the object pointed by 'd' are not available to function2().

Till now, we have seen the use of "static" keyword within the class.

Question: How can we refer the static elements of a class from another class?

Ans: To access the static elements of a class from another class, obviously we need to create the context of the class in which the static elements are present.

→ As we create object of a .class file to refer the non-static members of the class, we create the context of the class, to refer the static members of the class.

(P.T.O)

The following program illustrates the concept of accessing static variables of one class from another class.

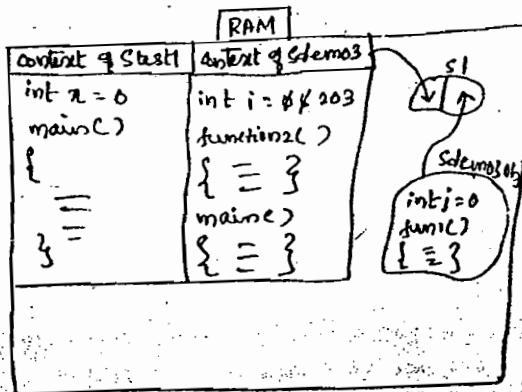
Assume that we already have a .class file, Sdemo3.class which has the elements as shown below.

c:\> Sdemo3.class

```
int i
static int :
void function1()
static void function2()
p.s. v. main()
```

Now, we write another program Test1.java through which we get access to the static elements of Sdemo3.class.

```
class Test1
{
    static int x;
    public static void main(String args[])
    {
        x = x + 1;
        Sdemo3.i = 5;
        System.out.println(Sdemo3.i);
        Sdemo3 s1 = new Sdemo3();
        System.out.println(s1.i); // s1.i=5;
        s1.i = 203;
        System.out.println(Sdemo3.i); // i=203
    }
}
```



(Q.T.Q)

Explanation:

In order to create context of the .class file, we just mention the name of the class in the program.

As soon as JVM encounters the name of the .class file for the first time, it creates the context of the class.

But, if the context of the class already exists in the RAM, then JVM just points to that existing context.

In the above program, when the control comes across the statement `sDemo3.i = 5`, JVM immediately creates the context of the class `sDemo3` and then initializes its static variable `i` with the value 5.

NOTE: Since context of a class is created only once we need not have any reference to access the contents of the context of that class file.

But in case of objects, we need to have a reference because, we can create in no. of objects for the same class and address of each object is different from that of the others.

As long as the main method, which is responsible for the creation of context of `sDemo3`, is under execution, the context of the class `sDemo3` will persist in RAM. It gets deleted only when the execution of the `main()` is completed.

(P.T.O)

NOTE: when we create the context of a class from a function, then when the control comes out of the function the function gets deleted from the RAM, but the context of the class still persists in the RAM.

This concept is illustrated in the following program.

```
class stest2
{
    void function()
    {
        Sdemo3.i = 115;
        System.out.println("Inside function()");

    }

    public static void main(String args[])
    {
        stest2 s1 = new stest2();
        s1.function();
        System.out.println(Sdemo3.i); //i=115
    }
}
```

Hint: If we want any variable or function to be available to all parts of the program, then declare that variable or function as static.

→ Context of a class will ~~not~~ never behave as local to a function in which it has been created.

Consider the following program to understand this except.

(Q.T.4)

(P.T.O)

```

class Stest3
{
    void function()
    {
        System.out.println(Sdemo3.i);
        Sdemo3.i = 15;
        System.out.println("Inside function()");
    }
}

public static void main(String args[])
{
    Stest3 s1 = new Stest3();
    Sdemo3.i = 2;
    s1.function();
    System.out.println("Sdemo3.i");
    Sdemo3 d = new Sdemo3();
    d.i = 4;
    System.out.println(Sdemo3.i);
    s1.function();
}
}

```

→ As we can declare a variable as static, we can even declare the object of a class as static.

This can be understood from the following program.

Prior to that, let us assume we have a class file A.class, as shown below.

```

A.class
└─ int i
└─ int j
└─ functionA()

```

Now in the following program, we can create object of A-class file and we can declare it as static.

```
class Sdemo4  
{  
    static int i=10;  
    static A a1 = new A();  
}  
  
class Stest4  
{  
    public static void main (String args)  
    {  
        Sdemo4.i = 15;  
        Sdemo4.a1.functionA();  
    }  
}
```

In the above program, the last statement inside main() is Sdemo4.a1.functionA(). This indicates that functionA() is a non-static method of an object a1 of some class (here it is A-class) where a1 is a static variable of Sdemo4 class.

** In a similar manner we can interpret the statement System.out.println(). This statement indicates that println() is a non-static method of 'out' object, (which is the object of some other class) where "out" is a static variable of System class.

NOTE: 'out' is not at all an object of System class.

'out' is the object of class PrintStream.

It can be understood as shown below.

```
class System  
{  
    static PrintStream out = new PrintStream();  
}
```

13-07-2006

compiler's Responsibility:

Whenever Java compiler converts a source code into its equivalent byte code it does two things:

- (i) It converts the source code into its equivalent bytecode statement wise and
- (ii) It checks whether that bytecode can be properly executed by the JVM or not. i.e. it finds out whether the JVM has proper environment or not.

Infact, it is the duty of the compiler to create a proper environment for the JVM to execute the byte code.

While compiling a source code, if, at any time the compiler finds out that the proper environment is not there, then the compiler tries to create the proper environment. Even then, if it is not possible, then it generates a compilation error.

Even though compiler creates the necessary environment for the JVM to execute the byte code, sometimes JVM will not be able to execute the byte code because of some errors. These errors are known as run-time errors.

Consider the following program:

```
class X      [x.java]
{
    int a,b;
    void functionX()
    {
        System.out.println("Inside functionX()");
    }
}

class Y      [y.java]
{
    int i;
    public static void main(String args[])
    {
        X x = new X();
        x.a = 10;
        x.functionX();
    }
}
```

In the above program, when we try to ~~execute~~ ^{compile} y.java file, as we had already discussed, the compiler creates the proper environment for the JVM to execute the byte code of y.class file. But in this program, we need the object of x.class file. But x.class file is not there in the harddisk, since we have not compiled it.

Actually, it should generate an error, but it will not. Instead the program will get executed perfectly. This is possible because of the compiler, as it has to create the proper environment, it compiles the x.java file to generate the x.class file and make it available to y.class file for execution.

Polymorphism: Defining more than one functionality with the same name in the same class is known as polymorphism.

static polymorphism: Defining more than one functionality with the same name but with different arguments in the same class is known as static polymorphism. [Refer *** on next page]

The process of defining more than one function with the same name, but with different arguments, within the same class is known as function overloading or function overwriting.

So, from the above two definitions, it is clear that static polymorphism is implemented using function overloading.

The following program illustrates the concept of static polymorphism:

```
class poly1
{
    void function1()
    {
        S.O.P ("Inside function1()");
    }

    void function1(int x)
    {
        S.O.P ("Inside function1 (int x)");
    }

    void function1(int x, int y)
    {
        S.O.P ("Inside function1 (int x, int y)");
    }

    int function1() /* It is not possible to define this function
                     * in this class because a similar function
                     * already exists. Even though the return data type is
                     * different, return data type of the functions is not
                     * considered in java w.r.t. polymorphism */
    {
        S.O.P ("Inside function1 (boolean flag)");
    }

    public static void main (String args[])
    {
        poly1 p = new poly1();
        p.function1(6,7);
    }
}
```

when the above program is executed,

we are passing values 6 and 7 as arguments to the function. So it is clear that void function (int x, int y) is going to receive these arguments and get executed i.e. the function will be chosen depending upon the arguments.

* * * : out of the four functions present in the program, which function has to be executed is decided at the compilation time itself. This is the reason why this type of polymorphism is known as static polymorphism.

NOTE: static means during compile time

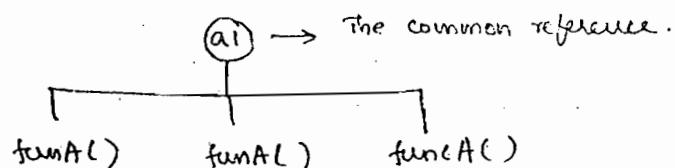
Dynamic means during run-time.

Java is highly dynamic. Everything happens at run-time. But still this polymorphism is known as static polymorphism just because of the above reason.

Dynamic polymorphism:

out of some functions with the same name, same arguments, the function which has to be executed is decided at the runtime. This concept is known as dynamic polymorphism.

NOTE: The three functions referred to, in the above sentence are scattered in different classes but associated with the same reference, as shown below.



- Dynamic polymorphism can be achieved only through Inheritance. It is completely dependent upon inheritance.

Note: Dynamic polymorphism is implemented using function overriding

NOTE: System.out.println() method basically works on the concept of static polymorphism.

Disadvantage of static polymorphism:

With static polymorphism, there is a chance for the JVM to go into an ambiguous state - which is completely undesirable. This is the ~~the~~ disadvantage of static polymorphism. The following program highlights this disadvantage.

```
class poly2
{
    void function1 (int x)
    {
        s.o.p ("Inside function1 (int x)");
    }
    void function1 (float x)
    {
        s.o.p ("Inside function1 (float x)");
    }
    void function1 (long x)
    {
        s.o.p ("Inside function1 (long x)");
    }

    public static void main (String args[])
    {
        Poly2 P = new poly2 ();
        int i = 6;
        float j = 6;
        long k = 6;
        P.function1 (6); // Now this argument can be
                        // received by any of the three functions
                        // because its datatype is compatible with
                        // the datatype of the parameters of all the three functions.
    }
}
```

Now, the question is, out of the three functions, which function will receive the argument and get executed?

Here JVM adopts a strategy to solve this conflict.
it makes use of priority concept.

The highest priority is given to the function whose parameter's datatype is same as that of the argument.
In this case void function (int x) receives the argument.
Suppose, let us assume that, this function is not there. Then JVM searches for the alternatives. Now it finds two functions - void function (float x) & void function (long x).

Again priority comes into picture. Obviously void function (long x) will receive the argument as datatype long has higher priority when compared to float.

Even if this function is not there, then argument will be received by - void function (float x). Till now there is no problem.

Now, what happens if there are two functions whose datatype of the parameters has equal priority?

Exactly in this case JVM will be unable to decide what to do and it goes into ambiguity - which is completely undesirable in real time. This happens in the case of static polymorphism and hence is the disadvantage.

14-07-2006

The following ^{two} programs also exploit the disadvantage of static polymorphism. The second program shows us how the JVM goes into ambiguous state when objects of classes are passed as arguments to the functions exhibiting static polymorphism.

```

④ class PolyA
{
    void function(int x, int y)
    {
        s.o.p("Inside function (int x, int y)");
    }

    void function(int x, float y)
    {
        s.o.p("Inside function (int x, float y)");
    }

    void function(float x, int y) // :: Hierarchy is changed,
                                    // this function is allowed.
    {
        s.o.p("Inside function (float x, int y)");
    }

    public static void main(String args[])
    {
        int a = 6;
        float b = 7.8f;

        PolyA p = new PolyA();
        p.function(a, b);
        p.function(b, a);
        p.function(a, a);
    }
}

```

when we execute the above program, we do not get any problem. Now, let us comment the first function inside the class. When we do so, compiler generates a compilation error when it comes to p.function(a, a), because 'a' is an integer variable and we are passing two integers as arguments in the function call. Note that we have

two alternate functions which are capable of receiving these two arguments. But upon keen observation we can find that both the alternate functions have the same priority. Now JVM will not understand which function has to be called. So it goes into ambiguous state resulting in a compilation error.

② /* Another program illustrating the disadvantage of static polymorphism */

```
class PolyB
{
    void function(int a)
    {
        System.out.println("Inside function (int a)");
    }

    void function(A a1)
    {
        System.out.println("Inside function (A a1)");
    }

    void function(B b1)
    {
        System.out.println("Inside function (B b1)");
    }

    public static void main(String args[])
    {
        PolyB P = new PolyB();
        P.function(6);
        P.function(new B());
        P.function(new A());
        P.function(null); // Here we get a compilation error.
    }
}
```

In the above program, we are passing objects of classes as arguments. When we pass objects of class A and class B as arguments we do not get any problem. But, when we pass null as an argument, a compilation error is generated in this program because, the class PolyB has two functions defined to accept objects of a class as arguments with same priority. Since, in this case the argument is null, the compiler will not understand, which function is to be called upon. So it generates a compilation error.

NOTE: consider the following statements.

```
int x=5; s.o.p(x); s.o.p(5);  
float y=6.8f; s.o.p(y); s.o.p(true) e.t.c.  
s.o.p(x,x); // compilation error
```

All the above statements are valid, except s.o.p(x,x); This means, println() method is not one in the PrintStream class but ^{and} it the printStream class has different println() methods to accept variables of each and every primitive datatype, as arguments.

There is no println() method defined to accept two arguments, so it gives a compilation error for s.o.p(x,x);

Consider the following statements:

```
A a1 = new A(); B b1 = new B();  
s.o.p(a1); s.o.p(b1);
```

Note that there is no `println()` method defined to accept objects of undefined classes as arguments. But still the above two `println()` statements are executed properly. How?

Theory: Actually, a reference of type some class X will point out only object of class X. But there is one and only one class which is defined whose reference points to objects of another class. That class is the Object class.

Ex : Object o1 = new A(); e.t.c.

Thus, whenever we pass an object of a class as an argument to a `println` method, then in the background, the `println()` method defined to accept object of class `Object` is going to get executed. It displays the address of the object which we have passed as an argument.

NOTE: When we write `s.o.p(null)`, then it generates a compilation error because, the compiler would not understand which `println` method has to be called.

This is because, three `println()` methods are defined to accept objects as parameters of different classes :

- (i) Object of class (userdefined)
- (ii) Object of string class.
- (iii) Object of array class.

15-07-2024

Constructors:

- A constructor is a function, which would be executed only at the time of creating an object.

Comparison between a constructor and a function:-

- A normal function can be called as and when we require to call it, but we can't call a constructor as and when we want.
- Constructor would be executed only once on the object and that too, only at the time of creating the object.
A function can be called on an object as many no. of times as we want.
- Name of the function can be any name, but the name of the constructor must and should be the name of the class.
- A constructor will not have any return data type (not even void) whereas a function should have a return datatype. (at least void).
- The keyword static cannot be used with constructors. It is applicable to functions and other variables.

→ We know that, whenever JVM encounters the new operator, it creates an object of that class and loads all the non-static members of the class into that object and then, the address of the object is assigned to a variable.

But something other than this happens prior to this when JVM encounters the 'new' operator. Here we can see the function of constructors.

Let us have a deeper look, at the statement used for creating objects:

```
A a1 = new A();
```

Here A() is a constructor. This statement literally means, asking JVM to create object of A.class file by executing the constructor A() defined in A.class file.

But note that, till now, in our programs, we have not defined any constructor in our programs. Then how can JVM execute a constructor which is not defined.

The theory behind this concept is as follows:

when we compile the program, we get the .class file for the .java file. Now if the .java file consists any constructor, then the .class file also will have the same constructor. Suppose if the .java file does not have any constructor defined in it, then the compiler adds a default constructor with the name of the class into the .class file and it is this default constructor that is going to get executed whenever the JVM encounters the 'new' operator and creates an object for the class.

NOTE: we can have a .java file without a constructor, but we can never have a .class file without a constructor. This is because of the compiler's duty, which we discussed above.

... sample programs on constructors */

```
class A
{
    int i, j;

    A()
    {
        S.o.p("Inside constructor A()");
        i = 5;
        j = 10;
    }

    void function()
    {
        S.o.p("Inside function");
        S.o.p(i);
        S.o.p(j);
    }

    public static void main(String args[])
    {
        A a1 = new A();
        A a2 = new A();
        a2.function();
    }
}
```

Since, in this program, we have a constructor defined inside the class, compiler will not insert any default constructor into the .class file.

Now, when JVM comes, sees the statement A a₁ = new A(), it will first create the object, load the constructor, execute it on the object and then delete it from the RAM. After all this happens, then the address of the object will be assigned to the variable a₁.

The constructor will not be loaded into the object. It gets executed only once at the time of creation of the object and hence its address is not maintained in the object.

NOTE

NOTE: If we want to execute a constructor once again on the same object, it is not possible. It can be executed on a different object created on the same class.

Ex: A a₂ = new A(); // a₂ is an object, different from a₁.

- The default constructor provided by the compiler is a zero statement constructor i.e. the constructor will not have any body. Ex: A() {}.

Question: What is the use of zero statement ^{default} constructor provided by the compiler?

Explanation: Strictly speaking, the default constructor is not a zero statement constructor. It will have one default statement.

Even in the constructor which we defined, we physically have three statements, but literally speaking it will have four statements. The fourth statement is the default statement and it will be the first statement inside the constructor. The use of this, will be studied later.

- Constructors are very much related to inheritance.
- We can use a constructor for any purpose, but usually programmers use it to initialize the instance variables.

NOTE: It is not the job of the default constructor to initialize the uninitialized variables. It is the job of the VM.

- We can define a function with the name of the class, as we do for constructors, but it is not a good programming habit.

/* According to coding standards, defining a function with the name of the class is not allowed */

Example programs:

```
class B
{
    int a,b;
    B()
    {
        s.o.p(a);
        s.o.p(b);
    }
    public static void main (String args[])
    {
        B b1 = new B();
    }
}
```

```
class D
{
    int a,b;
    D()
    {
        s.o.p ("Inside constructor");
    }
    void D()
    {
        s.o.p ("Inside function");
    }
    public static void main (String args[])
    {
        D d1 = new D(); d1.D();
    }
}
```

constructor overloading: It is similar to function overloading.

NC

```
class X  
{  
    int a, b;  
    X(int i)  
    {  
        a = i;  
        S.O.P("Inside X(int i)");  
    }  
    X(int i, int j)  
    {  
        a = i;  
        b = j;  
        S.O.P("Inside X(int i, int j)");  
    }  
    public static void main(String args[])  
    {  
        X x1 = new X(6); // const. X(int i) will be executed  
        X x2 = new X(); // here we get a compilation error.  
        X x3 = new X(true); // here also we get a ...  
    }  
}
```

It is not compulsory that a class should have a zero argument constructor.

In the above program, we have two constructors. So, the default constructor will not be provided. Now when ever the compiler comes to the last two statements, it searches for a constructor which is defined to receive zero argument and another constructor which is defined to accept boolean argument. It does not find any such constructors defined and hence gives a compilation error.

overloading.

NOTE: All the concepts related to static polymorphism are applicable to constructors even.

17.07.2006

In order to create an object of a class, the constructor has to be executed compulsorily.

'this' keyword: whenever it is required to point an object from a functionality which is under execution because of that object, then we use the 'this' keyword.

The job of 'this' keyword is only to point. It always points to an object that is executing the block in which 'this' keyword is present.

In a nut-shell, 'this' keyword points to the current object.

The following program illustrates the use of 'this' keyword.

```
class X
{
    int i;
    x(int i)
    {
        this.i = i+1;
        i = i+1;
        s.o.p(i);
    }
    void function()
    {
        int i = 67;
        s.o.p(i); // i = 67.
        s.o.p(this.i);
        this.i = this.i + 1;
    }
}
```

```

public static void main (String args[])
{
    X x1 = new X(5);
    s.o.p (x1.i);
    x1.function();
    s.o.p (x1.i);
    // this.i = 203; //error
}
}

```

NOTE: 'this' operator points to address of the instance.
It will never point to the context of the class.

We know that static functions are not executed with the address of the instance even though we call them using it.

That is why, 'this' operator works only in non-static blocks - It will not work in static blocks.

Hence, when we used `this.i = 203;` in `main()`, we got an error, since `main()` is a static block.

- Constructor is a non-static block. That is why, we are able to use 'this' operator inside a constructor.
- We can call a constructor explicitly, without creating another object of that class only through another constructor.
- We can not call constructors through functions.

Note: When we call a constructor explicitly (i.e. through another constructor), then the first statement inside the calling constructor must and should be the call to the constructor.

Example program

```
class Y
{
    Y()
    {
        this(6);
        s.o.p("Inside constructor Y()");
    }

    Y(int i)
    {
        this(i, i+1);
        s.o.p("Inside constructor Y(int i)");
    }

    Y(int i, int j)
    {
        s.o.p("Inside constructor Y(int i, int j)");
    }

    public static void main(String args[])
    {
        Y y1 = new Y();
        s.o.p("end of main()");
    }
}
```

Uses of 'this' operator:

- (i) To point a constructor of a class from the constructor of the same class.
- (ii) To point current object from the non-static blocks.

(P.T.O)

NOTE: Let us assume we have two classes:

class A and class B.

Now, the point is that, we can call constructor of class B from the constructor of class A, without creating an object of class B.

String Classes:

Before going to learn about String classes, we have to learn something about packages.

Normally, a class would be represented in the form of a file, in the harddisk.

If we save all the related .class files in one folder, we can say that we are binding all the related .class files together. Now this folder can be called as a package.

In simple terms, binding all the related classes together is nothing but packaging.

→ Every package will be a folder, but every folder will not be a package.

18-07-2006

→ If we are able to set path of contents of one particular folder to our .class file using import statement, then we can call that folder as a package.

Let us assume that we have two .class files:

A.class and Test.class

Let A.class reside in a folder by name folder1 in the C drive. Let Test.class exist at some other place in the C drive.

Now consider the following program.

```
class Test
{
    main p. s. v. main(String args[])
    {
        A a1 = new A(); // error.
    }
}
```

we get an error while creating the object of class A from Test class because the path to A.class file which is in folder1 is not set properly.

so, to set the path for the .class file, we use the import statement.

Ex: import folder1.A;

- when we use the import statement, the path would be set during the compilation time.
- import folder1.A; sets the path to A.class file from our .class file.
- import folder1.*; sets the path to all the .class files of folder1, from our .class file (i.e. * represents all files)
- import keyword works fine only if the folder is a package. (In this case it is folder1).

NOTE: System.class file belongs to a package by name lang.

- All the .class files used to execute the java programs are present inside a large package by name java. as shown below.

| java | | | |
|--------------|-----------------|----|-----|
| lang | awt | io | net |
| System.class | button.class | - | - |
| String.class | textfield.class | - | - |
| Thread.class | list.class | - | - |
| e.t.c. | e.t.c. | - | - |

Ex: import java.lang.*; 85

- Note that till now, we have not imported java.lang.* package into any of the user defined class file. but we have been continuously using System class (System.out.println) in our programs. Actually, since we are not setting the path, it should result in a compilation error. but it did not happen so.
- The reason is, compiler checks if we had imported the lang package. if not, compiler by default sets the path to the lang package from every .class file.
- Thus java.lang is the default package available for every .class file.
- Through some documentation, we can know all the functions e.t.c related to the predefined classes/packages. That documentation is nothing but API
- API : Application programming Interface.
- API is developed using HTML.
- API surfing : Learning about the different functions, packages, classes e.t.c. by referring to the API.

NOTE: Everything defined inside the System class would be defined as static. Nothing is non-static inside the system class.

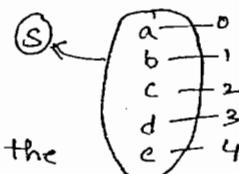
// Refer to the API, to learn more about different functionalities, classes, packages and the methods enter return data types, the arguments they accept e.t.c //.

String Classes:

- A string is a group of characters.
- String classes are used to represent three groups of characters.
- We can even create an object of a string class without using 'new' operator. Instead of the 'new' operator we use double quotes (" ") .
- As soon as JVM encounters " ", it understands that it has to create an object of the string class with the contents present in " " and for each and every individual character inside " " it assigns a unique index value starting from zero.

Ex: String s = "abcde";

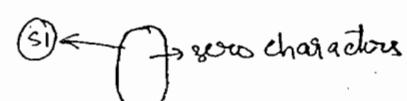
Address of the object is assigned to the variable s.



This object has a very near relation with arrays.

- Once, a string object is created, the data inside the object cannot be modified. This is the reason, why, we say that, Strings are immutable.
- Different ways of creating string objects are shown below.

String s1 = new String();

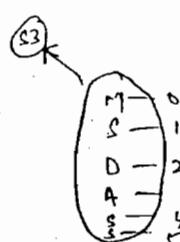


String s2 = new String("abc");



String s3 = "MSDASS";

e.t.c.



- consider the following two statements.

`A a1 = new A(); String s1 = "abcde";`

Both the above statements create the objects of their respective classes.

NOW, when we say `s.o.p(a1);` it prints the address of the object; whereas for `s.o.p(s1);` it prints abcde (i.e. the content of the object). WHY?

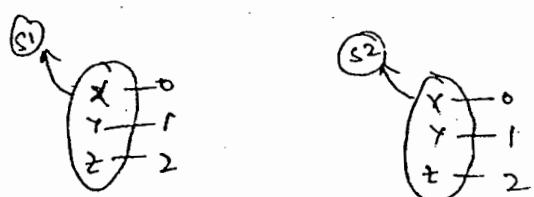
- The explanation is related to static polymorphism.
The `println()` method, called in the above two cases is not the same.
- one `println()` method is defined to accept object of Object class and its functionality is to print the address of the object.
- The second `println()` method is defined to accept object of String class and its functionality is to print the contents of the object pointed by the address.

Question: what is the difference between creating the string object using new operator and creating the string object using double quotes?

Ans: Using new operator, we can create many objects of the same string class.

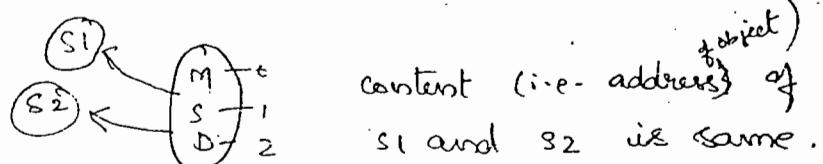
For example, `String s1 = new String("xyz");`
`String s2 = new String("xyz");`

Here, though the contents are same, two different objects are created and the addresses are assigned to different variables.



But when we create objects of String class using double quotes, we can utmost create only one object ^{in the entire program}. If we try to create more objects, the same address of the already created object will be assigned to the variables pointing to the newly created objects.

Ex: `String s1 = "MSD"; String s2 = "MSD";`



The above explained difference is illustrated in the following program.

```
class Demo1
{
    public static void main (String args[])
    {
        String s1 = "abc";
        String s2 = new String ("xyz");
        String s3 = new String ("xyz");
        if (s2 == s3)
            s.o.p ("s2 & s3 are equal");
        else
            s.o.p ("s2 & s3 are not equal"); // This statement
                                                // is executed
        String s4 = "abc";
        if (s1 == s4)
            s.o.p ("s1 & s4 are equal"); // This statement
                                            // is executed.
        else
            s.o.p ("s1 & s4 are not equal");
    }
}
```

19.07.2000

The following program is another example which indicates that using double quotes, we can create only one string class representing a group of characters, not only in the functionality but in the entire program.

```
class Sdemo2
{
    String s1 = "abc";
    void function()
    {
        String s1 = "abc";
        if (s1 == this.s1)
            System.out.println("s1 & this.s1 are equal");
        else
            System.out.println("s1 & this.s1 are not equal");
        System.out.println("end of function");
    }

    public static void main (String args[])
    {
        Sdemo2 d = new Sdemo2();
        d.function();
        String s3 = "abc";
        if (s3 == d.s1)
            System.out.println("s3 & d.s1 are equal");
        else
            System.out.println("s3 & d.s1 are not equal");
    }
}
```

→ String concatenation:

Appending a string to another string is known as string concatenation.

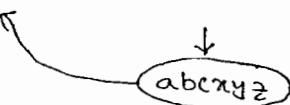
In the The '+' symbol acts as the concatenation operator.

NOTE: + operator works as an arithmetic operator if and only if the two operands supplied to the + operator are arithmetic numbers.

+ operator works as a concatenation operator if atleast one operand supplied to the + operator is a String object.

Concatenation operator always outputs a new string class object with the contents of LHS and RHS of the + operator.

Ex: String s1 = "abc" + "xyz";



s.o.p(s1); // it prints abxyz.

String s2 = "MSD"; s.o.p(s2); // MSD is printed.

s2 = s2 + "SSA"; s.o.p(s2); // MSDSSA is printed.

We said that strings are immutable (i.e. contents of the object + string class cannot be changed) but how are the above statements possible?

Actually, the content of the object pointed by s2 is not changed. In fact, a new object is created and some new content is appended to the existing content and this is stored in the object and the address of the object is assigned to the variable. That's all.

In the above example, the content "MSD" is not at all modified. Some new content "SSA" is added to it and then the entire content is stored in a new object.

- Using concatenation operator, we can represent all the primitive datatypes in the form of their equivalent string class objects.

```
int x = 10;
string s4 = "";
s4 = s4 + 10;           This 10 is not an integer now.
                        ↗ It is a string class object.
cout << s4; // 10 is printed.
```

```
int x = 10;
string s3 = "MSD" + 10
cout << s3; // MSD10 is printed.
```

```
string s5 = "xyz" + x + 10;
           ↓
           xyz10 + 10
           ↓
           xyz1010
cout << s5; // xyz1010 is printed.
```

```
string s6 = x + 10 + "abc";
           ↓
           20 + "abc"
           ↓
           20abc
cout << s6; // 20abc is printed.
```

```
String s7 = "abc" + x * 3;      * operator has more
                                ↓      priority (precedence) over
                                "abc" + 30      + operator.
                                ↓
                                abc30
cout << s7; // abc30 is printed.
```

```
string s7 = x * "abc" + 3; // It gives error.
```

Because, * operator works only on arithmetic numbers

API Surfing :

Question: How can we search one specific function in a group of functions of a class in an API?

Tips (i) First of all try to guess the return datatype of the function. If we know the return datatype, then we can search only those functions, which have that return datatype.

(ii) The names of the functions will give us some more details (as names of functions in Java are userfriendly i.e. name of the function is closely related to its functionality).

- whenever we find the term 'Deprecated' in front of any method or a class, it indicates that method or class is not in use in that particular version.

NOTE: we are not supposed to use deprecated methods while writing programs.

StringBuffer class :

We said that strings are immutable. But we have another class by name StringBuffer class, the contents of whose objects are mutable. i.e. we can alter the contents of the object which is created upon the StringBuffer class.

Ex: `StringBuffer sb1 = new StringBuffer("abc");`

Some of the methods available in the StringBuffer class are
append(); reverse(); replace(); etc.

By using these methods, we can alter the contents of the object of StringBuffer class.

- `StringBuffer sb2 = new String("xyz");` } //Gives a compilation error.
 `String s3 = new StringBuffer("abc");` }

Both the above statements are not valid statements. The reason

is, we are trying to assign object of one class to reference of another class which is against the rule that an object of a class should be assigned to the reference variable of the type same class.

→ Arrays: in java, arrays are treated as objects.

while using arrays, we create objects for arrays, whose class is non-existent (which we have not done so far).

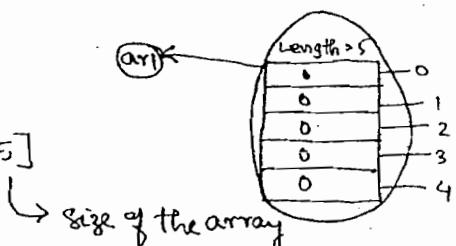
when ever JVM encounters [], it understands that it has to create an object.

thus, array objects can be created without using the 'new' operator.

25-07-06

- one of the peculiar characteristics of arrays is that, without using the name of the class, we use 'new' operator to create an array object.
- in any programming language, we cannot create an array without mentioning the size of the array.
- once the size is fixed, defined, it is fixed. we cannot alter it during run time.

Ex: int arr[] = new int[5]



arr is a reference to an array of integer type.

s.o.p(arr[0]); // 0

The above s.o.p statement prints '0' because uninitialized locations (values) would be initialized with their default values.

reference
in object
of the
array
class
it has
new
at,
operator
an
can not
0
1
2
3
4
initialized
values.

```
arr[1] = 10; // initializes the location with index 1, to 10.
```

length: it is a non-static variable defined inside array object and the value of that variable is nothing but the size of the array.

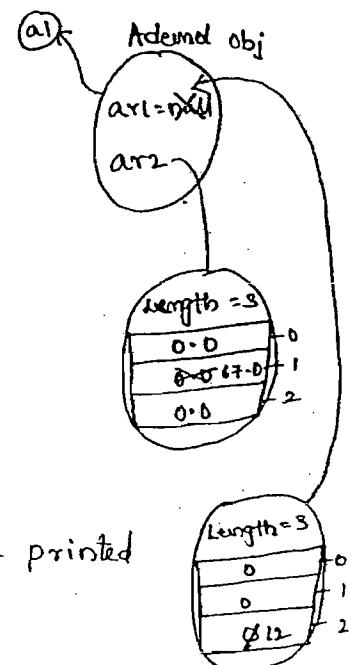
```
s.o.p (arr.length); // prints the size of the array arr.
```

NOTE: with respect to arrays, length is not a function. As mentioned above, it is a non-static variable that belongs to array object.

Another way of declaring arrays:

The following program illustrates the concept of declaring arrays and using them.

```
class Ademo1
{
    int arr[ ];
    float arr2[ ] = new float arr2[3];
    public static void main (String args[])
    {
        Ademo1 a1 = new Ademo1();
        s.o.p (a1.arr); // a1.arr = null.
        s.o.p (a1.arr2); // Address will be printed
        a1.arr2[1] = 67.0;
        a1.arr = new int [3];
        s.o.p (a1.arr[1]); // 0 is printed
        a1.arr[a1.arr.length - 1] = 12;
        int arr[ ] = new int [4];
    }
}
```



contd... (p.t.o)

```

for( int i=0 ; i < arr.length ; i++)
{
    arr[i] = i * i;
}

for( int i=0 ; i < arr.length ; i++)
{
    s.o.p( arr[i]);
}

} // end main

} // end class.

```

~~OP:~~
 null
 CF @ 1ba34f2
 0
 0
 1
 4
 9

Creating array object without using new operator:

```

class Ademo2
{
    public static void main (String args[])
    {
        int i[] = { 10, 9, 8, 7 } ;
        char ch[] = { 'a', 'b', 'c', 'g', '4' } ;

        s.o.p( ch[1]) ; // ch[1]=b
        s.o.p( i[3]) ; // i[3]=9
    }
}

```

NOTE: As soon as JVM encounters {} , it creates an object with the values present in {} and assigns the address of the object to a variable.

NOTE

In this way, we can create array objects of all primitive data types.

In a similar way, we can define arrays of objects of classes.

The following program illustrates this concept.

```
class Ademo3
```

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

```
    A a1[] = new A[3];
```

```
    s.o.p(a1[0]); // a1[0] = null.
```

```
    s.o.m a1[0] = new A();
```

```
    s.o.p(a1[0]); // address is printed
```

```
    a1[0].i = 10
```

```
    a1[0].j = 20;
```

```
    a1[0].functionA();
```

```
    a1[1] = a1[0];
```

```
    s.o.p(a1[1].i); // 10 is printed
```

```
    a1[2] = new A();
```

```
}
```

```
}
```

creates
and
ble.

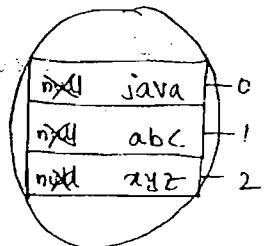
NOTE: int i[] = ... is same as int[] i = ...

(P.T.O)

// Another sample program.

NOTE

```
class Ademo4
{
    public static void main (String args[])
    {
        String s1[] = new String[3];
        System.out.println(s1[0]); // null is printed
        s1[0] = "java";
        System.out.println(s1[0]); // java is printed
        s1[1] = "abc";
        s1[2] = "xyz";
        System.out.println(s1); // prints address of array object of type String.
        System.out.println(s1[2]); // xyz is printed.
    }
}
```



26-07-2006

1. Another sample program.

```
class Ademos
{
    public static void main (String args[])
    {
        String s1[] = {"abc", "xyz", "msd", "csa"};
        Al a1[] = {new Al(), new Al(), new Al()};
        System.out.println(s1.length); // 4 is printed.
        System.out.println(s1[3]); // ssa is printed
        a1[0].i = 11;
        System.out.println(a1[0].i); // 0 is printed
    }
}
```

NOTE: we can't use {} to declare arrays wherever we want. i.e. if we declare an array at one place and initialize it at another place, then we can't use {} for initialization. In this case we have to use the 'new' operator.

consider the following example.

```
class Ademo5
{
    string s1[] = { "abc", "xyz" },
    int i1[];
    public static void main (String args[])
    {
        Ademo5 ar = new Ademo5 ();
        // ar.i1 = { 1, 2, 3 }; // This is not allowed. Refer note above.
        ar.i1 = new int[6]; // This is allowed. " "
        int j1[] = new int[7]; // This is not permitted because
                               // size is not defined.
        /* Any non-negative integer can be defined as the size */
        int j1[] = new int[0]; // An array with zero elements.
        s.o.p(j); // address of array object j will be printed.
        s.o.p(j.length); // 0 is printed.
        // s.o.p(j[0]); // error. Because array has no elements.
        String s1 [] = new String[0];
        String s2 [] = { } ;
    }
}
```

(P.T.O)

A function which returns array object

```
class Ademo6
{
    int[] getArray(int i)
    {
        int[] j = null;
        j = new int[i];
        for (int k=0; k<i; k++)
            j[k] = k*k;
        return (j);
    }
    public static void main (String args[])
    {
        Ademo6 a1 = new Ademo6();
        int i[] = a1.getArray(10);
        for (int j=0; j < i.length; j++)
            System.out.println(i[j]);
        a1.getArray(5); // return datatype not handled.
    }
}
```

function which accepts array object as an argument

Let function1 (int i[]) be defined to accept array object.

We can call function1 () in the following ways

function1 (null); // passing null as the argument

function1 (new int [0]); // passing a zero element array object

int i[] = {1, 2, 3};

function1 (i); // declaring an array and then passing it as an argument.

```

class Ademot
{
    void function1(int i[])
    {
        if (i != null & i.length > 0)
        {
            s.o.p ("Length: " + i.length);
            s.o.p (i[i.length - 1]); // prints the last element of
            the array.
        }
        else
            s.o.p ("improper values supplied");
    }

    public static void main(String args[])
    {
        Ademot a1 = new Ademot();
        a1.function1(new int[0]);
        a1.function1(null);
        int i[] = {1, 2, 3};
        a1.function1(i);
    }
}

```

point
object.
passing

As we can pass an object of an integer array as an argument to a function, we can pass an object of a string array as an argument to a function.

(P.T.O)

```

class Ademo
{
    public static void function1(String args[])
    {
        if (args != null)
            s.o.p(args.length);
        else
            s.o.p("null is passed as the argument");
    }

    public static void main(String args[])
    {
        function1(null);
        s.o.p("Inside main");
    }
}

```

on the above program, we have a class Ademo which has a function by name function1 and main method. Both are defined to accept string object as arguments. Signature of the both the functions is public static void. Then what is the difference between these two functions?

NOTE: JVM identifies public static void main (String args[]) but it does not identify p.s.v. function1 (String args[]). This is because, whenever JVM executes a .class file, it first searches for the main() method and ignores all other methods with the same signature.

Here JVM is calling the main method, but as far as function1() is concerned, we have to call it.

NOTE 1: we can call main method of one .class file from main method of another .class file.

In the following program Test.java, we call the main method of the above program Ademo.java.

```
class Test
{
    public static void main (String args[])
    {
        String s1 = { "MSD", "SSA" }
        Ademo.function1 (s1);
        Ademo.main (s1);
    }
}
```

NOTE 2: when JVM calls the main method, it executes main method as a thread. when we call the main method JVM executes it as a function.

- whether JVM calls the main method or we call the main method, the corresponding arguments have to be passed to the main method.

- **
- Now, the question is, when JVM calls the main method, it has to pass the relevant argument to the main method. What is that argument?

Hint: Definitely the argument is not null. Then what is it?

NOTE: we can call main method of one .class file from the main method of another .class file by passing null as argument. But JVM can not call main method of a .class file by passing null as an argument. (Refer note-2 on the previous page).

here comes the concept of command line arguments.

```
class ArgDemo
{
    p. s. v. main (String args[])
    {
        s.o.p (args.length);
        if (args.length > 0)
        {
            s.o.p (args[args.length - 1]);
        }
    }
}
```

we can give any name which is not a keyword. it is the usual practice to use args, but not compulsory.

c:\> javac ArgDemo.java

c:\> java ArgDemo abc xyz MSD SSA

{ "abc", "xyz", "MSD", "SSA" }

c:\> java ArgDemo Arg Demo

{ "Arg", "Demo" }

c:\> java ArgDemo

{ }

command line arguments.

WRAPPER CLASSES

There are many situations, where we need to represent the primitive datatypes in the form of objects.

∴ Some classes have been defined to represent the primitive datatypes in the form of objects.

These classes are known as Wrapper classes.

These classes wrap the concept of object on the primitive data types.

Uses of wrapper classes:

- (i) To represent primitive data types in the form of their equivalent objects.
- (ii) wrapper classes have some functions which are useful in representing the contents of a string class object of a primitive datatype in its original form (primitive datatype).

Ex: int x = 10;

```
String s1 = "x" ;
s1 = s1.trim(); → s1 is a string class object.
s.o.p(s1); // 10 is printed
```

Now, this string class object can be converted back into its equivalent primitive datatype by using the functions of wrapper classes.

NOTE: Since we have eight primitive data types,

obviously, we have eight wrapper classes, as shown below:

Byte, Short, Integer, Long, Float, Double, Character, Boolean.

- The wrapper classes are present in the java.lang package.

(P.T.O)

(i) one constructor to accept primitive data argument

Ex: `int x = 10;`

`Integer i1 = new Integer(x);`

(ii) Another constructor to accept string class object of primitive datatype as argument.

Ex: `int x = 10;`

`String s1 = x + " ";`

`Integer i2 = new Integer(s1);`

NOTE: No wrapper class supports a constructor defined to accept zero arguments.

Since, there are two constructors available, we can pass either the primitive datatype (or) String class object of the primitive datatype as argument.

- To get back the primitive datatypes from their equivalent object forms, we have ^{some} static and non-static methods shown below.

non-static methods

`intValue();`

`floatValue();`

`doubleValue();`

e.t.c.

static methods

`parseXXX(string class object)`

Ex: `parseInt(s1);`

`parseFloat(s2);`

`parseDouble(s3);`

e.t.c.

Ex: `int j = i1.intValue();`

Ex: `int y = Integer.parseInt(s1);`

NOTE: The functions of the wrapper classes can represent the string class object of primitive datatype in its equivalent primitive datatype only if the string class object contains numbers as characters.

But, normally a string class object can have spaces as characters, which can not be represented in the form of their primitive data types (numbers).

∴ it is highly recommended to call the trim() method on the string class object of the primitive datatype before passing it as an argument to the constructor of the wrapper class (or) the static methods of the wrapper class.

// the following program illustrates the use of wrapper classes

```
class SimpleInterest
{
    void gets() {double P, float r, int t)
    {
        double SI = (P * r * t) / 100;
        System.out.println("Interest is : " + SI);
        double ta = P + SI;
        System.out.println("total amount is : " + ta);
    }
    public static void main(String args[])
    {
        SimpleInterest si = new SimpleInterest();
        if(args.length >= 3)
```

contd--(P.T.O)

```

{
    String s1 = args[0].trim();
    Double d1 = new Double(s1); } using non-static methods.
    double p = d1.doubleValue();
    float r = Float.parseFloat(args[1].trim()); } using static
    int t = Integer.parseInt(args[2].trim()); } methods.
    si.gets(p,r,t);
}
else
    s.o.p("Improper command line arguments supplied");
} // main
} // class.

```

Try this: Integer i = new Integer(77);

s.o.p(i); // instead of address of i, it prints
77 on the console.

The reason for this can be explained using Inheritance.

~~28.07.06~~

Inheritance:

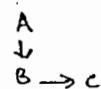
Question: How can you access non-static members of a class X from another class Y without creating the object of class X inside class Y.

Ans: with the help of the keyword extends, this is possible. This leads to the concept of inheritance.

Inheritance: Technically speaking, getting the properties of one object of a class to another object of another class is known as inheritance.

Inheritance of three types:

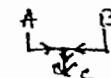
(i) Multilevel inheritance: One class extending another class as in a hierarchical structure is termed as multiple multilevel inheritance.



NOTE: Java supports only multilevel inheritance.

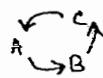
(ii) Multiple inheritance: One class extending two other classes simultaneously is known as multiple inheritance.

- C++ supports multiple inheritance.



(iii) Cyclic Inheritance:

In cyclic inheritance, one class extends the next class and so on and the last class extends the first class forming a circle.



Note: Cyclic inheritance is never used and no programming language supports it.

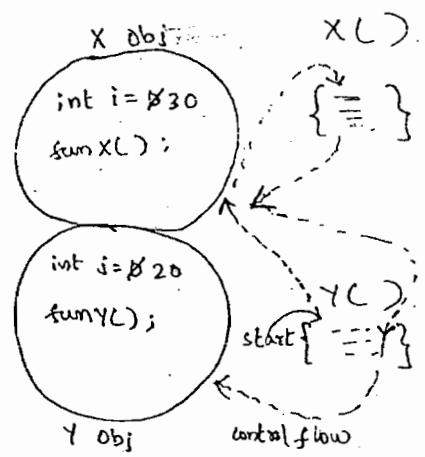
In inheritance concept, the class that extends another class is known as sub class and the class that is extended is known as super class.

Super class is also known as parent class
Sub class is also known as child class

(P.T.O)

following is a sample program which illustrates the concept of inheritance.

```
class X {  
    int i;  
    void functionX()  
    {  
        System.out.println(i);  
        // System.out.println(j); // error.  
    }  
}  
  
class Y extends X {  
    int j;  
    void functionY()  
    {  
        System.out.println(j);  
        System.out.println(i);  
    }  
    public static void main (String args[])  
    {  
        Y y1 = new Y();  
        y1.j = 20;  
        y1.i = 30;  
        y1.functionY();  
        y1.functionX();  
    }  
}
```



ates the

Theory behind the above program:

- As soon as JVM starts executing the Y.class file, it searches for the main method and the first statement inside the main is to create object of class Y.
- It creates object of class Y by loading the non-static members of class Y into the object and then loads the constructor of class Y into RAM and then starts executing it (if a constructor is not defined in class Y then it executes the default constructor).
- Now the control is inside the constructor of class Y. It first searches whether class Y is extending any other class or not.
- If class Y extends some other class (in this case it is class X) then it creates object of class X in association with object of class Y, loads the constructor of class X into the RAM.
- Now control is in constructor of class X and here also it searches whether class X is extending any other class or not. If yes, the same process as above repeats. If no, the control executes the statements of constructor of class X and then deletes it from the RAM (now object of class X is completely created) and then it reaches back to the calling place (i.e. constructor of class Y) executes the statements inside the constructor and then deletes it from the RAM (now object of class Y is ~~and~~ created completely).

From the above, we have three important points to understand.

- (i) Object of superclass is created first, before creating object of the subclass
- (ii) Super class object is created from the constructor of the sub-class.
- (iii) Super class object is created in association with sub-class object.

- Now, coming to the program, the next statement inside main is $y1.i = 20$; Thus upon the execution of this statement, the value of variable i inside object of class Y pointed by $y1$ changes from 0 to 20.

- The next statement is $y1.i = 30$.

NOW JVM searches for variable i inside the object of class Y. Since, it does not find it, it searches for i in the immediate super class object which is associated with the sub-class object. (Note that we do not have any direct reference to the super class object) once it finds the variable i it changes its value to 30.

Note that, we were able to change the value of a variable of an object whose reference is not directly available to us. This was possible just because of inheritance.

In a similar manner it deals with the statement $y1.funX()$, where $funX()$ is present in object of class Y.

Another important point is that, we have a statement
S. O.P(j) in function X(). Now when JVM encounters
this statement, and since funX() is operating through
object of class X, JVM searches for variable & j in
object of class. Since it does not find it, it searches
for j in the immediate super class object of X. Here
there is none. Then it generates an error.

Observe that variable j is available in the object of
class Y which is the sub-class object of class X.

NOTE: we can access non-static members of super class
object from sub-class object but we cannot access
non-static members of sub-class object from super class
object.

- In the above program, even though we have not created object of super class X manually, we were able to access its members. This was possible because, JVM in the background has created an object for class X (i.e. the super class).
- * Thus, the fundamental rule, i.e. without creating object of a class we can not access non-static members of that class is not violated.
- Since we are not manually creating the object of super class in the sub-class, we say that we are accessing non-static members of a class from another class without creating object of that class. But, in fact JVM is creating the object in the background.

NOTE: whatever oops concept is used, object of a class is created either directly or indirectly, through which we access the non-static members of that class.

Now, we can summarize the functions of 'new' operator as follows:

- (i) Reserves memory for the non-static members of the class file in the RAM.
- (ii) Loads all the non-static members to the instance.
- (iii) All un-initialized non-static variables would be initialized with their default values.
- (iv) Loads the corresponding constructor (defined / default) to the RAM
- (v) from the constructor, first it creates object of the super class (if there is any) and then the procedure continues with the super class object.

Function overriding:

It is the procedure of defining a functionality in the sub-class with the same signature of a function that is already existing in the superclass.

The following program illustrates the concept of function overriding.

(P.T.O)

```

class Abc
{
    int i, j;

    void functionC()
    {
        s.o.p ("Inside functionC() of class Abc");
    }

    void functionAC()
    {
        s.o.p ("Inside functionAC() of class Abc");
    }
}

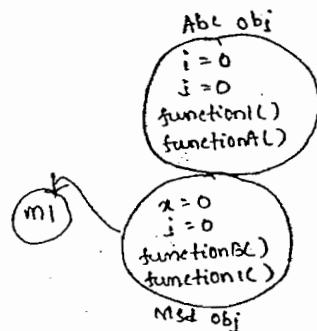
class Msd extends Abc
{
    int x, j;

    void functionB()
    {
        s.o.p ("Inside functionB() of class Msd");
    }

    void functionI()
    {
        s.o.p (x);
        s.o.p ("Inside functionI() of class Msd");
    }
}

public static void main (String args[])
{
    Msd m1 = new Msd();
    m1.j = 30; // j of object Msd will be changed.
    m1.functionC();
    m1.functionAC();
    m1.i = 115; // i of object Abc will be changed.
}

```

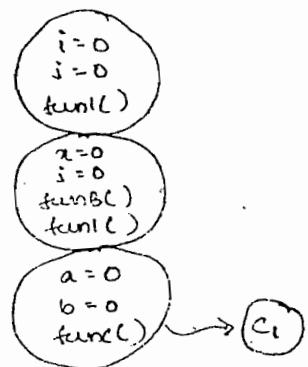


In the above program, function() is defined both in class ABC and class MCD. When m. function() is executed, function() present in MCD obj is executed. This is because local members have highest priority.

Hence, even though same variables/functions are present in both the objects, they are not available with the same priority. So there is no ambiguity.

Multilevel Inheritance

- A. - i, j, function()
- ↓
- B - x, i, funB(), funA()
- ↓
- C - a, b, func()



class Test

```
{  
    public static void main (String args[])  
    {  
        C c1 = new C();  
    }  
}
```

Now object of class C has properties of object of class B and object of class A.

NOTE: The priority of availability of objects to object of class C is first object of class B is available and then object of class A is available. This is multilevel inheritance.

red both
function() is executed.
priority.

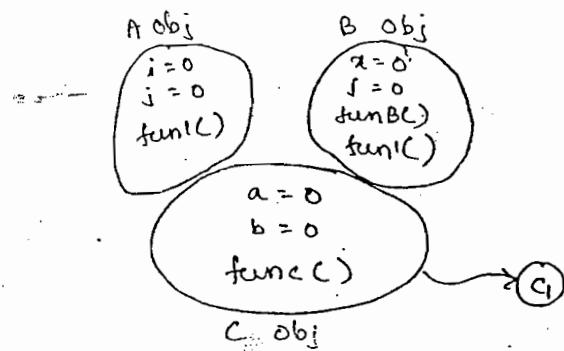
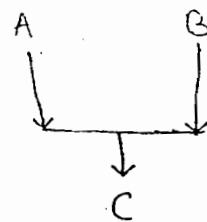
are
itable
iguity.

→ (c₁)

object

itable

Multiple Inheritance:



c₁ = new C();

c₁. function1(); // Ambiguous state.

In this case, two functions with same signature (function1()) and priority are available to the sub-class object. This leads to ambiguity.

∴ Java does not support multiple Inheritance.

Note: C++ supports multiple inheritance. But to avoid ambiguity concept of friend function is used. But this increases the complexity of the program.

In Java, anything which increases the program complexity is avoided.

Hence, Java does not support pointers and multiple inheritance.

NOTE: A super class should never extend a sub-class. If it does so, it leads to cyclic inheritance which is not supported by any programming language.

(P.T.O)

NOTE: In ~~in~~ java, every class extends ~~another~~ class.

when we mention it explicitly, the sub-class
extends the mentioned super class.

otherwise (i.e. if the class does not extend any other
class) then by default, the class extends Object class.

In other words,

whenever we use the 'extends' keyword, then
Object class will not be extended. Rather, the
mentioned super class will be extended.

If we don't use the 'extends' keyword, then the
class by default extends the Object class.

- Object of Object class would be the super most object
of object of any other class.

NOTE: Object class is the supreme class. It can not
extend any other class.

29.07.2006

Because of function overriding in multilevel inheritance, whenever
we call a function of the super class which is overridden in
the sub-class, it is always the sub-class function that will
be called, but not the function belonging to the superclass.

Q: Then how can you access a function of superclass when it is
overridden in the sub-class.

Ans: This is possible using the keyword 'super'

'super' keyword: it is a keyword using which we can point the immediate superobject of the current object.

Note: 'super' keyword can not be used in static blocks. This is because, super keyword points to the immediate superclass object of the current object and we know that static blocks will not work on objects.

Question: what are the two keywords, that should not be used in static blocks?

Ans: The keywords 'this' and 'super' are not supposed to be used in static blocks.

The following program illustrates the use of 'super' keyword.

```
class First
{
    int i, j;
    void functionA()
    {
        s.o.p("Inside functionA() of First class");
    }
}

class Second extends First
{
    int i, k;
    void functionA()
    {
        s.o.p("Inside functionA() of Second class");
        super.functionA(); // funA() of class First is called.
    }

    void functionI()
    {
        functionA(); // functionA() of class Second is called.
        super.functionA(); // funA() of class First is called
        super.i = 10;
    }
}
```

```
public static void main (String args[])
{
    Second s = new Second; // Three objects are created
                           // Second obj, First obj and
                           // obj of Object class.

    s.function1();
}

// super.functionA(); // error: (refer note on previous page)
```

consider the following program:

```
class X
{
    void function1()
    {
        S.O.P("Inside function1() of class X");
    }
}

class Y extends X
{
    void function1()
    {
        S.O.P("Inside function1() of class Y");
        super.function1();
    }
}

class Z extends Y
{
    void function1()
    {
        S.O.P("Inside function1() of class Z");
        super.function1();
    }
}
```

```
public static void main (String args[])
{
    Z Y = new Z();
}
```

(Previous page)

In the above program, all the three classes X, Y & Z have function1(). Now, the interesting point is that, without calling function1() of class Y from class Z we cannot call function1() of class X.

NOTE: therefore, it is obvious that, we can call functions of super class object only from a function of sub-class object

uses of keyword 'Super':

- (i) we can refer to the properties of immediate super class object from the sub-class object.
- (ii) we can explicitly call immediate super class constructors from the sub-class constructors.

NOTE: whenever a class extends another class, always the super class constructors are executed first.

The following program illustrates this concept.

(P.T.O)

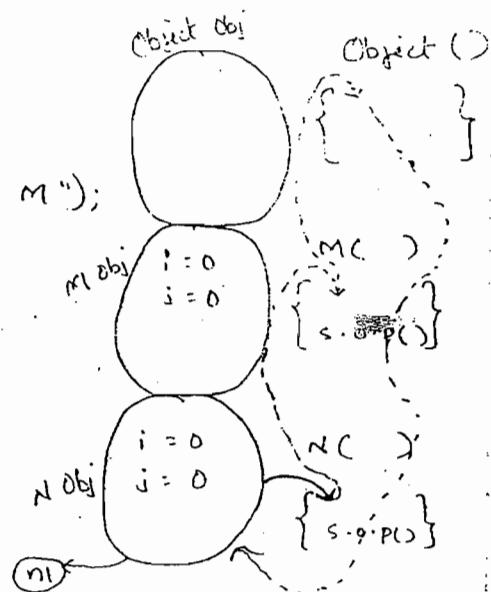
```

class M
{
    int i, j;
    M()
    {
        System.out.println("Inside M() of class M");
    }
}

class N extends M
{
    int i, j;
    N()
    {
        System.out.println("Inside N() of class N");
    }
}

public static void main (String args[])
{
    N n1 = new N();
}

```



we already learnt that, in every constructor we have a default statement (invisible to the user). This default statement is nothing but super(); As soon as the control enters the constructor of a class while creating its object, it first checks whether that class is extending any other class or not. If yes, it creates object of the super class by executing super(); statement in the current constructor.

This is the reason why, super class constructors are executed first.

consider the following program.

```
class M  
{ int i, j;  
  M (int x)  
  {  
    i = x;  
    s.o.p ("Inside M (int x)");  
  }  
}  
  
class N extends M  
{  
  int i, j;  
  N ()  
  {  
    s.o.p ("Inside N ()");  
  }  
  public static void main (String args[])  
  {  
    N n1 = new N ();  
  }  
}
```

In the above program constructor of class M (super class) is defined to accept integer arguments.

Now from the main, when object of class N is created, the constructor of class N will be executed. By default the first statement inside the constructor will be, to check if the class N is extending any other class or not.

In this case it is extending class M. So from the constructor of class N, now control moves upwards, object of class M is created and constructor of class M is

to accept an argument. But no argument is passed from the constructor of class A.

Hence, this program results in a compilation error.

calling super class const. by passing corresponding arguments from the sub-class constructor.

class I

```
{ int a,b;  
    I (int a)  
    {  
        a = a;  
        S·O·P ("Inside I (int a)");  
    }  
}
```

class J extends I

```
{  
    J ()  
    {  
        super();  
        S·O·P ("Inside J ()");  
    }  
}
```

public static void main (String args[])

```
{  
    J ii = new J();  
}  
}
```

FOOB
sup
the

Tiny

cla

§

S

P

C

C

C

C

C

C

NOTE

ov

st

bu

Q:

C

C

C

defined
accessed
error.
arguments

From this program it is clear that, whether we use 'super' key word or not in a constructor, this will be the first statement of every constructor to get executed.

Try this:

```
class A
{
    int i, j;
    void functionA()
    {
        System.out.println(i);
        System.out.println(j);
    }
    public static void main (String args[])
    {
        A a1 = new A();
        String s1 = a1.toString(); A@131571a
        System.out.println(s1);
    }
}
```

NOTE: `toString()` method belongs to `Object` class.

It does not belong to `String` class. In fact, it is overridden in the `String` class.

It is overridden not only in the `String` ~~but~~ class but also in many other classes such as `Integer` class etc.

Q: Why the ~~o/p~~ o/p of the `println` statement
S. O.P (s1) is address?

30.07.1

earlier we learnt that programmers generally use default constructor to initialize the instance variables. But now we should note that this is the secondary use.

The basic purpose of default constructor is to create super class object (in association with the subclass object) and to execute the super-class constructors. All this is possible just because of the default statement present in the default constructor.

NOTE: Whenever we use 'this' operator in a constructor to call another constructor of the same class, then 'super' can not be used in the calling constructor.

Consider the following program.

```
class A
{
    A()
    {
        this();
        S.O.P ("Inside constructor A()");
    }
    A(int i)
    {
        S.O.P ("Inside constructor A(int i)");
    }
    public static void main (String args[])
    {
        A a1 = new A();
    }
}
```

On the above program, class A has two constructors. One constructor is a zero argument constructor and the other is defined to accept an integer argument.

Now, we know that, for every class Object class is the super class. From one of the constructors of class A, object of Object class has to be created. Which constructor will do that?

It is clear that, from the zero argument constructor we ~~can't~~ are calling the constructor defined to accept integer argument using 'this' operator. Hence 'super' keyword will not be present in this constructor.

The other constructor A(int i) has only one statement and that too it is a println statement. So, the default statement will be inserted into this ~~sta~~ constructor, which creates object of Object class and constructor of Object class is executed.

→ Constructors of a class can execute constructors of only its immediate superclass, but not of supermost classes.

Ex:

class A

```
{  
    A (int x)  
    {  
        s.o.p ("Inside A(int x)");  
    }  
}
```

class B extends A

{

B();

{

s.o.p("Inside B()");

}

}

class C extends B

{

C();

protected {

super(b);

}

public static void main(String args)

{

C c = new C(); // Gives a compilation error.

}

}

Explanation: in the above example, constructor of class C can execute only the constructor of class B and it cannot execute the constructor of class A.

So, in constructor C(), when super(b); is encountered control checks for a constructor in class B (\because B extends A) which is defined to accept an integer argument. It finds none and hence generates a compilation error.

(Q. 4)

NOTE: if we had functions in place of constructors, then we do not have any problem i.e. we can call functions of super most classes from the bottom-most sub-classes.

Therefore, it is worth noting that, calling functions using super keyword is entirely different from calling constructors using super keyword.

→ A simple blind rule is that, the first statement of every constructor would be either 'this' or 'super'.

→ In Java, every keyword is defined for one specific functionality. But there is one keyword, whose functionality differs depending upon the situation where it is used.

That keyword is 'final'

use of the keyword 'final':

(i) When we use the keyword 'final' before a class, it restricts the properties of inheritance for that class.
i.e. final classes can not have sub classes.

Ex: final class A

{
Now we cannot extend class A
to any other class.
}

But class A can extend other classes. i.e. for example we can write "final class A extends B".

This is the reason, why, we can not extend String class, System class and many other classes of the JDK as they are declared as final.

(ii) Whenever we use final keyword before the declaration of a variable, then that variable becomes a constant. Its value can not be changed anywhere in the program.

Ex:

```
final class A
```

```
{
```

```
    final int x = 115;
```

```
    void functionA()
```

```
{
```

```
    System.out.println("Inside functionA()");
```

```
}
```

```
}
```

```
class Test
```

```
{
```

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

```
{
```

```
    A a1 = new A();
```

```
    System.out.println(a1.x);
```

```
    a1.x = 10; // error: constants cannot be modified.
```

```
}
```

```
}
```

NOTE: we can create objects of final classes from other classes as we do for normal classes.

In the above program, since the variable x is constant to the instance, it is known as instance constant.

On a similar manner, we can have local constants i.e. we can define a constant within a block i.e. function.

declaration

constant.

the program

Ex :

class Z

{ void function()

{ final int i=10;

System.out.println(i);

i = i+1; // error: i is a constant in the function
and can not be modified.

}

}

we can even define parameters of a function as constants.

Ex : void function(final int i)

{

System.out.println(i);

}

Now inside the function, we can not have any statements which changes the value of i (ex: i=i+1 etc. is not possible)

(iii) when we declare a function as final in a class, then it can not be overridden in the sub-classes of that class.

Ex :

class Z

{

final void funZ()

{

s.o.p("inside funZ()");

}

}

```

class A extends Z
{
    //void function2()
    {
        s.o.p("hai");
    }
}

```

This is not permitted because final functions of super-class cannot be overridden in sub-classes.

(**) The keyword 'finally':

It is similar to the keyword 'final'. It is used in exceptions.

*** super class reference - sub class object rule

This rule states that, to the reference of any super-class, we can assign any object of any sub-class.

Ex:

```

class A
{
    int i, j;
    void function1()
    {
        s.o.p("Inside function1() of class A");
    }
    void functionA()
    {
        s.o.p("Inside functionA() of class A");
    }
}

```

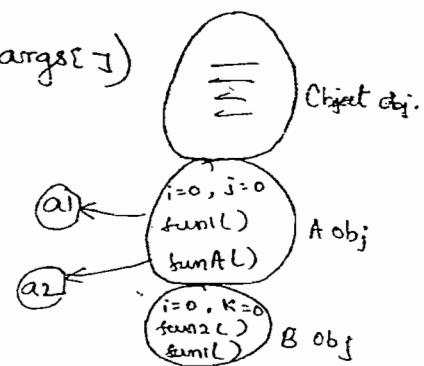
class B extends A

```
{  
    int i, k;  
  
    void function2()  
    {  
        S.O.P("Inside function2() of class B");  
    }  
}
```

```
void function1()  
{  
    S.C.P("Inside function1() of class B");  
}
```

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

```
    A a1 = new A();  
  
    A a2 = new B();  
  
    a2.function1();  
    a2.functionA();  
  
    // a2.function2(); //error.  
}
```



In the above program, inside the main, we are creating objects of classes. We do not have any confusion with the first statement `A a1 = new A();`

But in the second statement `A a2 = new B();`, we are assigning object of class B to reference of class A which violates the rule that object of a class cannot be

unary operator reference of derived class.

But this statement works properly and it does not generate any error. Why?

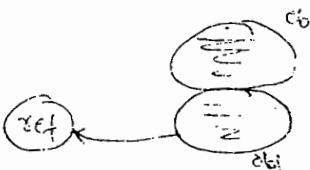
The reason is simple. Since class B extends class A object of class A will be created in association with object of class B. Thus, even though when we use `A a2 = new B();` the object of class A will be assigned to reference a2, but not object of class B.

This is the theory behind the statement.

This statement results in a failure i.e. it generates an error if class B does not extend class A.

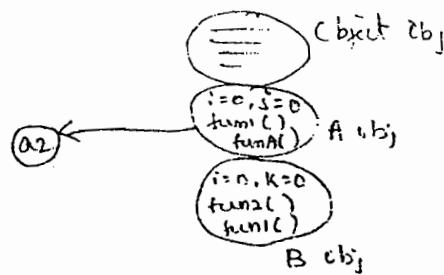
- Another interesting point to be noted in this program is, till now we have seen programs where a reference points to an object and that object is associated with the object of its super class.

This is something as shown in the figure.



But in the current example, we have a reference pointing to an object which is associated with objects of its super class as well as sub-class.

It is illustrated in the figure.



The important concept behind this program is super class reference - sub class object rule implementation.

We have to remember two important points here.

(i) When we call a function of an object through a reference, first JVM checks whether the function is present in the object or not.

If the function is present in the object, then with JVM searches whether the object has any sub-object or not. If the object has a sub-object, JVM searches whether the sub-object has any other further sub-objects. In this way it reaches the sub-most object. Now it finds for the function in the sub-most object. If it is found, the function is executed, if not, control searches for the function in the immediate super class object and so on.

(ii) When we call a function of an object through a reference, first JVM checks whether the function is present in the object or not.

If the function is not present in the object, then JVM searches for the function in the immediate super class object. In this way, if the function is not found even in the super-most object, then it generates a compilation error.

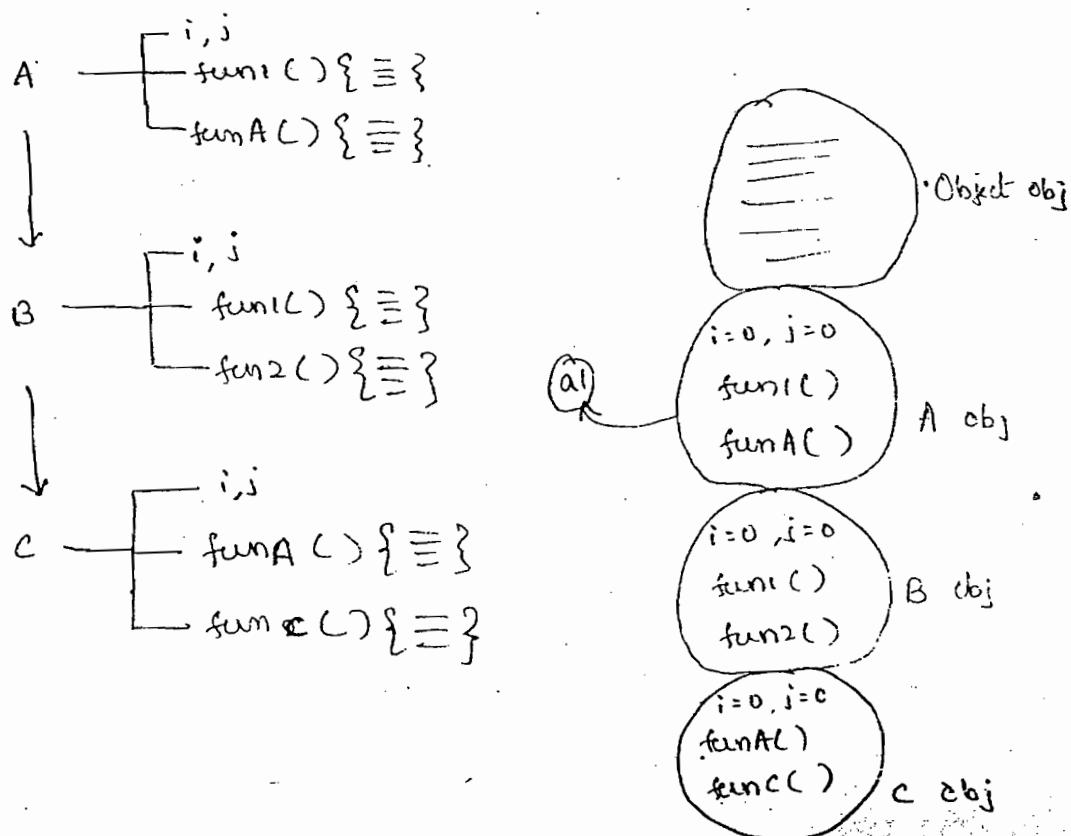
If the function is found in either the pointed out least one object or any of the super objects, then the process described in second paragraph of (i) continues.

NOTE: when we call a function which is overridden in three classes viz class A, class B, class C where B extends A, and C extends B, then according to (i) above, the overridden function in the sub-most object gets highest priority and it is executed first.

on the above program, when we write `a.function()` then according to (i) above, `function()` present in object of class B will be executed.

- when we write `a2.function2()`, according to (ii) it gives a compilation error.

for more clarity, consider the following example.



- verridden
where
according
the
is
function()
event
to(ii)
ject obj
bj
- A. a1 = new A(); // Address of object of class A is assigned to reference a1.
 - C c1 = new C();
 - B b1 = new B();
 - a1.function1(); // function1() of class B is executed.
 - a1.functionA(); // functionA() of class C is executed.
 - a1.function2(); // compilation error.
 - a1.functionC(); // compilation error.
 - c1.functionC(); // functionC() of class C is executed.
 - c1.functionA(); // functionA() of class C is executed.
 - c1.function1(); // function1() of class B is executed.

NOTE: The above theory works only for functions but not for variables.

This is because super class reference - sub class object rule works completely depending upon the concept of overriding and only functions can be overridden. we can not override variables.

Ex: a1.i = 25; // i present in object A will be changed to 25.
b1.j = 40; // j present in object B will be changed to 40.

01.08.2006

Dynamic polymorphism: out of some functions with the same name, same arguments, scattered in different classes, (function overriding) which function has to be executed is decided at the runtime.

Q How can you implement dynamic polymorphism using function overriding?

The following program illustrates the concept of dynamic polymorphism using function overriding.

A - funA()

B - funA()

C - funA()

```
class A
{
    void functionA(int i)
    {
        s.o.p("Inside functionA() of class A");
        int j = i * i;
        s.o.p("square of " + i + " is : " + j);
    }
}
```

class B extends A

```
{
    void functionA(int i)
    {
        s.o.p("Inside functionA() of class B");
        int sum = 0;
        for(int j=0; j <= i; j++)
            sum = sum + j;
        s.o.p("Sum of " + i + " numbers : " + sum);
    }
}
```

class C extends A

```
{
    void functionA(int i)
    {
        s.o.p("Inside functionA() of class C");
        int fact = 1;
        for(int j=1; j <= i; j++)
            fact = fact * j;
        s.o.p("factorial of " + i + " is : " + fact);
    }
}
```

```

class PolyTest
{
    public static void main (String args)
    {
        int i = Integer.parseInt (args[0].trim ());
        A a1 = new AC ();
        if (i >= 10 && i < 20)
            a1 = new BC ();
        if (i > 20)
            a1 = new CC ();
        a1.functionA (i);
    }
}

```

Now, just by seeing the program we cannot decide which functionA(int i) is going to get executed just at the compilation time.

The function to be called is completely dependant upon the value of i which is available only at run time.

This concept is known as dynamic polymorphism.

In dynamic polymorphism, which class object should be associated with the reference of some class would be decided at runtime. This is the key strategy of dynamic polymorphism.

In the above example, functionA (int i) is overridden in all the three classes - Because of this overriding, we are able to achieve dynamic polymorphism. So dynamic polymorphism is achieved only through function overriding.

NOTE : (i) The functionality of `toString()` method defined in `Object` class is to return address of the object on which it was called, in the form of `String` class object.

(ii) To the reference of `Object` class we can assign object of any class, because `Object` class is the super class of all classes.

- Statements present inside `println()` method of `PrintStream` class defined to accept object of `Object` class are shown below.

```
public void println (Object o)
{
    String s = o. toString();
    println(s);
}
```

consider the following program.

```
class A
{
    int i,j;
    void functionA()
    {
        s. o. p(i);
        s. o. p(i);
    }
    public void m (String args[])
    {
        A a1 = new A();
        String s1 = a1. toString();
        s. o. p (s1);
        s. o. p (a1);
    }
}
```

o/p
A@ 131571a
A@ 131f71a

Now, it is time to understand why we got the same o/p for both the println statements. It is as follows:

with the first statement inside main , A a1 = new A(); object of class A is created and its address is assigned to the variable a1.

In the next statement , String s1= a1. toString(). we are calling toString() method on a1 object. we know that, the functionality of toString() method is to return address of the object on which it was called , in the form of string class object.

So now s1 holds , the address contained in a1 in the form of string.



We know that when we pass object of a class to a println method, then it prints address, but when we pass ~~object~~ object of a string class as an argument to a println method, then it prints contents of the object pointed by the address (reference) which is passed as the argument.

This is the reason, why , we got the same output for both println statements in the above program.

- If we do not want the toString() method to return address, then we can override the toString() method and use it according to our requirement .

overriding toString() method

```
class B
{
    int x, y;
    public String toString()
    {
        String s1 = "This is my class B";
        return (s1);
    }
}
B b1 = new B();
s.o.p(b1); // "This is my class B" is printed
}
```

NOW, since we have overridden toString() method, there is no chance of s.o.p(b1); printing the address of object of class B.

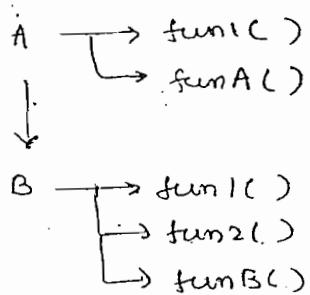
```
class B
{
    int x, y;
    public String toString()
    {
        String s1 = "x = " + x + "y: " + y;
        return (s1);
    }
}
s.o.p(b1); // "x:0 y:0" is printed
b1.x = 115;
b1.y = 205;
s.o.p(b1); // x:115 y:205 is printed.
```

This is the reason why we got 77 when we used
s.o.p(i) after Integer i = new Integer(77) in one
of the earlier programs. This is because Integer class
overrides toString() method in it.

03.08.2006

Typecasting super class reference into its equivalent sub-class reference

Consider the following scenario.



A a1 = new B();

Now, using the reference a1, we can call functions present in object of class A only and the functions of class A that are overridden in class B.

But we can not call functions of only class B.

To overcome this problem, we have the concept of typecasting the super class reference into its equivalent sub-class reference.

i.e. B b1 = (B) a1; [Here A is superclass, B is sub-class].

Now we can write,

b1.funB(); b1.fun2(); . These statements i.e. the function calls are valid only after typecasting the super class reference into its equivalent sub-class reference.

The
Now.
obj
cl
This
into
Object
refer
Now
ptree

NOTE
eve
da
ar
el
we
int
we
(i)
(ii)
(iii)
(iv)

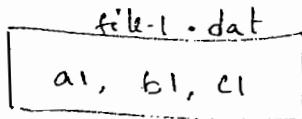
following is a common example which makes use of this typecasting concept.

In Java, files are created in order to store the objects of classes. There is a function by name `writeObject()` which accepts object of a class as argument and writes the state of the object (i.e. data inside the object) into the file:

Let us assume we have three objects `a1, b1, c1` of three classes A, B, C respectively.

Now by calling `writeObject(a1)`, state of the object `a1` is written into the file. By `b1` and `c1`.

Now, let us assume `a1, b1` and `c1` are written into a file



There is another method by name `readObject()` using which we can retrieve the objects (in fact the data inside the objects) present in the files.

Since `readObject()` returns object of a class, it should be defined to return object of any class.

Obviously, we can understand that return data type of `readObject()` method is object of `Object` class.

$\therefore \text{Object } o = \text{readObject();}$

This is something similar to `A a1 = new A();`

Let us assume `readObject()` is returning `a1` object.

The equivalent statement is Object o = new A();

Now, using O object we can not read elements of
Object of class A. (As mentioned in the beginning of this class)

∴ we should be able to represent objects of undefined
classes, in the form of Object of Object class.

This is possible only by typecasting superclass reference
into its equivalent subclass reference i.e. typecasting
Object class reference into its equivalent class A
reference.

i.e. A a1 = (A) o;

Now using the reference a1, we can access elements
present in Object of class A.

NOTE: As we were not able to typecast each and
every primitive datatype into its equivalent primitive
datatype, similarly, we can not typecast reference of
any class into its equivalent reference of any other
class.

We can typecast only the reference of a superclass
into its equivalent reference of a sub-class.

We can classify typecasting into four categories as shown below.

(i) Primitive data type - primitive datatype (Only selected types)

(ii) reference of a superclass - reference of a sub-class.

(iii) reference of an interface - reference of a class

(iv) reference of an interface - reference of an interface.

→ Late Binding and Early Binding:

Consider the following scenario.

Assume that some person Mr X has created a class with some statements which is responsible for the creation of a GUI (Ex: a submit button submit).

```
class MyButton {
```

```
{ myButton()
```

```
{
```

==== } statements responsible for creation of the GUI.

```
}
```

```
}
```

Let us assume that we are using this MyButton class ^{object} in our program.

```
class Test
```

```
{ void fun1()
```

```
{ == }
```

```
p.s.v. main (String args)
```

```
{ Test t1 = new test();
```

```
MyButton b1 = new MyButton() → [submit]
```

~~then~~ b1.click();

```
}
```

```
}
```

Now, the requirement is, when we click the button, some specific functionality should be performed. Here, in this case

let
but
NOTE:
;
counter
button
for
the

Now
fur
The
sho
may
click

now,
Mr
he
to
to
the

Let us assume that the specific functionality is nothing but execution of the statements present inside function().

NOTE: we can call fun() using t1.fun() but now it is not the point of concern.

contents of function() should be executed only when the button is clicked. So, for this, we should have another function in MyButton class which recognizes the click of the button. Let that function be as follows.

```
void findclick()
{
    wait for the button to be clicked
    if (button is clicked)
        Execute statements of function();
}
```

Now, the question is what should be the body of function()? what should be its functionality?

The answer is, the user who uses the MyButton class should define the functionality of function(). Each user may need to perform a different action upon the clicking of the button.

Now, if we analyze the entire scenario, it is as follows.

Mr X has created a class MyButton and in that he has a function function() whose body is not known to him. The body of the function() changes according to the need of the users who use his class in their programs.

It is something like, calling a function whose functionality is not known during compilation time.

Now, when the button is clicked, 'bt1.onclick()' is called and from that function, a specific action (executing func()) in this case) should be performed. That means, body or functionality of function() should be defined dynamically i.e. at run time.

[Realtime Ex: Filling a reservation form in a railway station. The form is printed already. But the details are entered only during the process of reservation. The form is common to all, but the reservation changes depending upon the details entered in it.]

* The concept of binding the function body to the function definition during runtime is known as

Late Binding

If the function body is binded with the function definition at the compilation time, then it is known as

Early Binding

NOTE: we cannot define a function in a class without body. It should have at least zerobody.

→ Difference between Late Binding & Dynamic polymorphism.

In dynamic polymorphism the function to be executed is decided at run time but note that by the time of compilation the function has a body.

But, in Late binding, the body of the function itself is going to get associated with the function definition during run time.

NOTE: Late binding is highly dynamic than anything else in java.

Interfaces:

- we implement Late binding through interfaces in java.
- class is like a structure and it can be called as a fully implemented structure (providing bodies for every definition)
- interface is a fully unimplemented structure i.e. no function definition has a body inside an interface. The function will not have atleast zero body.
- Just as we define classes using the keyword 'class' we define interfaces using the keyword 'interface'

Ex: interface xyz Xyz.java

```
{ public void funx();  
    public void funy();  
}
```

NOTE: Every definition inside an interface should be public.

Q when we compile Xyz.java we get .class file even though the functions funx(), funy() have no body.

03.08.2006

There are three types of structures in java, as given below.

- (i) fully implemented structure — class
- (ii) fully unimplemented structure — interface
- (iii) partially implemented/unimplemented structure — Abstract class.

The job of the java compiler is to convert any valid java structure (source code) into its equivalent byte code.

At the bottom of the previous page, we have a file Xyz.java
it is a valid java structure. Hence it got compiled and
Xyz.class file was produced.

Another example

```
interface Xyz
{
    public void functionX();
    public void functionY();
}

class Abc
{
    ===
}
```

Let us save the above program as Xyz.java. Now when we compile Xyz.java we get two .class files —

Xyz.class and Abc.class. This is because ~~we know~~ every valid java structure can be converted into its equivalent .class file and one .class file represents only one java structure. In this program we have two ~~two~~ ~~file~~ and java structures and hence, two .class files.

- As we can extend one class to another class, we can implement an interface to a class using the keyword 'implements'.
- we should keep in mind that whenever we implement an interface to a class, we have to provide bodies for the definitions present in the interface, inside the class that we are implementing the interface.

- References can be defined for any valid java structure but the corresponding object can be created only for classes. In other words, "we can define references in java for everything which can be represented in the form of a .class file, but the corresponding objects can be created only for classes"

Ex :

```
class A implements Xyz
{
    public void functionX()
    {
        // ...
    }

    public void functionY()
    {
        // ...
    }
}
```

| Defining bodies inside the
class for the definitions inside
the interface, where class implements
the interface.

```
public static void main (String args[])
{
    Xyz a1 = null; // we can assign null to the reference of
                    // an interface.
```

```
    Xyz a2 = new Xyz(); // compilation error: new operator
                        // cannot be used with interfaces.
```

```
    A a1 = new A();
```

```
    a1.functionX(); a1.functionY();
```

```
}
```

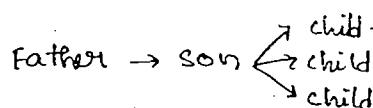
Q: In the above program, we can even directly use the function X() and function Y() inside class A and call them by creating object of class A. Then, what is the use of implements 'keyword'?

Auf: Using 'implements' keyword, we can achieve late binding.

- Difference between 'extends' and 'implements'

- Using 'extends' keyword we get properties of one structure to another structure.
- Using 'implements' keyword we provide bodies for function definitions inside a structure (interface).
- In simple terms, extends means getting and implements means giving.
- Therefore functionality of 'implements' is quite opposite to that of 'extends'
- Hence, 'implements' is against 'inheritance'

A more general example is as follows:



Father is super class
Son is sub class
child-1, 2, 3 are interfaces.

Son has to extend from father and can implement to all children.

** This is the reason, why, we can extend only one class but can implement any no. of interfaces.

Example

```

class B extends L implements I, M
{
    public void functionX() { }
    public void functionY() { }
    public void functionM() { }
    public void functionN() { }

    public static void main (String args[])
    {
        B b1 = new B();
        b1.functionX();
        ;
        b1.functionN();
    }
}

```

Defining bodies for the definitions in the interfaces funM() & funN() are the definitions of the interface M and.

Q what is the situation if there is a common definition in two interfaces and the two interfaces are implemented by the same class.

```

interface Xyz
{
    public void functionX();
    public void functionY();
}

interface Isk
{
    public void functionI();
    public void functionX();
}

```

contd... (P.T.O)

```

class Idemo1 implements Xyz, Ijk
{
    public void functionX() { }

    public void functionY() { }

    public void functionI() { }

    public static void main (String args[])
    {
        Idemo1 d1 = new Idemo1();

        d1.functionX();
        d1.functionY();
        d1.functionI();
    }
}

```

in the above program, Idemo1 implements Xyz and Ijk. functionX() is defined both in Xyz and Ijk. But we need not define functionX() two times, ^{in our class} as a part of defining bodies for function definitions present in the interface.

NOTE: This prevents us from violating the rule that we can not have two functions with same name and same signature and equal priority inside a class.

*** To the reference of an interface we can assign object of any class that implements that interface.

```

Xyz x1 = new A(); // possible only if class A implements Xyz

x1.functionX();
x1.functionY();

```

using the reference `al`, we can call the functions that are present in `Xyz`.

- All rules regarding super class reference - subclass object are applicable here also.

- Using the above rule (i.e. object of a class can be assigned to the reference of an interface, if the class implements that interface) we can achieve late binding.

For this let us consider the `MyButton` class example which was discussed earlier.

```
class MyButton {  
    MyButton() {  
        { }  
    }  
    public void findclick (Interf i) {  
        {  
            wait for the click  
            if (button is clicked)  
                i.functionI();  
        }  
    }  
}
```

```
interface Interf  
{  
    public void functionI();  
}
```

when we compile `MyButton.java` file, we get two .class files, but we do not see any body associated with `functionI()`.

contd... (P-T-0)

Remember that MyButton class and Interface
are supplied by Mr X. Now he has to give to us
the documentation (which indicates the functions present
in both the .class files along with other information)
along with the .class files.

To call function findclick(), we have to pass an
argument. The function parameter is reference of
an interface. The argument is object of a class.
Clearly, in this case, we need to assign object of
a class to the reference of an interface. This is
possible only if the class implements that interface.

** Therefore, whenever it is required to assign object of
a class to the reference of an interface, then we
make use of the 'implements' keyword.

```
class A implements Inter1  
{  
    public void functionI()  
    {  
        // Implementation  
    }  
    public static void main(String args)  
    {  
        A a1 = new A();  
        MyButton b1 = new MyButton();  
        b1.findclick(a1);  
    }  
}
```

Now, an object is assigned to interface reference i.
Now i. functionI() will be executed through the object g
class A (i.e. a1) which is passed as an argument
to the function findclick (Interface).

That means, we are providing body for the function
~~defined~~ definition in the interface, through class A
dynamically (i.e. at run time) which is nothing but
late binding.

In this way, we can achieve late binding using
'implements' keyword.

04-08-2006

observe the following program.

```
interface Xyz
{
    public void functionX();
}

class Idemo2
{
    void fun1(Xyz a1)
    {
        s.o.p("Inside fun1() of Idemo2");
        a1.functionX();
        s.o.p("end of fun1() of Idemo2");
    }
}
```

worl... (P.T.O)

class A implements Xyz

```
{  
    public void functionX()  
    {  
        System.out.println("U R in funX() of class A");  
    }  
}
```

class Test

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

```
{  
    Idemo2 i = new Idemo2();
```

```
    A a1 = new A();
```

i.function1(a1); // funX() will be executed in association
with statement present in function1()

// a1.functionX(); // only funX() is executed

```
}
```

```
}
```

In the above class Test, we are creating object of
class Idemo2 and on that object we are calling
function1(), by passing object of class A (i.e. a1) as
argument where this argument is received by the
parameter - a reference of interface. Upon this object
we are calling functionX() inside function1().

Instead, we can directly create object of class A
and call functionX() on that object a using
a1.functionX(). what is the difference between

there two processes?

Explanation:

when we use `i.function1(a1)`, we are calling `functionx()` on object of class A (i.e. `a1`) from `function1()`. That means, `functionx()` will be executed in association with the statements present in `function1()`.

But, when we call `functionx()` directly on object `a1` (i.e. `a1.functionx()`) only `functionx()` will be executed which may not be useful for us.

— As we can define functions whose parameters are interface references, we can think of defining functions which return interface reference.

The following example illustrates this concept.

```
class Idemo3
{
    static Xyz getXyz()
    {
        Xyz x1 = null;
        x1 = new A(); // class A implements Xyz.
        return (x1);
    }
}
```

```
}
```

class A implements Xyz
{
 public void functionx()
 {
 \Rightarrow
 }
}

155

```
interface Xyz
{
    public void functionx();
}
```

contd... (P.T.O)

```

class Test
{
    public static void main (String args[])
    {
        XYZ x1 = Idemo3. getXyz();
        x1.functionX();
        XYZ x2 = new A();
    }
}

```

In the above program, `getXyz()` method returns, reference of an interface. But that reference contains object of a class.

The above scenario can be clearly observed in two important statements of JDBC.

```

Connection con = DriverManager . getConnection (url);
Statement st = con. createStatement();

```

These two statements can be compared with the first two statements inside `main()` of class `Test`.

```

XYZ x1 = Idemo3. getXyz();
x1.functionX();

```

`Connection con` \Leftrightarrow `XYZ x1`

`DriverManager . getConnection ()` \Leftrightarrow `Idemo3. getXyz();`

`con. createStatement ()` \Leftrightarrow `x1.functionX();`

can be a reference of connection interface. As we can assign objects of a class to interface reference (discussed earlier) we are assigning the object of a class returned by getconnection() method of DriverManager class.

* if getconnection() method returns object of a class, then what is the name of that class?

Ans: Actually getconnection() method returns object of an Anonymous inner class.

* Anonymous inner classes are those classes, which do not have any name. But we can create objects of such classes.

(Creating objects of Anonymous objects will be discussed later)

Some more points regarding interfaces:

- Apart from function definitions, we can define constants within an interface
- even if we do not define a variable as public static final inside an interface, by default any variable defined inside an interface would be public static final.
- we can not use static keyword before function definitions inside the interfaces
- static constants inside the interface should be initialized by the programmer (at least with zero) because default initialization is not provided.

The following program illustrates the above key points:

```
interface Abc
{
    public static final int x=20;
    int y=30; // it is valid because by default it would
               // be public static final int y=30;
    public void functionA();
}

// public static void funB(); // error - function definitions
                           // cannot be static.

// int y; // error: static constants should be initialized.

}

class ITest2
{
    public static void main (String args[])
    {
        System.out.println(Abc.y); // 30 is printed
        int i = Abc.x;
        System.out.println(i); // 20 is printed
    }
}
```

Consider the following case:

class A implements XYZ

class B extends class A

Now what can be the features of class B.

```
interface Xyz  
{  
    public void functionX();  
}
```

class A implements Xyz

```
{  
    void functionX()  
    {  
        // Implementation  
    }  
}
```

```
Xyz x1 = new A();  
x1.functionX();
```

}

class B extends A

```
{  
    void functionB()  
    {  
        // Implementation  
    }  
    /* void functionX()  
    {  
        // Implementation  
    } */  
}
```

class Test

```
{  
    A a1 = new B();  
    Xyz x1 = new B(); // This is also possible.  
    x1.functionX();  
}
```

(P.T.O)

In the above program, we can assign object of class B to the reference of interface i.e. `Xyz x1 = new B();`

This is possible because, even though class B does not implement Xyz, it is extending class A which implements Xyz.

When we say `x1.functionX()`, ~~the~~ functionX() present in class A is executed. But when we override functionX() in class B (i.e. remove comment lines for functionX() in class B), then when we use `x1.functionX()`, functionX() of class B is executed.

- An interface can extend any no. of interfaces.

NOTE: A class can extend only one other class at a time.

contd... Book-2

05-08-200

- An interface can not implement another interface. This is because, whenever a structure implements an interface then that structure should provide bodies for all the function definitions inside that interface and we can not define function bodies inside an interface. Hence, an interface can not implement another interface.
- An interface can 'extend' any no. of other interfaces. Thus, by using the keyword 'extends' function definitions of one interface can be made available to another interface.
- Because of the above possibility, many people say that . we can implement concept of multiple inheritance using interfaces. Note that, even though we are extending , we are extending only the function definitions, but we are not inheriting the function bodies.

Ex:

```
interface Xyz
{
    public void functionX();
    public void functionY();
}
```

```
interface Ijk
{
    public void functionI();
    public void functionX();
}
```

```
interface Abc extends Xyz, Ijk
{
    public void functionA();
    public void functionB();
}
```

```

class Idemo4 implements Abc
{
    public void functionA() { }
    public void functionB() { }
    public void functionX() { }
    public void functionY() { }
    public void functionI() { }
    public void functionL() { }

    public static void main(String args[])
    {
        Abc al = new Idemo4();
        al.functionA(); al.functionX(); al.functionL();

        Xyz xl = new Idemo4();
        xl.functionX(); xl.functionY();

        Ijk il = new Idemo4();
        il.functionI(); il.functionX();
    }
}

```

NOTE: when a class implements an interface, then that class should provide bodies for all the function definitions available to that interface.

In the above example, class Idemo4 implements Abc. But Abc extends Xyz and Ijk. That means function definitions of Xyz and Ijk are available to Abc and since Idemo4 extends Abc, Idemo4 has to provide bodies for all the function definitions in Abc, Xyz and Ijk.

- if a function definition is defined in more than one interface and if we provide body for that function definition in our class which implements all those interfaces (directly or indirectly). Then, the same function body would be executed on different interface references.
- we can typecast an interface reference into its equivalent reference of a class.
In a similar manner we can typecast an interface reference into its equivalent reference of another interface.

Ex:

```

class Idemos
{
    public static void main (String args[])
    {
        Abc a1 = new Idemo4();
        Idemo4 d1 = (Idemo4)a1; // Typecasting interface reference to its
                                // equivalent reference of a class.
        XYZ x1 = new Idemo4();
        Idemo4 d2 = (Idemo4)x1;
        Abc a2 = (Abc)x1;
        Ijk i1 = new Idemo4();
        Abc a3 = (Abc)i1; // Typecasting interface reference to its
                            // equivalent reference of another interface.
    }
}

```

- 'public' is a keyword. It is an access specifier.
An access specifier specifies the accessibility of the contents of a block (or structure), which is outside that block.

Abstract classes:

A structure (class) which is partially implemented - partially unimplemented is known as an abstract class.

- When we use the keyword 'abstract' before a class, then that class becomes an abstract class.
- Since abstract classes are partially implemented - partially unimplemented structures, they can have function definitions as well as function definitions with bodies.

NOTE: Any function definition without a body, inside an abstract class would be by default 'public' even though we mention it or not.

- When we extend an abstract class to a normal class, then we have to provide bodies, in the class, for all the function definitions defined inside the abstract class.

NOTE: We can not create objects of Abstract classes directly using 'new' operator. Instead, we can define a reference to an abstract class and to that reference, we can assign object of a class which extends the abstract class.

Example:

```
abstract class Abc1
{
    public abstract void function1();
    void function2()
    {
        S.O.P("Inside fun2() of abstract class Abc1");
    }
}
```

```

    {
        public void function1()
        {
            ...
        }

        public void functionD()
        {
            ...
        }

        public static void main (String args[])
        {
            Demo1 d1 = new Demo1();

            d1.function1();
            d1.functionD();
            d1.function2();

            // Abs1 a1 = new Abs1(); // error: cannot create object
            // of Abs1 using new operator.

            Abs1 a2 = new Demo1();
            a2.function1(); // function1() of Demo1 is executed.

            a2.function2();
        }
    }
}

```

- An abstract class can extend other classes.
- It is obvious that when an abstract class does not extend other user defined classes, then, just like other classes, it extends object class.

(P.T.O)

... define an abstract class without abstract function definitions inside it. Hence, defining abstract function definitions inside an abstract class is not compulsory.

abstract class Abs2

```
{  
    public void functionA() // class Abs2 has no abstract  
    { == } // function definitions even though  
}  
it is abstract.
```

```
class Demo2 extends Abs2 // since funA() in Abs2 is  
{  
    void functionD() == } // not abstract, we need not  
    public static void main (String args[]) once again provide body for it  
    extending Abs2.  
}
```

```
{  
    Demo2 d1 = new Demo2();  
    d1.functionD();  
    d1.functionA();  
}
```

- we can define an abstract class only with abstract function definitions.

NOTE : when a class (subclass) extends an abstract class (super class) then the subclass should provide bodies for all the abstract function definitions ^{present} inside the abstract class (superclass). if the subclass does not

provide the bodies, then it should be declared as abstract. Else, we get a compilation error.

Ex:

```
abstract class Abs3
```

```
{  
    abstract void functionA();  
    abstract void functionB();  
}
```

```
abstract class Ademo1 extends Abs3
```

```
{  
    public void functionA() { }  
    public void function1() { }  
}
```

```
class Ademo2 extends Ademo1
```

```
{  
    public void functionB() { }  
    public static void main (String args)  
    {  
        Ademo2 a1 = new Ademo2();  
        a1.functionB(); a1.functionA(); a1.function1();  
        Ademo1 a2 = new Ademo2();  
        a2.functionA(); a2.function1(); a2.functionB();  
    }  
}
```

Since Ademo1 is extending Abs3, but it is not providing bodies for all function definitions in Abs3, Ademo2 is declared as abstract.

class Ademo2 is not abstract. So we can say which class is abstract. Note that Ademo1 has no abstract function definitions in it. But observe that, there is an abstract definition, ^{funB()} available to Ademo2 from Abs3 for which it (Ademo1) has not provided the body.

Now, since Ademo2 is extending Ademo1 and is not abstract, it has to provide body for the abstract function definition (i.e. funB()) of Abs3 which is available to it through its super class (i.e. Ademo1).

06-08-06
sunday

Abstract classes can extend ~~any~~ ^{other} abstract classes and they can even implement interfaces.

Ex interface XYZ
{
 public void functionX();
 public void functionY();
}

abstract class Ademo3 implements XYZ
{
 void functionA()
 {
 //
 }
}

NOTE: we can definitely implement an interface by not providing bodies to the function definitions present in the interface by placing the keyword 'abstract' before the class.

```

class Test extends Ademo3
{
    public void functionX()
    {   }

    public void functionY()
    {   }

    public static void main (String args[])
    {
        Test t = new Test();
        t.functionX();
        t.functionY();
        t.functionA();
    }
}

```

This is mandatory. Otherwise we have to declare class Test as abstract.

- objects of abstract classes cannot be created using 'new' operator, but they are created along with sub-class objects (i.e. when a class extends an abstract class).

Ex :

```

abstract class Abs1
{
    int x, y;

    Abs1 ()
    {
        s.o.p ("Inside const. of Abs1 ()");
    }

    abstract void function1 ();
}

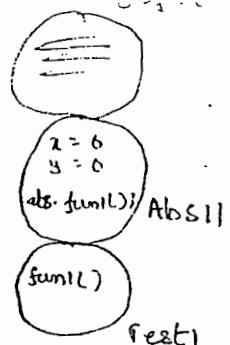
```

contd....(P.T.O)

```
{
    public void function1() { }

    public static void main (String args[])
    {
        Test1 t1 = new Test1();

        s.o.p(t1.x); // t1.x=0
        s.o.p(t1.y); // t1.y=0
    }
}
```



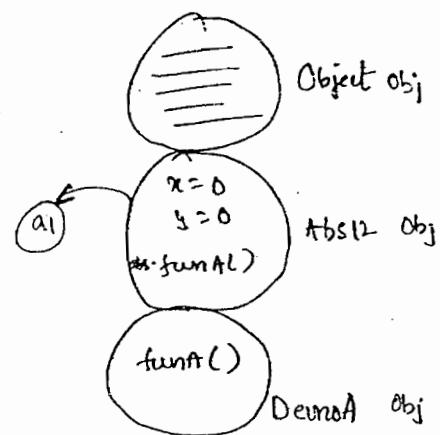
- we can define constructors in the abstract classes. The abstract keyword is applicable only to classes and functions but not constructors.
- if parameterized constructors are defined in abstract classes, we have to explicitly call the constructor by passing the arguments, just as we do for normal classes.
- Just as in the case of a normal class, even in the abstract classes also, the compiler will provide a default constructor if the class has no constructor defined in it.
- we can define both, fully implemented static functions and fully implemented non-static functions in abstract classes.

Ex:

```
abstract class Abs12
{
    int x, y;
    abstract void functionA();
    public static void main (String args[])
    {
        System.out.println("Inside main of class Abs12");
        Abs12 a1 = new DemoA();
        a1.functionA();
        System.out.println(a1.x);
        System.out.println(a1.y);
    }
}
```

class DemoA extends Abs12

```
{
    public void functionA()
    {
    }
}
```



In the above program, we are assigning object of a class to the reference of an abstract class. We are not directly creating object of abstract class. Hence it does not generate any compilation error.

All the static members are loaded into the RAM and now JVM searches for the main method and starts executing it. If we declare abstract functions as static, we get a compilation error.

NOTE: All the rules of inheritance are also applicable to abstract classes.

- Interfaces - Late binding
 - Abstract classes - partial late binding
 - ? - Early binding

— packages

Q : How can we bind related concepts logically?

Adv: By using packages.

Ex: c:1> pack1 x.java → package pack1;
 X-class
 Y-class
 public class X { }

- whenever we want to keep a class inside a package,
use the keyword public before the class.

Q: create a package by name pack1 which contains two class files.

Q4: first define the files X.java & Y.java as shown below

| | |
|---|---|
| <p>x.java</p> <pre>package pack1; public class X { public int x,y; public void functionX() { } } }</pre> | <p>y.java</p> <pre>package pack1; public class Y { public int i,j; public void functionY() { } }</pre> |
|---|---|

compile X.java and Y.java to obtain the .class files.
Now we have to paste these .class files explicitly in
the folder pack1. Also note that, we have to set the
path in the environment variable explicitly.

Now, this package pack1 can be used in other programs.

Ex:

```
import pack1.*;           [Test.java]
class Test                ↳ represents all .class files present in
{                           pack1.
    public static void main (String args[])
    {
        X x1 = new X();
        Y y1 = new Y();
        x1.functionX();
        y1.functionY();
    }
}
```

- The above mentioned process is one way of creating packages (i.e. explicitly copying the .class files into the package folder)

- The second method is as follows:

There is a dos command which inturn creates a folder in the name of the package and it places all the .class files in the package folder. It automatically sets the path for it.

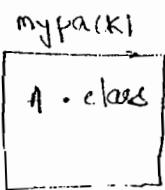
(P.T.O)

```

package mypack1;
public class A
{
    public int i, j;
    public void functionA()
    {
        /**
    }
}

```

A.java



Now, the following dos command is used in order to create the folder and place the .class file in it

c:\> javac -d . A.java
 ↳ represents current directory.

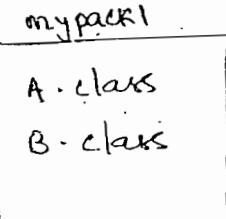
In this case, the .class file is placed inside the folder and will not be seen along with the .java file.

Ex:

```

package mypack1;
public class B
{
    public int i, j;
    public void functionB()
    {
        /**
    }
}

```



c:\> javac -d . B.java

NOTE: Since package mypack1 already exists, a new package will not be created. Instead, the existing package is opened and the .class file (B.class) is copied into it.

Rule: Even though we are relating class A and class B with the same package, we should not declare them in one source file, as shown below.

```
package mypack1;  
public class A  
{  
}  
public class B  
{  
}
```

This is not allowed.
one source file can have
only one public class.

Hence, it is not advisable to have a common source file for all the classes. Individual classes should have individual source files.

```
// import mypack1.*;  
import mypack1.A;  
class Test1  
{  
    public static void main (String args[])  
    {  
        A a1 = new A();  
        a1.functionA();  
    }  
}
```

Test1.java

C:\> javac -d . Test1.java ↴

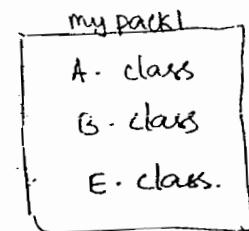
If we want to use
comment line for the first statement), then we have
to set the path in the environment variables explicitly
and it is mandatory.

08.08.06

- A file may contain both import and package keywords.
but always the package keyword comes first and then
comes the import statements.
- we can definitely import any no. of packages to the
current file but package keyword should be used only
once in a file.

Ex :

```
package mypack1;  
import java.awt.*;  
import java.util.*;  
public class E  
{  
    public void functionE()  
    {  
    }  
    public static void main (String args[])  
    {  
    }  
}
```



⇒ javac -d . E.java

Since mypack1 is already an existing package, the E.class
file will be directly copied into it. Note that we already
have A.class file and B.class file in the package mypack1.

**
Q what would be the first key word with which you start writing a java program.

Auf: package.

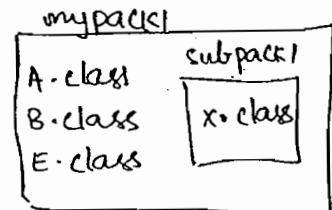
NOTE: In real time there would not be any files existing individually. All the files will be placed either in one or other packages.

Whenever a project is supplied to a client, it will be supplied in the form of a package. No file will be supplied individually.

- we know that, a folder can contain files as well as folders. package is also a folder. Hence we can have sub-packages inside packages.
- Assume that we want to create a subpackage by name subpack1 inside the package mypack1 and subpack1 should contain X.java file.

To achieve this we have to do the following.

```
package mypack1.subpack1;  
public class X  
{ public void functionX() { } }
```



C:\> javac -d . X.java ↴

(contd... (P.T.O))

Since we have mentioned `subpack1` in the `X.java` file, as soon as we compile it using that `javac -d . X.java`, the `subpack1` folder will be created inside `mypack1` and `X.class` file will be directly present inside `subpack1`.

- consider the following example.

```
import mypack1.*;  
class Test  
{ public static void main (String args[])  
{ E e1 = new E();  
 X x1 = new X(); // error.  
 }  
 }
```

we get an error saying that `X.class` is not available. This is because, in the import statement, `*` represents only files but not folders and `X.class` is present inside folder `subpack1` which is a sub package (sub folder) of the package (folder) `mypack1`.

- To avoid this compilation error, we should explicitly import the subpackage as shown below.

```
import mypack1.subpack1.*;
```

Since we are using * operator, we are supposed to set the path \$ in the environment variables for the sub packages also, as we do for the parent packages.

- Access specifier:

Access specifiers are keywords which mention about the accessibility of contents of a class, somewhere outside of that class.

There are four types of access specifiers:

(i) public (ii) private (iii) protected (iv) default

public, private and protected are keywords.

When none of these is used, then the access specifier of that particular statement or block would be default. (There is no keyword for the default access specification).

public keyword specifies that the accessibility is global.

private keyword specifies that the accessibility is confined to that class.

default: default is similar to public as long as the control is in the same package. Else default acts as private.

(P.T.O)

A program to illustrate the use of new operator.

```
class Access1
{
    public int i;
    private int j;
    public void functionA()
    {
        i = i + 1;
        s.o.p(i);
        s.o.p(j);
    }
    private void functionB()
    {
        i = i * i;
        j = j * j;
        s.o.p(i);
        s.o.p(j);
    }
    public static void main (String args[])
    {
        Access a1 = new Access1();
        a1.functionA();
        a1.functionB();
        s.o.p(a1.j);
    }
}
```

```

class Test
{
    public static void main (String args[])
    {
        Access a1 = new Access();
        a1.functionB(); // compilation error: funB() is private
                        . and can be accessed only with class Access
        s.o.p(a1.i); // i is public and is accessible
                      . every where.
        s.o.p(a1.j) // error : j is private.

        a1.functionA(); // no error, even though funA()
                        . is accessing private members.
    }
}

```

NOTE: we can get access to the private members
of a class ~~from~~ ^{through} the public members of the same
class , from other class.

Ex:

```

class Test extends Access
{
    public static void main (String args[])
    {
        Test t1 = new Test();
        t1.functionB(); // error.
        t1.functionA(); // no error since funA() is public.
    }
}

```

NOTE: we can not access private members of the
super class, with the subclass object.

Ex.

```
class Access2
{
    int i, j;
    void functionA()
    {
        ...
    }
}
```

```
class Test2
{
    public static void main(String args[])
    {
        Access2 a1 = new Access2();
        a1.functionA();
    }
}
```

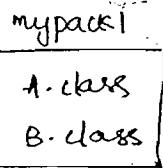
when we don't mention any access specifier to the content of a class, then its accessibility is default.

Q Then what is the difference between default & public?

Ans: default would be similar to public as long as the .class files are within the same package (i.e same path)

outside the packages the default accessibility would be similar to private.

Ex:



Let the contents of A.java be as follows:

```
package mypack1;
public class A
{
    public int i;
    int j;
}
```

Now, from B.class file, we can access variable i of class A because both the .class files are in same package. Access specification of i is default and within the same package, it is equivalent to public.

```

class Test
{
    public static void main (String args[])
    {
        A a1 = new A();
        s.o.p (a1.i); // i is public
        s.o.p (a1.j); // error
    }
}

```

Error occurs because, accessibility of j is default and outside the package it is private. So when we try to access it from Test-class which is outside the package, we get a compilation error.

- The above rules of accessibility will be applied even to classes.

Ex:

```

package mypack1;
class B
{
    public int i, j;
    public void funB()
    {
        { } // error
    }
}

```

```

import mypack1.B;
class Test1
{
    public static void main (String args[])
    {
        B b1 = new B(); // error
    }
}

```

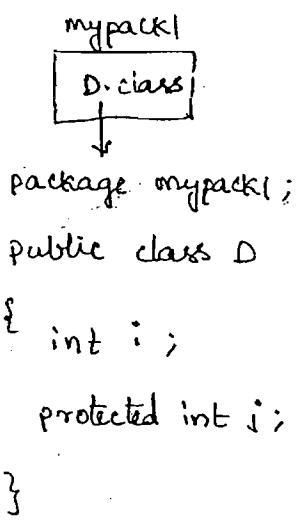
The error occurs because access spe accessibility of class B is default and we are trying to create object of class B from class Test1 which is outside the package.

protected: The behaviour of this access specifier is also similar to default but with a small difference.

We know that default would be acting as private outside the packages. Same is the case with protected. But the difference is, protected would behave as public with respect to subclass objects even outside the packages.

This differentiates protected from default.

Ex:



Demo.java

```
import mypack1.D;
class Demo
{
    public static void main (String args)
    {
        D d1 = new D();
        s.o.p(d1.i); //error
        s.o.p (d1.j); //error
    }
}
```

We can not access `i` and `j` of `D.class` file from `Demo.class` file because `Demo.class` is outside the package `mypack1` and default and protected - both will act as private outside the package.

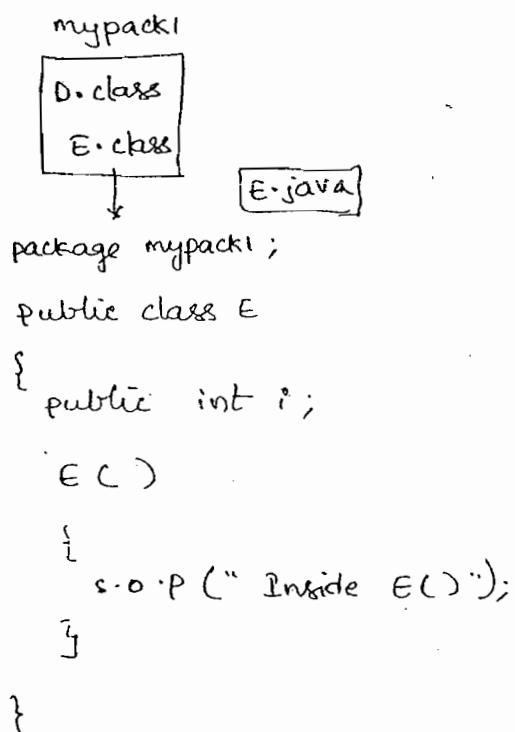
Now, if at all class `Demo` extends class `D`, then we can access variable `j` (protected) of `D.class` through object in `Demo.class` because protected would behave as public with respect to subclass objects.

Ex :

```
class Demo extends D
{
    public static void main (String args[])
    {
        D d1 = new D ();
        Demo d2 = new Demo ();
        s. o. p (d2.i); // error
        s. o. p (d2.j); // This is possible.
    }
}
```

Access specifiers and constructors:

consider the following program.



Test.java

```
import mypack1.E;
class Test
{
    public static void main (String args[])
    {
        E e1 = new E(); // error
        s.o.p (e1.i);
    }
}
```

we get an error while creating object of class E in Test.java because, constructor defined in class E has no access specifier which means it's accessibility is default and we know the behaviour of default outside the packages.

use the constructor, as you know, we have to make
class can not be created.

So, to avoid the errors, it is a must to define the
constructor as public.

NOTE: Access specifier of the default constructor that would
be provided by the compiler is always public.

Therefore, whenever we use packages, it is necessary
to define the constructors (if any) as public of all the
classes as public, so that it would be possible to
create objects of those classes even outside their
package. In this case, the class should also be public.

Question: Can you mention a class as private, protected, static?

Ans(i) No, we can't mention a class as private, protected, static
if the class is an outer class.

(ii) Yes, we can mention a class as " " "
if the class is an inner class.

- An outer class can almost be defined as public/default
- we can mention a constructor as public, private,
protected, default. It can not be static.
- we should not mention a constructor ^{of class} as private if
object of that class needs to be created outside
that class.

- So far we have the `System` class. we cannot create object of `System` class using `new` operator because all the constructors defined in `System` class are private.

∴ The constructors of `System` class are private, its object can not be created and without object we cannot access the non-static members. Because of this reason, everything defined inside the `System` class (except the constructors) ~~are~~ is declared as static.

- For example, if a class has a non-static member and a private constructor (definitely object can not be created) then how would you access the non-static member?

Ans: In such situations, we need to define a function inside the class as public static which returns object of that class. Such functions/methods are known as Factory Methods.

NOTE: Factory Methods are always public and static.

- Factory methods are used, when we want to access non-static members of a class (whose constructors are defined as private) from another class.
- The following example illustrates the use of factory methods.

contd..(P.T.O)

```

class Demol
{
    public int i;
    private Demol()
    {
        s.o.p("Inside constructor of Demol()");
    }
    public static Demol function()
    {
        Demol d = new Demol();
        return(d);
    }
}

class Test
{
    public static void main(String args[])
    {
        Demol d = Demol.function();
        s.o.p(d.i);
    }
}

```

In the above program, in class Demol, the function
`public static Demol function()` is a factory method
which returns the object of the class in which it
is present.

- Access specifiers are not applicable to the local variables because the scope of local variables is confined to the method or function in which they are present.

Ex : ~~outside~~ void function()

```
i int i;  
s.o.p(i);  
}
```

variable i has no access specifier because, it is a local variable. Because of this we can not even mention a local variable as static.

NOTE: Since, there is no access specifier for variable i do not think that its accessibility is default.

Remember that access specifiers are not applicable to local variables.

Question : what all a class can contain?

Ans : A class can contain the following 7 elements.

- (i) variables
- (ii) constants
- (iii) constructors
- (iv) functions
- (v) Inner classes
- (vi) static blocks
- (vii) non-static blocks.

Till now, we have learnt about the first four elements.

Inner class: A class defined in another class is known as an Inner class.

- Inner classes are categorized into four types:

- (i) Member Inner classes
- (ii) Static Inner classes
- (iii) Local Inner classes
- (iv) Anonymous Inner classes

(i) Member Inner class: If we define a class in another class just as a member (like variables & functions) of that class, then we call such a class as Member inner class.

Ex:

```
class Outer  
{  
    int x;  
  
    class Inner  
    {  
        int i;  
        public void funIn()  
        {  
            System.out.println("Inside funIn() of Inner");  
            x = 25;  
            funOut();  
        }  
        public void funOut()  
        {  
            x = x + 1;  
            Inner in = new Inner();  
            in.i = 25; System.out.println(in.i);  
        }  
    }  
}
```

```

public static void main (String args[])
{
    Outer.Inner oi = new Outer().new Inner();
    oi.funIn();
}
}

```

④ → This is the syntax for creating objects of inner classes from static functions of outer classes.

10.08.2006

- for a member inner class, any non-static member of the outer class is directly available.
- Since a class can contain static and non-static members, we may think of defining static members in the inner class. But, rule says that non-static member inner classes should not have static members.
- In the above program, if the main method is present in some other class Test within the same path, even then we follow the same (④) syntax for creating an object of inner class.

Ex class Test

```

{ public static void main (String args[])
{
    Outer.Inner oi = new Outer().new Inner();
    oi.funIn();
    System.out.println(oi.i);
}
}

```

- thus we can create objects of Inner class from its outer class as well as from some other class which is outside the outer class.
- If we declare an inner class as private, then it is possible to create object of inner class only in its outer class. Outside the outer class, it is not possible to create object of inner class.
- An inner class can extend any class which is available to its outer class.
- An inner class can extend an outer class but there is no use of it because, since inner class is a member of the outer class, even without using the 'extends' keyword, the properties of outer class are available to the inner class.

(ii) Static Inner class: If we define any inner class as static, then it is known as static inner class.

~~(*)~~ The properties of static inner classes are as follows:

- (a) we can define any access specifier before a static inner class
- (b) static inner classes can have static members as well as non-static members.
- (c) static members of a class are directly available to the static/non-static members of that class but non-static members of a class are not directly available to the static members of that class.

(d) In a static inner class, even though if we define a non-static function, it is still static because, it is in a static class. i.e. it is in static domain.

∴ From static inner class, we can not access any non-static member of the outer class directly. In order to do so, we need to create object of outer class explicitly.

Ex:

```
class Outer
{
    int x;
    static int y;
    static class Inner
    {
        public static void func1()
        {
            System.out.println("Inside func1() of Inner");
            y = y + 1;
        }
        public void func2()
        {
            System.out.println("Inside func2() of Inner");
            x = x + 1; // error: (refer (d) above)
        }
    }
    public void func()
    {
        System.out.println("Inside func() of Outer");
        Inner s = new Inner(); // (refer (e) above)
        s.func2();
        Inner.func1(); // func1() is static
    }
}
```

```
    run main (String args[])
```

```
{    y = y+1; // (refer property (2))
```

```
    Sinnerl s = new Sinnerl();
```

```
    s.fun2();
```

```
    Sinnerl.fun1();
```

```
}
```

Q: How can you access members of the static inner class if the main method is somewhere outside the outer class?

Ex:

```
class Test
```

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

```
{
```

```
    Outer1.y = 10;
```

⊕ Outer1.Sinnerl o1 = new Outer1.Sinnerl();

⊖ o1.fun2();

⊖ Outer1.Sinnerl.fun1();

```
}
```

```
}
```

⊕ → Syntax for creating object of static inner class from outside the outer class.

⊖ → This is the way in which we refer static and non-static members of static inner class from a class outside the outer class.

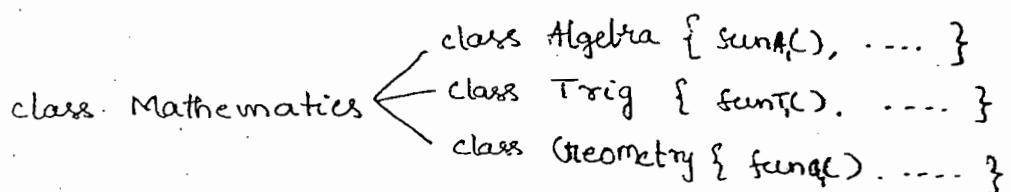
Question: what are the uses of inner classes?

Ans: we can achieve more modularity using inner classes.

whenever we need to define large no. of functions in a class, then we go for inner classes.

Ex:

If we have a class Mathematics, then instead of defining all the functions in that single class, then we can group functions of topics individually.



(ii) Local Inner classes: when we define a class, local to a function, just as a local variable, then it is known as local inner class.

Ex:

```
class Outer
{
    int x;
    public void fun1()
    {
        int i;
        final int j=30;
        class Inner
        {
            public void funL()
            {
                System.out.println("Inside funL() of Inner");
            }
        }
    }
}
```

```

        s.o.p(a);
        // s.o.p(i); // error (refer (a) below)
        s.o.p(i); // (refer (d) below)
    } // end of funL()
} // end of class Linneal
Linneal l = new Linneal(); // (refer (e) below)
l.funL();
s.o.p("end of funL()");
} // end of funL()
public static void main (String args[])
{
    Louter l = new Louter();
    l.funL();
}
}

```

Properties of local inner classes:

- Local variables of a function are not accessible from the local inner class even though local inner class is defined inside the function.
- Since local inner class is local to a function, it is similar to a local variable inside the function.
So, all the rules of local variables apply to the local inner classes.
- Thus, local inner classes can not have any access specifier.
- Local constants of a function are accessible from its local inner class.

(ii) we can not create object of local inner class inside the main of the outer class. If at all, we can create its object only inside the function in which it is defined.

iv, Anonymous Inner classes: Anonymous inner classes are the inner classes defined without a name.

- In order to define an anonymous inner class we definitely need an interface (or) an abstract class.

11-08-2006

- Anonymous inner classes are two types

local anonymous inner classes
Member anonymous inner classes.

Example program:

interface Xyz

```
{ public void funX();  
    public void funY();  
}
```

object of anonymous inner class is assigned to interface reference xl.

class Anonyl

```
{ static Xyz xl = new Xyz() {
```

public void funX() {}

public void funY() {}

};

}

class Test

```
{ p. s. v. main (String args[])  
{    Anonyl . xl . funX();  
}
```

→ This class is an anonymous inner class. It has no name.

NOTE: We are not even using the keyword 'class' to define this anonymous inner class.

we are talking
about interfaces.

The example which we have referred to, in that context
is JDBC connection establishment.

The following program is very much similar to this
concept.

interface Xyz

```
{ public void funX();  
    public void funY();  
}
```

class Anony2

```
{ public static Xyz getXyz()  
{  
    Xyz x1 = null;  
    x1 = new Xyz() {  
        public void funX() {}  
        public void funY() {}  
    };  
    return (x1);  
}
```

class Test1

```
{ public static void main (String args)  
{  
    Xyz x1 = Anony2.getXyz();  
    x1.funX();  
    x1.funY();  
}
```

The following program is a still more reasonable example.

interface Ijk

```
{ public void funI (int i); }
```

class Anony3

```
{ public static Ijk getIjk (int x) {
```

```
    Ijk ii = null;
```

```
    if (x > 10)
```

```
    { ii = new Ijkl() { public void funI (int i) {  
        int sum = 0;  
        for (int k=0; k<=i; k++)  
            sum = sum + k;  
        S.O.P ("sum is " + sum);  
    } };
```

```
    }
```

```
    else
```

```
    { ii = new Ijkl() { public void funI (int i) {  
        int sq = 0;  
        sq = i * i;  
        S.O.P ("square is " + sq);  
    } };
```

```
    }
```

```
    return (ii);
```

```
}
```

```
} // Anony3
```

```
class Test2
{
    public static void main (String args[])
    {
        Ijk i1 = Anony3 . getIjk (25);
        Ijk i2 = Anony3 . getIjk (6);
        i1 . funI (5); // sum is 15
        i2 . funI (5); // sq is 25
    }
}
```

in the above class, the ^{first}/_{second} statement in main and the statement in the connection example can be compared.

```
Ijk i1 = Anony3 . getIjk (25);
Connection con = DriverManager . getConnection (url, user, pwd);
```

static and non-static blocks

class. Bdemo1

Bdemon1.java

```
{  
    int i;  
  
    Bdemon1()  
    {  
        S.O.P("Inside the constructor");  
    }  
  
    static  
    {  
        S.O.P("Inside the static block");  
    }  
  
    {  
        S.O.P("Inside the non-static block");  
    }  
  
    public static void main (String args [ ] )  
    {  
        S.O.P("Inside main");  
        Bdemon1 b1 = new Bdemon1();  
    }  
}
```

Output

Inside static block

Inside main

Inside non-static block

Inside the constructor.

(P.T.O)

Explanation for the output of the above program:

- As soon as JVM starts executing the .class file, it first creates context of the class.
- Then all the static members of the class are loaded into the context.
- Now JVM searches for the main method. Once it finds the main method, it starts executing all the static blocks other than main method. After that it starts executing main method.
- Inside the main method, we are creating object of the class. As soon as JVM encounters the new operator, it reserves memory space in the RAM, i.e. creates the object, loads all the non-static members of the class in to the object, initializes the uninitialized variables and then it loads the constructor.
- But, after the creation of the object and before the loading of the constructor, it searches for non-static blocks. At this point of time, it executes all the non-static blocks (if any) and then loads and executes the constructor.
- This is the control flow in any java program.

NOTE: Non-static blocks would never be executed, if the object of that class is not created.

use of static blocks:

in real time scenarios, static blocks are used to provide information regarding the project (i.e. version, copy rights, name of the company that developed the project e.t.c.) before actually the project is executed. (i.e. main() is called).

use of non-static blocks:

if we have many constructors in a class and if every constructor has some common statements, then instead of repeating those statements in each constructor, we place those statements in a non-static block.

in this way we can avoid the duplication of the code.

12-08-2006

Exception Handling

Exceptions: exceptions are objects created by JVM that represent the corresponding logical errors.

Syntax errors: Syntax errors occur whenever a statement written by the programmer can not be compiled by the compiler.

Whenever a source code is compiled, first compiler checks whether a proper environment is available for the JVM or not to execute the byte code which it has generated for the source code.

- Thus, Syntax errors and improper environment result in compilation errors.

Logical errors: Sometimes, JVM would be unable to execute statement(s) in the bytecode for some reasons. This results in logical errors.

NOTE: whenever a logical error occurs, program is terminated and control comes out of the program unconditionally without mentioning from which part of the program it has come out.

Ex: class A

```
{ public static void main (String args[])
{
    int i=100;
    int j=0;
    int a=i/j;
    s.o.p (a);
}
```

The above program will be perfectly compiled. But, while executing it, value of α becomes "infinity" and since α is of type integer (32-bit) JVM would not understand how to store infinity value in 32-bit location.

- Thus, it terminates the program.
- There are some predefined classes in `java.lang` package whose objects are created by JVM whenever it encounters a logical error. (Names of such classes are found under the Exception classes in API).
- For each and every logical error, there is a corresponding exception class in the `java.lang` package.
- whenever JVM encounters a logical error, it creates an object of the corresponding exception class and since this object is not explicitly handled, this object reaches back to the JVM. JVM (a set of programs) accepts these objects and terminates the program.
- Now, if we are able to handle those exception class objects, we can prevent the program from getting terminated.
- Thus, concept of handling the exception class objects and avoiding them from reaching back to the JVM is known as Exception Handling.
- we can handle the exception class objects by assigning them to the corresponding exception class references.

Procedure of handling the exceptions:

- we have two key words 'try' & 'catch' to implement exception handling.
- The 'try' keyword recognizes the exception class objects created by the JVM and it gets hold of those objects and transfers them to the "catch" block.
- In the 'catch' block we should be able to assign those objects to the appropriate references so that they will not reach JVM and therefore program would not be terminated.

Ex :

```
function()
{
    try {
        =====
    }
    catch (Exception e1)
    {
        s.o.p ("Exception in fun1()");
        s.o.p (e1);
    }
}
```

- Thus, as soon as an exception is raised, the user would easily know in which part of the program the logical error has occurred.
- `s.o.p(e)` prints some message instead of printing the address contained in `e`. This is because the exception class overrides the `to-string` method of the object class.
- In this way, we can use try and catch blocks in any part of the program.

Uses of Exception handling:

- (i) we can know in which part of the program the logical error has actually occurred.
- (ii) we are not allowing the JVM to terminate the entire program. only the 'try' block in which the error has occurred would be terminated.
- (iii) Thus, this is an easy way of debugging the program.

NOTE: (i) once control enters the catch block, it executes the statements in the catch block and moves on further. It will not come back to the 'try' block because of which the 'catch' block was executed.

(ii) if no exception class object is created in the 'try' block, control will never enter the catch block.

- every 'try' block should be associated with atleast one 'catch' block.
- A 'try' block can have any no. of 'catch' blocks.

Ex:

```
class Elemenol
{
    public static void main (String args[])
    {
        int i = Integer.parseInt (args[0]);
        int j = Integer.parseInt (args[1]);
        int k = Integer.parseInt (args[2]);
        int l = Integer.parseInt (args[3]);

        A a1 = null;

        try
        {
            System.out.println ("Inside try block");
            int x = i/j;
            System.out.println (x);
            int arr[] = new int [k];
            arr [l] = x;

            if (x > 10)
                a1 = new A();
            a1.funA();
            System.out.println ("end of try block");
        }

        catch (ArithmaticException e1)
        {
            System.out.println (e1);
        }
    }
}
```

class A

```
{
    funA()
    {
        System.out.println ("funA()");
    }
}
```

```
catch (ArrayIndexOutOfBoundsException e2)
```

```
{  
    s.o.p(e2);  
}
```

```
catch (NullPointerException e3)
```

```
{  
    s.o.p(e3);  
  
    s.o.p();  
    s.o.p(i);  
    s.o.p(k);  
    s.o.p(l);  
}
```

- For each and every statement in the try block which is proven to generate an exception class object, there should be a corresponding catch block.

Disadvantages of this approach:

- If statements in the try block are more that generate the exception class objects, then we should have those many catch blocks which is not feasible.
- The user should have thorough knowledge regarding, what statement is proven to generate what type of exception class object. At times, this would be very difficult.

(P.T.O)

- To overcome the above disadvantages, we have to define catch block in such a way that object of any exception class can be assigned to the reference defined in the catch block.
- Hence we define Exception class reference in the catch block. Exception class is the super class of all the exception classes.

Question: can we define a try block without a catch block?

Ans: Yes, but that procedure is not recommended.

- we have altogether five keywords in the concept of exceptions. They are:
 - (i) Try (ii) catch (iii) throw (iv) throws (v) finally.
- Logical errors are of two types:
 - (i) Simple logical error (ii) Serious logical error.

14-08-2006

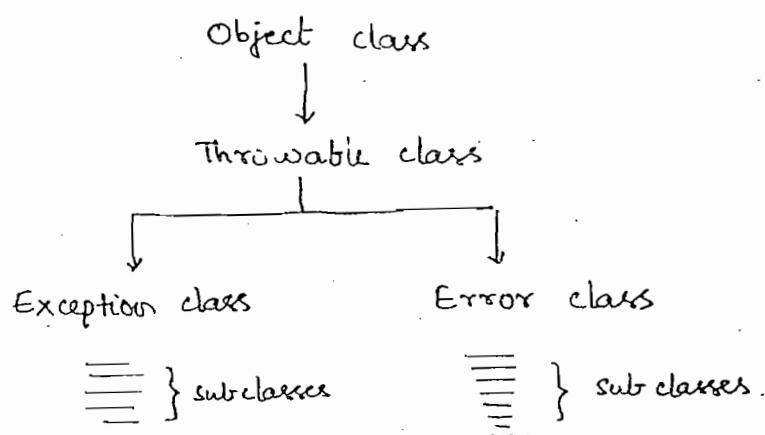
- Simple logical errors: There are some logical errors which can be neglected by the JVM. Such errors are known as simple logical errors.

Ex: int $\alpha = 100/0$; c.t.c.

NO

- Serious logical errors: Some logical errors can not be neglected by the JVM. Such errors are known as serious logical errors.
- Ex: creation of an object of a class, when there is no availability of memory space in the RAM.

- All the classes coming under 'exceptions' are defined to represent 'simple logical errors'.
- All the classes coming under 'error' are defined to represent 'serious logical errors'.
- The class hierarchy of errors and exception classes is as follows:



- Exception class is the super class of all exception classes and Error class is the super class of all error classes.
- Throwable class is the super class of Exception class and Error class.
- Object class is the super class of Throwable class.

NOTE: 'try' block and 'catch' block are designed in such a way that, try block can transfer only objects of Throwable class and its sub classes. Similarly, catch block can receive only objects of Throwable class and its sub classes.

- try block can not transfer objects of any other classes and similarly catch block can not receive objects of any other classes.

- This is why we can't write `catch (Object e)
{ };`
- we can use `catch (Throwable t) { };`, but there is a problem.
 - on real time, we are not permitted / supposed to handle Error class objects. They will be handled internally by the JVM.
 - In case, if we handle them, then it means that we are not allowing objects of Error classes to reach JVM. In other words, we are directly neglecting the serious logical errors, which is highly dangerous.
- We know that Throwable class is a super class of Error class and by using `catch (Throwable t) { };`, we are indirectly handling Error class objects, which is problematic.
- we never use `catch (Throwable t) { };`
 - we can always use `catch (Exception e) { };`
- Thus, we can /are permitted to handle only objects created because of simple logical errors and we are not permitted to handle serious objects created because of serious logical errors.
- Another important concept in exception handling is checked Exceptions and unchecked Exceptions.

- Sometimes, in a program, applying concept of exception handling to some statements is not compulsory at the compilation time. For each and every statement which is proven to generate an exception, compiler will check whether we are handling the corresponding exception class object or not.
- In the process of checking, compiler will not bother much even though if we don't handle objects of some exception classes.
- But there are some statements for which applying concept of exception handling is mandatory at the compilation time itself. If we do not handle them, compiler compells us to handle them by generating a compilation error.
- The exception objects for which compiler compells to handle them are known as checked Exceptions.
- The exception objects for which compiler does not compel to handle them are known as unchecked Exceptions.
- ClassNotFoundException, IOException etc. are examples of checked exceptions.
- ArithmeticException, ArrayOutOfBoundsException etc. are examples of unchecked exceptions.

Question: How can you find out which exception comes under which category (i.e. checked or unchecked)?

(P.T.O)

- In the category of exceptions, there is one class by name RuntimeException. It is a sub-class of Exception class.
- Any class which is a sub-class of RuntimeException class is an unchecked exception
- Any class which is not a sub-class of RuntimeException class is a checked exception.

16-08-2006

Consider the following program:

```

class Edemo3
{
    public void fun1(int i, int j)
    {
        int x = i/j;
        System.out.println(x);
        A a1 = null;
        if (x > 10)
            A a1 = new A();
        a1.funA();
    }
}

public static void main (String args[])
{
    int x = Integer.parseInt (args[0]);
    int y = Integer.parseInt (args[1]);
    Edemo3 e1 = new Edemo3 ();
    e1.fun1 (x, y);
    System.out.println (x);
    System.out.println (y);
}

```

on the above programs, if $y=0$, then we definitely get an exception in `func()`. Now, we need not do anything explicitly for the exception object to reach the calling place. The control itself, transfers the object to the calling place (in this case, it is `main()` method) and then it is sent to the JVM and the program is terminated.

- Let us assume that the main method is not present in the above program & some designer has developed `Edemo3.java` file and he has supplied us only the `Edemo3.class` file. We are using this program in our application.

Now, how can we know, that what type of exception objects will be created because of the statements present in `func()`.

To solve this problem, the developer himself will provide a facility i.e. along with the function definition, he will mention the type of exception that might be raised inside the function. All this, is provided in the documentation.

That 'mentioning' is done using the 'throws' keyword.

throws : It is a keyword using which the end user is informed that a particular function is proven to transfer exception class objects to its calling place.

NOTE : Most of us will be in the wrong perception that using ''throws'' keyword we can handle exceptions. But it is not true.

- whether we use 'throws' keyword or not, the exception class object generated in a function will definitely reach the calling place (if the exception class objects are not handled inside the function).

Ex :

```
class Edemo3
{
    public void fun1(int i, int j) throws ArithmeticException,
                                         NullPointerException
    {
        int x = i/j;
        s.o.p(x);
        A a1 = null;
        if (x > 10)
            a1 = new A();
        a1.funA();
    }
}
```

- In the above example, since fun1() is throwing unchecked exceptions, it is not mandatory to call fun1() inside 'try' block from its calling place.
- But if we use 'throws Exception', then it is mandatory to call fun1() inside 'try' block from its calling place because 'throws Exception' indicates that fun1() can throw checked Exceptions as well as unchecked exceptions.

- consider the following function, inside class A

```
class A
{
    public void funA() throws ClassNotFoundException
    {
        ==
    }
}
```

Let us assume that, we have a statement(s) inside funA() which is proven to generate a checked exception. Now, if we do not want to handle it inside the function, then we need to mention 'throws <exception>' along with the signature. But the condition is that, at the calling place, funA() has to be called within the 'try' block.

Uses of 'throws' keyword:

- (i) By seeing the 'throws' keyword along with the function definition, we can understand that, the function has statements which are proven to generate the mentioned exception class objects and they are not handled in the function itself.
- (ii) If we want to skip the 'try' block inside the function, even though if it is generating checked exception(s), then simply place 'throws' keyword along with the type of exception generated. But condition is, the function call should be made inside a 'try' block at the calling place.

Another example on the usage of 'throws' keyword

```
class Edemo3
{
    public void fun1(int i, int j) throws Exception
    {
        int x = i/j;
        s.o.p(x);
        A a1 = null;
        if (x > 10)
            a1 = new A();
        a1.funA();
    }
}
```

```
class Test
{
    int x, y;

    void funT() throws Exception
    {
        Edemo3 e1 = new Edemo3();
        e1.fun1(x, y);
    }
}
```

```
/* public static void main(String args[]) throws Exception.
{
    Test t = new Test();
    t.x = 100; t.y = 0;
    t.funT();
    s.o.p("end of main()");
}
*/
```

- In the above program, fun1() of class Edemo3 has some statements which are proven to generate Exception class objects. Since, we are not interested in handling the exceptions we are using 'throws Exception' in the function definition.
- Since we are using 'throws' with fun1(), the function call should be inside 'try' block at the calling place. calling place of fun1() is funT() of class Test. Here also we are not interested in handling the exceptions. Hence we are using 'throws Exception' for funT() also.
- obviously, at the calling place, the call to funT() should be inside 'try' block. The calling place is main() method. Here also, if we are not interested in handling the exceptions, then we can use 'throws'.
- Now, what happens is, the exception object which will be generated inside main because of the above function calls, will reach the calling place of main method.
- what is the calling place of main method?
Ans: It is JVM.
- ∵ The exception object reaches JVM. we know what happens when an exception object reaches JVM. control comes out of the program and JVM terminates the program in the middle.

∴ Even though 'throws' is applicable to main(), it is highly recommended not to use 'throws' with main().

NOTE: 'throws' keyword is applicable to any function.

- The main commented main() method, therefore, should be replaced with the following modified main() method.

```
public static void main(String args[])
{
    Test t = new Test();
    t.x = 100; t.y = 0;

    try
    {
        t.fun();
    }
    catch (Exception e)
    {
        S.O.P(e);
    }

    S.O.P("end of main()");
}
```

NOTE (i): whenever we use catch (Exception e) with a 'try' block, then, we are assigning unchecked exception class objects to the checked exception class reference.

(ii) whenever we use 'throws' compiler understands it as that, the function generates checked exceptions even though there are no statements inside the function that create checked exceptions.

- 'throws' keyword and constructors:

Definitely we can use 'throws' keyword with constructors because constructors may have many no-q. statements which are proven to generate exception class objects.

```
class Edemo4
{
    int i, j, k;
    Edemo4 (int x, int y) throws Exception
    {
        i = x; j = y;
        k = i/j;
    }
}

class Test4
{
    public static void main (String args[])
    {
        try
        {
            Edemo4 e1 = new Edemo4 (10, 0);
        }
        catch (Exception e1)
        {
            s.o.p (e1);
        }
        s.o.p (e1.i); // compilation error.
    }
}
```

Error occurs because, e1 is defined inside try block and hence it is local to 'try'.
s.o.p (e1.i) is outside try and is again proven to generate checked exceptions.

To avoid this problem, define `e1` outside the `try` block and inside the `main()` so that it will be available to all other parts of `main()`, as shown below.

```
class Test4
{
    public static void main (String args[])
    {
        Edemo4 e1 = null;
        try
        {
            e1 = new Edemo4 (10, 0);
        }
        catch (Exception e)
        {
            s.o.p (e);
        }
        s.o.p (e1.i); // Now we do not have any compilation error.
    }
}
```

Question: How can we handle exceptions of super class constructors in their subclasses?

Ans: We know that, when a class extends another class, and when we try to create object of subclass, first the super class object is created and then the subclass object is created.

Now, what should we do if super class constructor has 'throws Exception' and subclass has no user defined constructor init?

The following program explains the answer for the above question.

```
class A
{
    int i, j;
    A() throws Exception
    {
        s.o.p("Inside A()");
    }
}

class B extends A
{
    int x, y;
    B() throws Exception
    {
        //super();
    }
}

public static void main (String args[])
{
    try
    {
        B b1 = new B();
    }
    catch (Exception e)
    {
        s.o.p(e);
    }
}
```

↑
Assume it is commented
temporarily.
↓

Explanation:

- Assume that constructor of class B is not present (temporarily). we know that when no constructor is present, compiler provides a default constructor.
- Also remember that, first statement of every constructor will be a 'call' to the super class constructor.
- we are creating object of class B inside the 'try' block . (The reason will be known at the end of this explanation).
- when we try to create object of class B, first of all constructor of class B is executed. Since we have assumed that there is no constructor in class B, we will have the default constructor.
- Since class B extends class A, from the default constructor of class B , constructor of class A is called. Note that constructor of class A has 'throws Exception'.
- Therefore, the exceptions generated in constructor of class A should be handled at its calling place. Here, it is default constructor of class B.
- NOTE: A default constructor will not have any 'throws' keyword
- Since default constructor of class B has no 'throws Exception' and it is not handling the exception, compiler gives us a compilation error.

- For avoiding this situation, we have to define an empty constructor in class B - as shown in the program, which transfers the exceptions to its calling place.
- calling place of constructor of class B is nothing but main() method of class B in which we are creating object of class B.
- Since constructor of class B has 'throws Exception' it's calling place should handle the exceptions because calling place is main() for which we ~~can~~ are not supposed to use 'throws Exception'.
- This is the reason, why, "B b1 = new B();" should be placed inside 'try' block in this program.
- This is how, we handle exceptions of super class constructors in their subclasses.

Point to remember regarding 'try-catch'

whenever a try block has n no. of catch blocks, then, the o if an exception object is created, then the order of preference of catch blocks is, the order in which they are present below the 'try' block

(P.T.O)

consider the following scenario.

```
try
{
    ...
}
catch (Exception e)
{
    s.o.p (e);
}
catch (ArithmaticException e1)           → unreachable blocks.
{
    s.o.p (e1);
}
catch (NullPointerException e2)
{
    s.o.p (e2);
}
```

Now, if at all an exception is raised in try block,
it will be transferred to the immediate catch block.

If that catch block is not capable of handling that
exception, then it is transferred to the next catch block
and so on.

- In this example, the very first catch block is
`catch (Exception e)`. Since Exception class is the super
class of all exception classes, its reference will be
capable of receiving any exception class objects.
- Thus, in the above program, the remaining catch blocks
are not going to get executed at any instance of time.

such blocks which are not going to get executed at any instance of time are known as unreachable blocks.

NOTE: Unreachable blocks are not supported by Java.

Hence, whenever we compile a source code, compiler searches for unreachable blocks in the source code. If it finds any, it gives a compilation error.

Hint: whenever we need to define multiple catch blocks and if catch(Exception e) is one among those, then we should see that catch(Exception e) will be the last catch block in those group of multiple catch blocks.

18.08.2006

'finally' keyword:

The finally keyword ensures that all the statements present inside the block finally block are compulsorily executed.

Question: what is the need of finally block?

consider the following situation.

Sometimes, some statements inside a try block need to be executed compulsorily whether exceptions occur or not.

Ex:

```
try {
    get connection to a printer
    ===
    con.close()
}
catch (Exception e)
{
    s.o.f(e);
}
```

Now, after getting connection to a printer, let us assume that an exception is generated in one of the statements after getting the connection. Then control comes out of the try without executing the rest of the statements.

The last statement inside try block is `con.close()`. This statement will not be executed if an exception occurs before this.

Now, the scenario is, the resource (printer) has been allocated to our system, but it is not released. This is ineffective resource management.

- To overcome this problem, many people suggest one solution i.e. to write `con.close()` outside try catch. Not only `con.close()`, but any statement which needs to be compulsorily executed is supposed to be placed outside try catch.
- In fact, this solution works well, but it is not at all recommended. Why?

try: By doing so, we break the coding standards

- Integrity, Modularity etc.

- Coding standards say that, all the statements related to one specific block should be in that block only (i.e. Integrity)
- By writing the statements (which are to be compulsorily executed) outside the try block, even though they belong to try block, they get mixed up with other statements present in the

function. Thus, there is no proper separation and hence we break modularity.

- In order to maintain modularity, all the statements present in try block that are to be compulsorily executed irrespective of exceptions, are to be placed in a separate block - finally.
- 'finally' is a block associated with try block just as the catch block.
- After the execution of either try or catch blocks, it then executes the finally block and then goes for the rest of the statements present in the function, outside the try - catch.
- finally block can not exist individually without a try block.

Ex

```
public void fun()
{
    try { }
    catch (Exception e)
    {
        // ...
    }
    finally { }
}
```

NOTE: A try block can have any no. of catch blocks but it can have only one finally block.

Question: ~~Section~~ can you define a try block without a catch block?

Ans: Yes, by defining a finally block with the try block.

```
try { }
```

```
finally { }
```

NOTE: The above code is not at all recommended. Why?

We know that, whether exceptions occur or not, finally block is going to get executed. Thus, if no exception occurs it is well and good. Suppose, if an exception occurs, even then, the finally block is going to get executed and hence we do not know whether an exception has occurred or not. Even, if an exception occurs, we do not know what type of exception occurred and where it has occurred.

- This is the disadvantage by defining a try-finally block without a catch block. So, this is not recommended.

Userdefined Exceptions:

- All these times we have been using predefined exception classes whose objects were created by JVM, because of predefined logical errors.
- In a similar manner, we can define our own exception classes whose objects will be created when user defined logical errors occur.

- Strictly speaking, the user defined logical error is not a logical error from the JVM's point of view.
- whenever an erroneous situation is expected to occur in a function, then we can terminate the function in the middle by using user defined exceptions. Thus, we can prevent the function from returning incorrect values.
- In this situations, we use user defined exceptions.

The following program illustrates the use of user defined Exceptions:

```

class InvalidAgeException extends Exception
{
    public String toString()
    {
        String s1 = "InvalidAgeException : Invalid age value passed";
        return(s1);
    }
}

class Emp extends Exception
{
    public float genPension (int age, float sal) throws Exception
    {
        float pen = 0;
        if (age > 45 & age < 100)
        {
            pen = (age * sal)/100;
        }
        else
        {
            throw new InvalidAgeException();
        }
        return(pen);
    }
}

```

↑ This is some logic to calculate pension. We have considered so. for simplicity.

```

class Test
{
    public static void main (String args[])
    {
        int age = Integer.parseInt (args[0]);
        float sal = Float.parseFloat (args[1]);
        Emp e = new Emp();

        try {
            float p = e.getPension (age, sal);
            System.out.println (p);
        }
        catch (Exception e1)
        {
            System.out.println (e1);
        }
        System.out.println (age);
        System.out.println (sal);
    }
}

```

Explanation:

- first define a user defined exception class - InvalidAgeException
- Since InvalidAgeException class object is generated because of simple logical error, it should be a sub-class of Exception class. Thus, we have to extend Exception class to our class.
- class Emp has one function getPension() which takes two arguments and calculates the pension and then returns the pension value.

- In class Test, we call the `getPension()` method to calculate the pension of an employee.
- whenever we pass the arguments - age & sal, to the `getPension()` method, it returns the pension.
- Now, our aim is to throw an exception object from the function, when the user passes an invalid age. (say for example, 780 (or) -25 e.t.c.).
Since age is of type integer, JVM will not encounter any logical error (because java supports signed integers). But from the user's point of view, this is a logical error.
So, in the function, we specify a condition for age. If the condition is not satisfied, it should throw an exception so that function will not calculate pension with that invalid age and return incorrect value.
- Since, this is a user defined exception, we have to throw that exception explicitly by creating object of that exception class. Hence the statement,
`throw java new InvalidAgeException();`
- Since, we are not handling the exception inside the function, it should be indicated. Thus, we write throws Exception in the function definition.

contd-- (P.T.O)

- Sure, we mentioned 'throws Exception' in the function definition, we should handle the exception class object at the calling place of the function.
That is why, we make the 'function call' inside the try block in the main() method.
- This is how, we make use of user defined exceptions.

Question: what is the difference between 'throw' & 'throws'

Ans: 'throws' is used to mention (or) indicate to the compiler as well as the end user, that a particular function is proven to generate exception class objects and those objects are not handled in the function itself.

'throw' is used to explicitly transfer the userdefined exception class object from the function to the calling place.

MULTITHREADING

- Before we go into multithreading we have to understand what is the difference between multithreading.

Consider the following program:

```
class MSD
{
    fun1()
    fun2()
    fun3()
}

public static void main (String args[])
{
    MSD s1 = new MSD();
    s1.fun1();
    s1.fun2();
    s1.fun3();
}
```

Normally, if the class is compiled and JVM executes the program, then the order of execution is that, first fun1() is called and after complete execution of fun1(), then control comes back to main() and then fun2() is called. After complete execution of fun2(), then fun3() is called and executed. This is how the flow goes on.

Now, let us assume that fun1() has some statements which involve data transfer.

We know that data transfer is not the job of the processor.

contd... (P.T.O)

It is the duty of a separate individual circuit (DMA) which functions under the control of the processor.

Since func() has data transfer statements, processor assigns that job to DMA. During this time the processor should remain Idle.

- The state of CPU where CPU waits for DMA circuit to transfer the data is known as Idle state.
- This made software developers to think that, efficiency of the processor would be increased if processor is made to do some other useful work during this idle state. The useful work is nothing but, executing other functions present in the program.

This need led to the concept of multitasking.

- Remember that processor is also an electronic circuit and at any instant of time, it can execute only one statement. It can not execute multiple statements simultaneously.

Multitasking: The concept of executing multiple functionalities simultaneously is known as multitasking.

If we apply the concept of multitasking to our program, then control flow would be as follows:

A part of func() will be executed and then control switches to function fun2(). Now, here also, a part of fun2() is executed and then control switches to fun3(). A part of fun3() is executed and then control again comes back.

to func(). Now it continues executing func() from where it had stopped earlier. Similarly, for func2() and func3(). (This is not guaranteed. The sequence may change)

- Now, when we see the output, we feel that the processor is executing the three functionalities simultaneously.

** Thus, concept of multitasking came into existence to avoid the idle state of CPU.

- In multitasking we said that, a part of a functionality is executed one at a time. Now, the question is 'part' means how many statements.
- This conflict is resolved by Scheduling.

Scheduling: It is the process in which a specific time period is allocated to a functionality where the control remains in that particular functionality for that specific time period.

Once the time period is lapsed, control switches to another functionality with another time slice and so on..

- Scheduling is supervised by the Scheduler.
- The time slices allocated will be in the order of nano seconds.
- Thus, by the time our eye recognizes the execution of one part, control switches to another part of the program.

(P.T.O)

- Now, it is clear that, without applying multitasking, one function will depend upon another function i.e. after execution of fun1(), fun2() will be executed and so on.

Advantages of multitasking:

- (i) Multitasking was invented to avoid idle states of CPU
- (ii) But interestingly, it is used to make the functions get executed independently.

NOTE: Actually, in realtime we use multitasking to make the most of the second advantage rather than the first, because in case of small projects, idle states of CPU is not a big concern.

- Animation along with form submission is a very good example of multitasking.
- Multitasking is of two types:
 - (i) Thread based multitasking
 - (ii) Process based multitasking
- Java supports only thread based multitasking.

21-08-2006

Process based Multitasking:

Concept of executing more than one program simultaneously which are present in different locations of RAM is known as process based multitasking.

- Thus, in process based multitasking, addresses of both the programs have to be maintained since control has to shift from one part of the RAM to another.
- This increases the overhead on the processor. This, it is the disadvantage of process based multitasking.

Thread based Multitasking:

Concept of executing more than one functionality simultaneously belonging to the same memory domain (i.e. same program) is known as Thread based multitasking.

Thread: Thread is a functionality (group of statements) which would be getting executed simultaneously with the other part of the program.

Question: what is the difference between a process and a thread?

-Ans: Process is any program under execution.

Thread is a part of the process.

In other words, thread is the smallest part of a process.

(P.T.O)

Implementing a thread in java:

Implementing thread in java is a two step process.

- (i) First of all we have to define a function which can be executed as a thread.
- (ii) Now, this function has to be executed as a thread.
 - The signature of a functionality using which we implement a thread is defined in an interface by name 'Runnable'. This interface has one function definition - public void run() for which we need to provide body.
 - After providing body, we need to execute this functionality as a thread (i.e. simultaneously with other part of the program). This is done as follows.
 - There is a class by name Thread present in the API. This class supports many functionalities which ~~execute~~^{make} a given functionality to get executed as a thread. One such functionality is the start() method.
 - The start() method recognizes the run() method of Runnable interface and then the run() method is executed as a thread.

- Thread class and Runnable interface are the two structures using which we implement thread based multitasking in java.

| | |
|--|---|
| <ul style="list-style-type: none"> - interface Runnable <pre>{ public void run(); }</pre> | <pre>class Thread implements Runnable { public void run() { } }</pre> |
|--|---|

Now we can implement threads using either of the following two ways.

class A implements Runnable

```
{
    public void run()
    {
        =====
    }
}
```

(OR)

class A extends Thread

```
{
    public void run()
    {
        =====
    }
}
```

contd..(P.T.O)

NOTE: whether we extend Thread class or implement Runnable interface directly, we are using run() method of the Runnable interface i.e. using Runnable interface directly or indirectly is compulsory.

**

Question: How do you define threads in java?

Ans: By using Runnable interface.

- An example program on thread based multitasking.

```
class MSD extends Thread {  
    public void run() {  
        for (int i = 0; i < 40; i++)  
            System.out.println(" Inside run method : " + i);  
        System.out.println(" end of run() method ");  
    }  
  
    public static void main (String args[]) {  
        MSD a = new MSD();  
        a.start();  
        for (int i = 37; i < 80; i++)  
            System.out.println(" main : " + i);  
        System.out.println(" end of main ");  
    }  
}
```

Explanation of the above program:

Functionality of the start() method is to pick the run() method stored in the object or class. The start() method is called and it causes the run() method to run in a separate thread. It also takes some arguments, parameters and other arguments if needed, in addition to the run() method.

- If we call run() method directly on the object of the class (i.e. a.run()), run() method will be executed just as a method instead of getting executed as a thread because, run() is itself an ordinary method.
- It is going to get executed as a thread, when start() method hands it over to the local OS.

This is done by calling start() on the object.

Behaviour of a Thread:

In the above program, run() is the thread and main() is the other part of the program.

We can't say which one starts first or ends first.

Since both the methods are getting executed simultaneously, the start/end of execution of a method completely depends upon the time slice allocated for each functionality.

(P.T.O)

NOTE: Thus, because of scheduling, we can't guess the output of thread based programs.

22-08-06

Defining multiple threads

we define a thread using run() method of the Runnable interface. Remember, a class can have only one P run() method. Therefore, to define multiple threads, we need to define those many classes with run() methods for those many threads we want to define and implement.

- Thus, by extending Thread class to our classes, we can define and implement multiple threads.

Ex:

```
class ThreadA extends Thread  
{  
    public void run()  
    {  
        for( int i=100; i<140 ; i++ )  
            System.out.println(" run of ThreadA : "+i);  
        System.out.println(" end of ThreadA");  
    }  
}  
  
class ThreadB extends Thread  
{  
    public void run()  
    {  
        for( int i= 400; i< 437; i++ )  
            System.out.println(" run of ThreadB : "+i);  
        System.out.println(" end of ThreadB");  
    }  
}
```

```

class ThreadC extends Thread
{
    public void run()
    {
        for( int i = 4 ; i < 40 ; i++)
            System.out.println(" sum of ThreadC : " + i);
        System.out.println(" end of ThreadC ");
    }
}

```

```

class ThreadTest
{
    public static void main ( String args[])
    {
        ThreadA ta = new ThreadA ();
        ThreadB tb = new ThreadB ();
        ThreadC tc = new ThreadC ();
        ta.start();
        tb.start();
        tc.start();

        for( int j = 0 ; j < 40 ; j++)
            System.out.println(" Inside main : " + j);
        System.out.println(" end of main ");
    }
}

```

NOTE: The output of the above program can not be guessed in advance because, once the start() method hands over the run() method of the corresponding object to the local OS, it is upto the local OS to schedule them. This is why the output can not be guessed.

On the above example, we have defined and implemented multiple threads by extending the Thread class.

Now, the question is:

Q How can you define and implement multiple threads when your class is extending another class other than Thread class?

NOTE: In java, a class can not extend two classes simultaneously.

Auf: when our class is extending a class other than Thread class, then we define and implement thread(s) by implementing the Runnable interface to our class.

NOTE: Normally, we follow only this process to define and implement threads.

Ex:

```
class ThreadA implements Runnable  
{  
    public void run()  
    {  
        for( int i=0 ; i<40; i++)  
            System.out.println(" run of ThreadA : " + i);  
    }  
  
    public static void main( String args[] )  
    {  
        // code... (P.T.O)  
    }  
}
```

```

class.      public static void main (String args[])
ds          {
            ThreadA ta = new ThreadA ();
            Thread t1 = new Thread (ta);
            t1.start ();
            for (int s= 400; s< 437; s++)
                System.out.println ("main : " + s);
        }
am.    }

```

Explanation of the above program:

- Remember that run() is defined in Runnable interface and start() is defined in Thread class.
- we are only implementing Runnable interface, but we are not extending Thread class to our class.
- Duty of start() method is to handover the run() method ^{to local OS} present in the object on which the start() method is called. to the local OS for scheduling.
- Since start() method is present in Thread class, we have to create object of Thread class to get access to its start() method.
 (This is done in the second statement of main()
 in the above program)
- Now, when we say t1.start(), start() method searches for run() method defined in t1 object.

contd... (P.T.O)

- it will not find any, because, the `run()` method is present in `ta` object (i.e. object of class `ThreadA`). and `ta` object is not available to `t1` object.
- in order to execute the `run()` method as a thread, it has to be invoked through the `start()` method. we do not have any other alternative.
- so, if `run()` method is to be available to the `start()` method, obviously, `ta` object should be available to `t1` object.

Q How can you make object of one class available to object of another class without using 'extends' keyword?

Ay: whenever we need to make object of one class available to object of another class, then we make use of constructors.

- constructor of `Thread` class is defined to accept object of `Runnable` interface.
 \therefore only if `ThreadA` class implements `Runnable` interface, then we can pass object of `ThreadA` class as an argument to the constructor of `Thread` class.

Hence, the statement `Thread t1 = new Thread(ta);` in the program.

- Now, when we write `t1.start();` it searches for `run()` method in `t1` object and since `ta` is available to `t1`, `run()` of `ta` will be handed over to the local os.

- To control the execution of threads, we have to wait by suspending.
- Some times, it is necessary to control the ~~execution~~ ~~availability~~ of the threads.
- For that purpose, we have some functionalities in the Thread class which help us in gaining control over the execution of run() methods which are already scheduled by the local operating system.
- A thread which is already under execution can be suspended (prevented from getting executed further) based on three criteria:
 - (i) time (ii) conditionally (iii) unconditionally.

(i) Suspending a thread based on time:

- There is a static function by name sleep() in Thread class which takes time (in milliseconds) as an argument

Ex: Thread.sleep(1000);

- The current thread under execution, in which sleep() method is encountered, will be suspended for those many milliseconds passed as argument to the sleep() method.
- sleep() method is prone to generate checked exceptions. Hence, it should be called inside try block.

(ii) Suspending a thread conditionally:

Let us assume we have three threads available, namely - t₁, t₂, t₃.

Now, if we have a condition that, inside run() method of t₃, we need to use some of the values calculated in run() method of t₁, then in this case, we have to stop execution of run() method of t₃ until run() method of t₁ is completely executed.

In such situations, we make use of join() which is a non-static method of Thread class.

(iii) Suspending a thread unconditionally:

In order to suspend a thread unconditionally, we make use of non-static method suspend() present in Thread class.

Alternatively, in order to resume a thread which has been suspended long back we use resume() method.

NOTE : suspend() and resume() are deprecated in the current version.

The alternate methods are wait() and notify(). These are the non-static methods, defined in the Object class.

- Thus, all the above methods are used to control the execution of the threads according to our requirement which are already under execution.

23.08.2022

- implementing the sleep() method:

```

class Sdemol extends Thread {
    public void run() {
        try {
            Thread.sleep(837);
        }
        catch (Exception e) {
            System.out.println(e);
        }
        for (int i=0; i<37; i++) {
            System.out.println("run of Sdemol : " + i);
        }
        System.out.println("end of run()");
    }
    public static void main (String args[])
    {
        Sdemol s1 = new Sdemol();
        s1.start();
        for (int i=400; i<440; i++)
            System.out.println("Inside main : " + i);
        System.out.println("end of main");
    }
}

```

- we know that, Thread sleep() method is prone to generate checked exception. Therefore the function call should be made inside the try block.

contd... (P.T.O)

- suppose, if we want to skip the try block, then we should use 'throws Exception' in the function signature in which sleep() method is called. (here it is run() method)
- But, this leads to a problem, because, if we use 'throws Exception' in the definition of run() method, then, we directly violate the concept of overriding.
- The signature of run() method defined in Thread class is just public void run(). It does not have any 'throws Exception'. Now, if we add this to our run method, then we are not overriding run() of Thread class.
- ∵ when we call start() method, it searches for the overridden run() method and when it does not find any, it gives a compilation error.
- Thus, 'throws Exception' can not be used with run() method and compulsorily we have to use try - catch inside run() in order to call the sleep() method.
- we know that main() is also thread and other threads get executed simultaneously with main(). Suppose, if we use Thread.sleep() in main(), then what happens? (Refer to the program on next page.)

```

class Sdem02 extends Thread {
    public void run() {
        for( int i=0 ; i<40 ; i++ )
            System.out.println("Inside run : " + i);
    }
}

public static void main (String args[])
{
    Sdem02 s1 = new Sdem02();
    s1.start();
    try {
        Thread.sleep(800);
    }
    catch (Exception e){ System.out.println(e);}
    for (int i =100 ; i<140 ; i++)
        System.out.println("Inside main : " + i);
    System.out.println("end of main");
}
}

```

In the above program, we are calling `sleep()` method in `main()`. Also remember that `run()` is called through `start()` from `main()` only.

Q: Since `main()` is being suspended, will `run()` also be suspended just because it is indirectly invoked from `main()`?

Ay: No, because, `main()` and `run()` are two separate threads. And suspension of one thread will not effect the other. Each executes individually.

NOTE: we can call sleep() method wherever we want.
we can call it in functions, main() method and even
in constructors.

In the following example, sleep() is called in a function.

```
class Sdemo3 extends Thread [Sdemo3.java]
{
    void function (String s1)
    {
        for (int i=0; i<22; i++)
        {
            System.out.println(s1+i);
            if (i == 25)
            {
                try {
                    Thread.sleep(440);
                }
                catch (Exception e)
                {
                    System.out.println(e);
                }
            }
        }
    }

    public void run()
    {
        function("run: ");
    }

    public static void main (String args[])
    {
        Sdemo3 s = new Sdemo3();
        s.start();
        s.function("main: ");
    }
}
```

- implementing join() method:

```
class Thread1 extends Thread  
{  
    int sum;  
    public void run()  
    {  
        System.out.println("Beginning of run of Thread1");  
        for(int i=0; i<37; i++)  
        {  
            sum = sum + i;  
            System.out.println("run of Thread1 : "+i);  
        }  
    }  
  
    class Thread2 extends Thread  
    {  
        public void run()  
        {  
            for (int i=0; i<137; i++)  
                System.out.println("run of Thread2 : "+i);  
        }  
    }  
  
    class Thread3 extends Thread  
    {  
        Thread1 t1;  
        Thread3(Thread1 t1)  
        {  
            this.t1 = t1;  
        }  
    }  
}
```

contd... (P.T.O)

```
public void run()
{
    for(int i=400; i<437; i++)
    {
        System.out.println("run of Thread " + i);
        if(i == 422)
        {
            try
            { t1.join(); } catch (Exception e) { System.out.println(e); }

            int x = t1.sum - i;

            System.out.println("sum : " + t1.sum);
            System.out.println("difference : " + x);
        }
    }
}
```

```
class JoinTest
{
    public static void main (String args[])
    {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        Thread3 t3 = new Thread3(t1);
        t1.start(); t2.start(); t3.start();
    }
}
```

Explanation:

We know that join() method is used to suspend a thread conditionally.

In the above example also, we did the same.

We want to ~~use~~ sum value which is calculated in threads to be used in thread 3.

∴ In thread 3 (i.e. run of Thread 3), we call join of on Thread 1 (i.e. t1.join()).

- Functionality of join() is to check whether run() of t1 is completely executed or not. As long as run() of t1 is not completely executed, control will not come out of it. Thus, thread 3 i.e. t3 will be under suspension. Once control comes out of run() of t1, then it starts executing the statements present in the run() method of current thread (in this case it is t3).

Q: In the above program, we have four threads - t1, t2, t3 and main(). What happens, if main() gets executed first? Is it going to terminate the program?

A: No. Every thread in the program, would be by default joined as the last statement in main().

This ensures that, before control comes out of main(), all threads are completely executed and only then the program will be terminated.

question: what can be the consequence when multiple threads act upon a single object?

Let us assume, we have a class Common which has a function fun1(). Let this fun1() be available to three threads - t1, t2, t3.

```
class Common
{
    public void fun1(String s1)
    {
        System.out.print(" [Hello");
        try {
            Thread.sleep(1822);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.print(" [" + s1 + "] world]");
    }
}
```

/* The above function is supposed to give an o/p as

[Hello [<string that we pass to s1>] world] */

Now, let us see what happens when two threads act on this function simultaneously.

```
class ThreadA extends Thread  
  
    Common c;  
  
    ThreadA (Common c)  
    {  
        this . c1 = c1;  
    }  
  
    public void run()  
    {  
        c1 . fun1 ("Java");  
    }  
}  
  
class ThreadB extends Thread  
  
{  
    Common c1;  
  
    ThreadB (Common c1)  
    {  
        this . c1 = c1;  
    }  
  
    public void run()  
    {  
        c1 . fun1 ("C++");  
    }  
}
```

contd ... (P.T.O)

```
class CommonTest
```

CommonTest.java

```
{ public static void main (String args)
{
    Common c2 = new Common ();
    ThreadA ta = new ThreadA (c2);
    ThreadB tb = new ThreadB (c2);
    ta.start ();
    tb.start ();
}
```

Now, we get an o/p as follows after we compile and run `CommonTest.java`.

[Hello [Hello [java] world]

[c++ world]. which is not the required o/p.

This is because the two threads `ta` and `tb` are entering into `fun1()` and acting upon it simultaneously.

- we can get the desired output if we avoid `ta` and `tb` acting on `fun1()` simultaneously.

This is nothing but, the concept of Threadsafe.

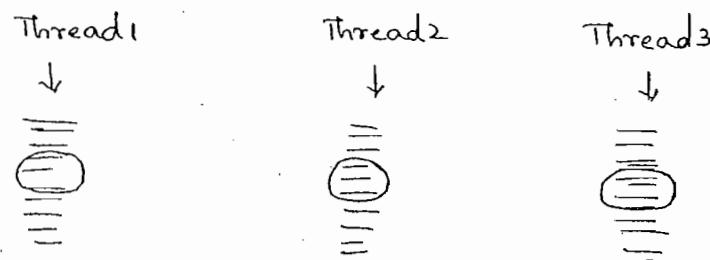
Threadsafe: It is the concept of avoiding multiple threads acting upon the same functionality simultaneously.

- Thread safety is achieved by using a keyword 'synchronized'.
 - If we use, 'synchronized' keyword in the signature of the function on which multiple threads are acting upon, then, it will not allow ~~another~~ ^{a new} thread to act upon the function, until the thread which is ^{already} acting upon the function, comes out of it.
 - Thus, `fun()` of class Common in the above example should be modified as follows.
- public synchronized void fun (String s) { }
- Now, if we recompile and re-execute the same program, then we get the desired output.

24-08-2006

Q: Actually, multitasking is the concept of executing more than one functionality simultaneously. But, by using 'synchronized' keyword, we are avoiding the same. Then what is the use of 'synchronized' keyword?

Consider the following scenario.



contd ... (P.T.O)

Let us assume we have three threads - Thread1, Thread2 and Thread3.

~~each of these runs) - methods~~

The run methods of those threads have some statements and the circled statements are common in all the run methods.

Now, the condition is when those statements of one thread are being executed, another thread should not execute those same statements, until the previous thread finishes the execution of those statements.

In such situations, we use the 'synchronized' keyword.

use of 'Synchronized' keyword:

- (i) synchronized keyword can be used with a functionality.
- (ii) It can be used with a ^{block of} statements inside a function.

Ex:

```
public void function()
```

```
{
```



synchronized { } } → This is a synchronized block.

```
}
```

- Till now, we have seen threads getting created simultaneously.
- Suppose, if we want the threads to get executed according to our logic and specifications, then how should we define the threads?

(or)

How can you gain partial control on the scheduling mechanism which is supervised by the local OS?

For achieving this we make use of some methods:

(i) wait() method: As it is a method belonging to the object class.

Whenever we need to suspend a thread unconditionally, then we use the wait() method.

(ii) NOTE: wait() method works only in synchronized blocks.

(ii) notify() method: This method is also present in the object class.

Whenever we need to resume a suspended (waiting) thread ~~and suspend the current thread~~, then we use notify() method.

- To gain the above mentioned 'partial control' the threads should have a sign of mutual understanding between them i.e. they should be able to communicate with one another.

This is known as "Thread Synchronization" or "Inter Thread communication".

- The producer-consumer problem is an excellent example which illustrates Inter Thread Communication.

(P.T.O)

Producer - Consumer Problem :

- we have two threads - producer and consumer.
- also we have a common class object on which these two threads act upon.
- The common class has two variables (int i & boolean flag) and two methods - produce() and consume.
- now, our requirement is as follows:

producer thread should call the produce() method and by doing so, it has to generate a number.

After this, consumer thread should call the consume() method. consume method should return the value and this should be printed on the console by the consumer thread.

- condition is, no number should be generated twice and no number should be printed twice on the console. That means, first, producer thread should produce a number. After this, it has to notify the consumer thread and it (producer thread) should go into wait state.

Meanwhile, consumer thread should print the produced value on the console and then it should go into wait state. Before going to wait state, it should notify the producer thread so that producer thread will generate a new value and this process continues....

- To keep the value intact we will use the
boolean flag variable.

If flag is true : producer thread should publish the
value and consumer thread should wait.

If flag is false : consumer thread should consume the
value and producer thread should wait.

From this, it is clear that, run() method of the
producer thread should always be scheduled first.

∴ Initially, we set boolean flag = true.

- The algorithm for the produce method is as follows-

```
public synchronized void produce (int i)  
{  
    step-1 : if (flag is true)  
    {  
        value ← i  
        flag ← false  
        notify();  
        wait();  
    }  
    step-2 : else  
    {  
        wait();  
    }  
}
```

NOTE : we should use 'synchronized' keyword in the function
signature because, we are using wait() method inside
the function and wait() method works only in
synchronized blocks.

NOTE: we should always call `notify()` method before calling the `wait()` method. Why?

Let t_1 and t_2 be two threads which need to communicate with one another.

Suppose, let t_2 be already in wait state.

Now, in t_1 , if we call `wait`, t_1 will go into wait state without notifying t_2 . Thus, both t_1 and t_2 will go into wait state and they won't resume until notified which is not done elsewhere in the program.

Also, there won't be any use even though, if we call `notify()` after `wait()`.

\therefore `notify()` method should be called before `wait()` method is called.

- The overall functionality of the `produce()` method is to receive a value from the producer thread and assign this value to the ~~variable~~ instance variable i . Resuming and suspending the respective threads is also a part of its functionality.
- Now, when ^{consumer}thread calls the `consume()` method, then `consume()` method should get hold of the value in the ~~instance~~ variable i and this value should be returned to the consumer thread which prints it on the console. Here also, resuming and suspending the corresponding threads is a part of its functionality.

- The algorithm for the consume method is as follows.

```
public synchronized int consume() {  
    step-1: if (flag is true)  
        {  
            wait();  
        }  
  
    step-2: set flag to true  
    step-3: notify()  
    step-4: return (value);  
}
```

- The total java program for the producer-consumer problem is as follows:

```
// The common class on which producer & consumer threads act upon //
```

```
class Common {  
    static int value;  
    boolean flag = true;  
    public synchronized void produce (int i) {  
        if (flag == true)  
        {  
            value = i;  
            System.out.println ("producer produced : " + value);  
            flag = false;  
            notify();  
        }  
        try {  
            wait();  
        }  
        catch (Exception e) { System.out.println (e); }  
    }  
}
```

```
public synchronized int consume()
{
    if (flag == true)
    {
        try {
            wait();
        }
        catch (Exception e)
        {
            s.o.p(e);
        }
    }
    flag = true;
    notify();
    return (value);
}
}
```

// class.

// The producer thread class.

```
class Producer extends Thread
{
    Common c;
    Producer (Common c)
    {
        this.c = c;
    }
    public void run ()
    {
        int i=0;
        while (true) // An infinite loop which produces
                     0, 1, 2, 3, .....
        {
            c.produce (i);
            i = i+1;
        }
    }
}
```

```

    try
    {
        Thread.sleep(1822);
    }
    catch (Exception e)
    {
        s.o.p(e);
    }
} //while

} //run

} // class.

```

// The consumer thread class.

```

class Consumer extends Thread
{
    Common c;
    Consumer (Common c)
    {
        this.c = c;
    }
    public void run()
    {
        while (true) // infinite loop which prints (consumes) the
        {
            int i = c.consume();
            s.o.p("consumer consumed : " + i);
            try {
                Thread.sleep(1822);
            }
            catch (Exception e)
            {
                s.o.p(e);
            }
        } //while.
    } //run
} //class

```

// The program which invokes the threads.

ProducerConsumerTest.java

```
class ProducerConsumerTest  
{  
    public static void main (String args[])  
    {  
        Common c1 = new Common ();  
        Producer pr = new Producer (c1);  
        Consumer co = new Consumer (c1);  
        pr.start ();  
        co.start ();  
    }  
}
```

Comments: Now local os may schedule either of the threads first. But we have developed the logic in such a way that, run() method of producer thread is always executed first. This was achieved with the help of Inter Thread communication.

- The output of the above program would be as follows:

producer produced : 0

consumer consumed : 0

producer produced : 1

consumer consumed : 1

⋮

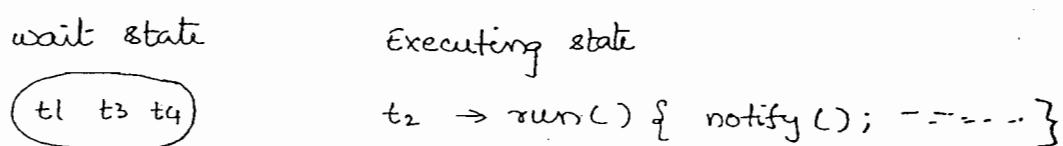
This continues till we terminate the program by pressing ctrl c.

- In this way, producer - consumer problem stands as an example for Inter Thread communication (or) Thread Synchronization

a **NOTE :** (i) `notify()` method resumes only one thread which was already in wait state.

(ii) `notifyAll()` method resumes all the threads that are in wait state.

Now consider the following scenario.



Totally, we have four threads - t_1, t_2, t_3, t_4 .

t_1, t_3, t_4 are in wait state. Only t_2 is in execution.

Q Now, if we call `notify()` in t_2 , which thread is it going to resume? Is it going to resume the thread that has entered into the wait state long back?

A: NO. It is completely decided by the local OS.

Thus, because of this improper ~~scheduling~~ ^{notifying}, sometimes, we may not be able to achieve inter thread communication when there are more than two threads.

To overcome this problem we assign priorities (or) preferences to the threads.

- Thread Priorities :

- Priorities are integer values which range from 1 to 9.
- 1 indicates lowest priority and 9 indicates highest priority.
- Thus, when we have multiple threads in wait state and if we call `notify()` in the current thread, then the thread with highest priority will be notified first.

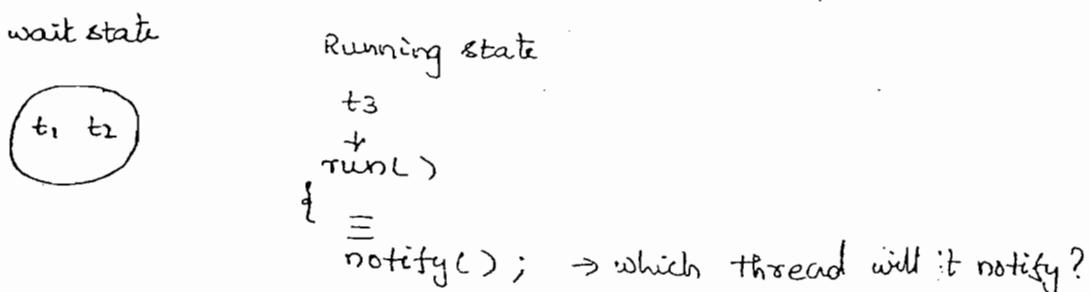
- Priorities should be assigned to the corresponding threads before calling the `start()` method.
 - To assign the priorities to threads, we use the `setPriority()` method. It is a non-static function which accepts an integer value (1 to 9) as an argument.
 - Priority concept can be used only when there is a possibility for the threads to enter the wait state.

NOTE: local OS will never schedule the threads depending upon their priorities.

consider the following scenario

$t_1, t_2, t_3 \rightarrow$ threads

t1.setPriority(4); t2.setPriority(9); t3.setPriority(?);



We have three threads t_1 , t_2 and t_3 and their priorities are set as shown above.

Let us assume that t_1 and t_2 are in wait state and t_3 is in running state.

Now, if `notify()` is encountered in the `run()` method of `t3`,

then it obviously notifies t_2 because, t_2 has highest priority.

- If we don't set a priority to a thread explicitly, then every thread will have a default priority. The default priority is none other than the priority of the parent thread from which the current thread is spawned.

NOTE: The default priority of the main() thread is 5. Thus, the default priority of all the threads which are spawned from main will be 5.

- As an absolute value, a priority is meaningless: a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

The rules that determine a context switch are simple:

- (i) A thread can voluntarily relinquish control. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU time.
- (ii) A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted (no matter what it is doing) by the higher priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

NOTE: In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

- thus, we should always be careful regarding thread priorities. We should see that no two threads will have the same priority.
- DeadLock:

A special type of error that we need to avoid which specifically relates to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, let us assume one thread enters a synchronized method of object X and another thread enters a synchronized method of object Y.

Now, if the thread in X needs to call the synchronized method in Y, it will not get access because the method in Y is synchronized and already a thread is acting on it.

However, if the thread in Y, tries to call the synchronized method in X, it waits forever, because to access the method in X, it would have to release its own lock on the Y object so that thread acting on X could complete, and this will not happen.

Hence, there is a circular dependency of X and Y on each other and it results in a deadlock.

- Deadlock is a difficult error to debug for two reasons:
 - (i) In general, it occurs only rarely, when the two threads time-slice in just the right way.
 - (ii) It may involve more than two threads and two synchronized objects.

NOTE: Earlier, suspend() and resume() methods were used in the process of establishing inter-thread communication. But suspend() method has been highly proven to ^{cause} generate the deadlock situations. So, these two methods have been deprecated in the current version.

In place of suspend() and resume(), we use wait() and notify() respectively.

(P.T.O)

daemon threads:

consider the following example.

JVM executes a .class file and while in this process, many objects will be created. Simultaneously, garbage collector collects all the unreferred objects in the background. Thus, garbage collector is a background process.

- In other words, garbage collector is an infinite loop program and runs as long as JVM is in execution. That means garbage collector thread provides its services in the background to support the JVM thread, and as long as JVM thread is in execution, the garbage collector thread will also be in execution.
- In this case, garbage collector thread is nothing but a Daemon thread.
- Concept of Daemon threads is to design a thread with infinite loop which would be supporting the parent thread as a background process and it is supposed to get executed along with the parent thread. Also, this background thread should stop when the parent thread stops.
- we can define daemon threads by using `setDaemon()` method.
- If we want to set a thread as Daemon thread, then pass boolean value 'true' to the method, as shown below.

```
obj.setDaemon(true);
```

NOTE : Daemon threads would never be joined. Not even in main (which is the default join).

Once the control comes out of the parent thread, the daemon thread is automatically terminated.

Example:

```
class Daemondemo extends Thread  
{  
    public void run()  
    {  
        int i=0;  
        while (true)  
        {  
            i = i+1;  
            System.out.println("Daemon : "+i);  
        }  
    }  
    public static void main (String args[])  
    {  
        Daemondemo d = new Daemondemo();  
        d.setDaemon (true);  
        d.start();  
        for (int i=100; i<137; i++)  
        {  
            System.out.println("main : "+i);  
        }  
        System.out.println("end of main");  
    }  
}
```

Life cycle of a thread

Threads exist in several states.

A thread can be running. It can be ready to run as soon as it gets CPU time.

A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.

A thread can be blocked when waiting for a resource.

At anytime, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

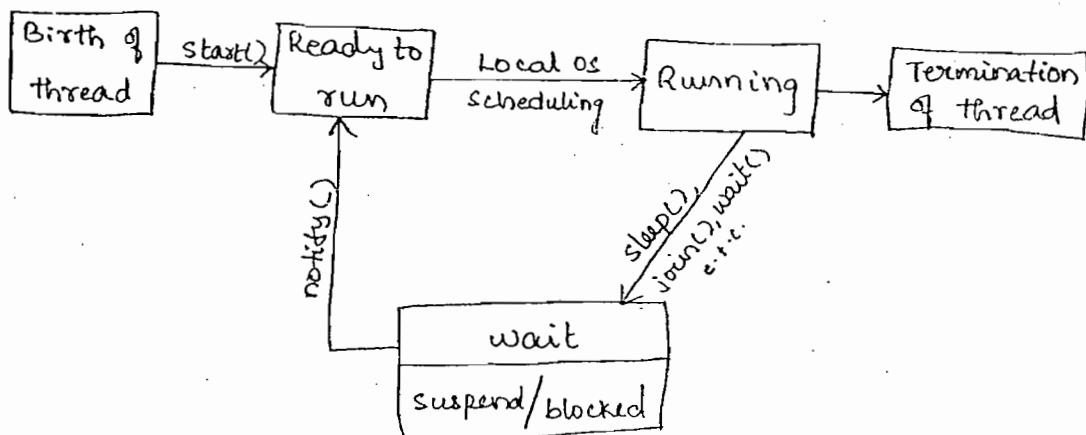


fig: Block diagram of life cycle of a thread.

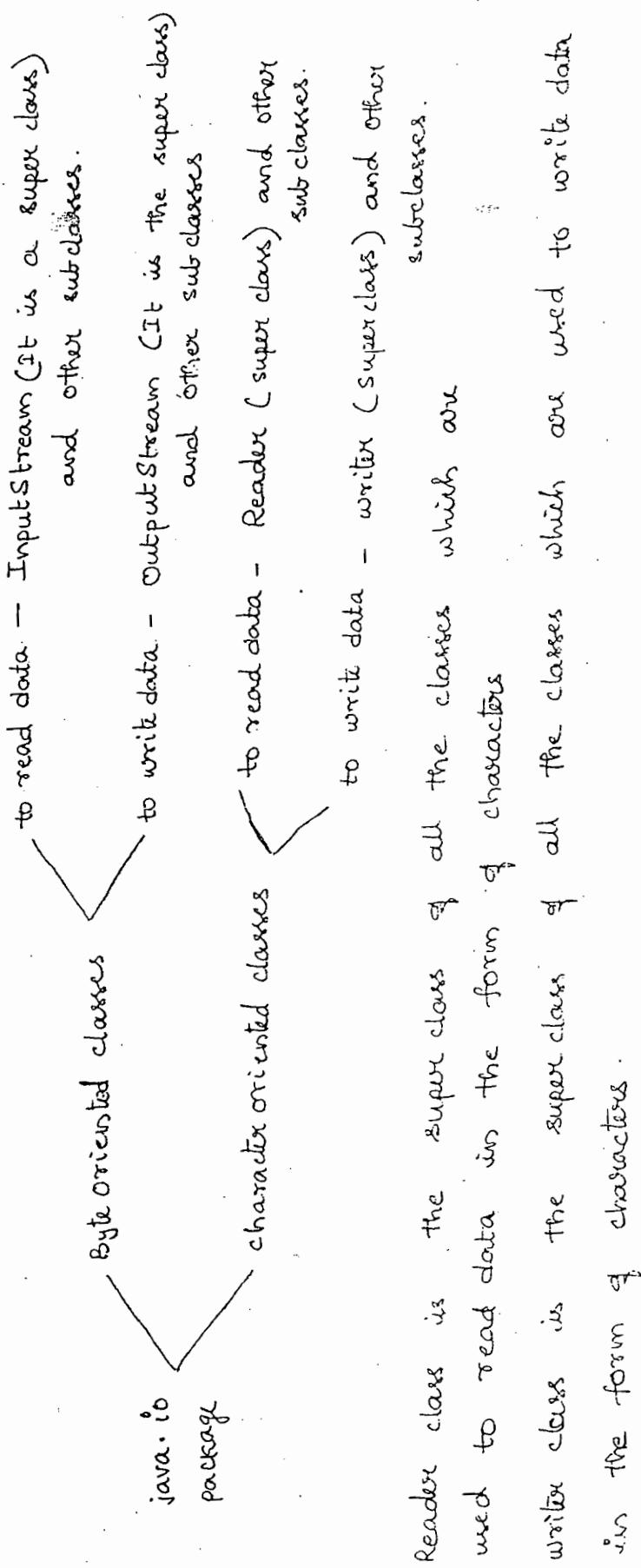
I/O Streams

Stream: A Stream is defined as a continuous flow of data from one place to another.

- Here the flow of data is in between the program and peripherals. Hence the name I/O streams.
- Thus, in java programming, whenever we need to make a data transfer we use streams.
- The java package that supports streams is `java.io`.
The `java.io` package contains many classes using which we make the data transfer.
- For data transfer to be done, first of all the data should be represented in its low level equivalent i.e. either in the form of bytes (8 bits) or in the form of characters (16 bits).
- Depending upon the representation of the data to be transferred, the classes in the `io` package are classified into two categories.
 - (i) Byte oriented class (ii) character oriented classes.
- The byte oriented classes are again classified into two sections - classes using which we can read data and classes using which we can write data.
- Similarly, the character oriented classes are also classified into two sections - classes for reading data and classes for writing data.

InputStream class is the super class of all the classes which are used to read data in the form of bytes
OutputStream class is the super class of all the classes which are used to write data in the form of bytes.

The class hierarchy is shown below.



Identifying the classes:

we can identify, whether a class is a byte oriented class or a character oriented class by with the help of the following rules, but these may not be 100% true, always.

- If the last part of the name of the class is 'stream' then we can say that, the given class is a byte oriented class.

Ex : DataInputStream, DataOutputStream, FileInputStream, PrintStream e.t.c.

- Similarly, if the last part of the name of the class is 'Reader/writer' then we can say that, the given class is a character oriented class.

Ex: BufferedReader, FileWriter e.t.c.

- whenever, it is required to transfer the data, then we should mention from where to where the transfer is to be made i.e. we should mention the address of the source and destination.

Q: How can you represent the addresses?

Ay: By creating objects of those classes which contain the addresses of the sources/destinations.

Q: what are those classes?

Ay: Those classes are nothing but the byte oriented classes and character oriented classes.

- For example, whenever we want to transfer data in byte format, then we use the `InputStream`, `OutputStream` and `ObjectStream` classes.
- Thus, `InputStream` class object is used to represent address of the resource and `OutputStream` class object is used to represent the address of the destination.
- Consider the statement: `System.out.println("sandeepr");`
we know that `println` method is a non-static method present in `out` object and `out` is an object of `PrintStream` class and `out` is declared as a static variable in `System` class.

Question: Similar to 'out' can we create an object of `PrintStream` class and call `println` method on that object to print data on the console as shown below?

```
class Test
{
    public static void main(String args[])
    {
        PrintStream ps = new PrintStream();
        ps.println("HelloWorld");
    }
}
```

Ans: We can't use `ps.println("...");` because, we are indicating a data transfer but we are not

mentioning the address of the destination. Above all, we can't create object of the PrintStream class as we did in the program, because, the PrintStream class has no constructors which are defined to accept zero arguments.

NOTE: PrintStream class has three constructors. All the three constructors are defined to accept object of OutputStream class as an argument in common, along with other arguments.

∴ To mention the destination address we have to create the object of the OutputStream class and this should be made available to the object of the PrintStream class using which the data is transferred to the destination. So, now, we should create object of OutputStream class.

NOTE: But we can't create object of OutputStream class directly by using 'new' operator because, it is an abstract class.

For a moment, let us assume that, we have created the object of OutputStream class as,

OutputStream os =;

Now, the object os ~~is~~ contains the address of the destination and this object is made available to the object of the PrintStream class as shown below.

PrintStream ps = new PrintStream (os);

In this way, we make address of the destination available to the `println()` method (by calling `println()` method on the `PrintStream` class object `ps`), so that it sends data to the corresponding destination.

Question: How can you create object of the `OutputStream` class which is an abstract class?

Ans: we can create object of `OutputStream` class with the help of an anonymous inner class.

```
class System
{
    public static final PrintStream out
        = new PrintStream(new OutputStream() { });
}
```

- In this way, we create object of `PrintStream` class by passing the object of `OutputStream` class as an argument. The object of the `OutputStream` class contains the address of the destination.
- The address of the destination is made available to the object of the `OutputStream` class through an anonymous inner class. (we do so because `OutputStream` class is an abstract class).

- Therefore, in the statement `System.out.println("...");`, we make address of the destination available to the 'out' object and on this object we call the `println()` method which transfers the data from the program to the console.
- Now, it is obvious that, the anonymous inner class contains the address of the VDU (visual display unit). That is why, whenever we use `s.o.p("...");`, we see the output on the console.

NOTE : we use `System.out` ... because 'out' is a static variable in `System` class and we know static members are accessed using the name of the class; when we access them from outside the class.

- we have different methods to read the data in different formats - bytes, characters & strings.

For example,

`read()` method is used to read data in bytes
`readLine()` method is used to read data as strings.

- The `readLine()` method is defined in both the byte oriented classes as well as the character oriented classes.

(P.T.O)

- Any class which supports the functionalities using which we can read the data from the resources, would be defining a constructor to accept object of InputStream class.
- Similarly, any class which supports the functionalities using which we can write the data to the destination, would be defining a constructor to accept object of OutputStream class.

29-08-2006

- Address of any resource from which we need to read data is represented in the form of an object and this object is nothing but the object of InputStream class.
- Similarly, address of any resource destination to which we need to write data is represented in the form of an object and this object is nothing but the object of OutputStream class.
- 'in' is an InputStream class object and is defined as a static variable in the System class.
- Now this object (i.e. in) will be passed as an argument to the DataInputStream class. Now, ~~we~~ we make a method call on the object of DataInputStream class to read the data.

- Generally, we use the `readLine()` method to read the data from a resource. We learnt that `readLine()` method reads the data in the form of strings. This is not true. The actual process is, it reads data in the form of bytes and converts them into string format and thus, we are able to see the data as strings after reading the data from the resource.

Question: Write a program to read data from the keyboard.

Ans:

```
import java.io.*;
class ReadKeyboardData
{
    public static void main (String args[])
    {
        DataInputStream dis = new DataInputStream (System.in);
        System.out.print ("Enter the data: ");
        try {
            String s1 = dis.readLine();
            System.out.println (s1);
        } catch (Exception e) { System.out.println (e); }
    }
}
```

We need to observe two important issues in this program:

- (i) As long as the user is entering the data, the program should wait. That means, the `readLine()` method makes the `main()` thread go to the wait state till the

user enters all the data and ^{presses} clicks the enter key.

- (ii) when we compile the program; it gets compiled but it gives a message saying that we are using deprecated methods.
- In other words, `readLines()` method of `DataInputStream` class is deprecated. So we are not supposed to use it.
- But, since we want to read the data in the form of strings, we use the `readLine()` method of `BufferedReader` class.
- To use this method, we have to make the address of the resource (i.e. object of `InputStream` class) available to the object of `BufferedReader` class. This is done as follows:

Actually, constructor of `BufferedReader` class is defined to accept object of `InputStreamReader` class and the constructor of `InputStreamReader` class is defined to accept object of `InputStream` class. (We know that object of `InputStream` class represents address of the resource).

- This is how we make address of the resource available to the object on which we call the `readLine()` method.

Example:

```
import java.io.*;
class ReadfromKeyboard
{
    public static void main (String args[])
    {
        /* InputStreamReader isr = new InputStreamReader (System.in);
        BufferedReader br = new BufferedReader (isr); */
        BufferedReader br = new BufferedReader (new InputStreamReader
            (System.in));
        System.out.print ("enter the data : ");
        try {
            String s1 = br.readLine ();
            System.out.println (s1);
        } catch (Exception e) { System.out.println (e) }
    }
}
```

In the above program, we have commented the first two statements inside main because functionally, they are equivalent to the third statement.

We are creating object of InputStreamReader class only to pass it as an argument to the constructor of BufferedReader class. This is the only purpose of creating that object in this program.

contd... (T.O)

- coding standards say that whenever we do not have to use an object of a class more than once, then the object need not have a reference. Instead we can use it directly, since we are using it only once in the program.
- This is the reason why, we have commented the first two statements and wrote the third statement which follows the coding standards.

→ Files :

- whenever we need to store data permanently then we use the concept of files.
- we can read data from an existing file. we can create a file and write data into it. we can copy the contents of one file to another file, e.t.c.
To perform all these operations, we use io streams.
- Some of the classes using which we can operate on files are FileInputStream and FileOutputStream , FileReader and FileWriter e.t.c.
- Objects of FileInputStream and FileOutputStream classes come under byte oriented classes and FileReader & FileWriter come under character oriented classes.

- object of FileInputStream class contains address of the file from which we need to read the data.
- similarly, object of FileOutputStream class contains address of the file to which we need to write the data.

Question: write a program to read data from the keyboard and write this data into a file named file1.txt.

Ans:

```

import java.io.*;
class Filedemo1
{
    public static void main(String args[])
    {
        try
        {
            BufferedReader br = new BufferedReader(new InputStreamReader
                (System.in));
            System.out.println("enter the data : ");
            String s1 = br.readLine();
            // FileOutputStream fos = new FileOutputStream ("file1.txt");
            FileOutputStream fos = new FileOutputStream ("file1.txt", true);
            fos.write(s1.getBytes());
            byte ba[] = s1.getBytes();
            fos.write(ba);
            fos.close();
        }
        catch (Exception e) { System.out.println(e) }
    }
}
  
```

(P.T.O)

Explanation:

- Object of FileOutputStream class (in this case fos) contains the address of Fil1.txt. If the file is not existing, then the file is created with the given name and then its address will be assigned to fos.
- Remember that whenever we open a file in read/write mode, the file pointer always points to the zeroth location.
- In this example we are writing data into the file. For the first time when we open the file and write the data we don't have any problem. But when we open the file for the second time and write the data, the existing data will be over-written because whenever the file is opened, the file pointer points to the zeroth location.
So, it is our duty, ^{to} bring the file pointer to the end of file before writing data into it.
- To do this, we need to pass the boolean value 'true' as an argument to the constructor of the FileOutputStream class along with the name of the file.
- write() is a method present in FileOutputStream class and it writes the data into the file in the

form of bytes. But we read the data from the keyboard in the form of strings. So we need to convert it into bytes.

- For this we use the `getBytes()` function. But, this function returns a group of bytes (i.e. an array of bytes). so the returned byte should be stored in an array (or) in other words, - the return data type of `getBytes()` function is a byte array.
- So, we use `byte ba[] = si.getBytes();`
- Now we pass 'ba' as an argument to the write method i.e. `fos.write(ba);`
- According to coding standards, whenever we use a resource, we should release it after the job is over. Hence `fos.close()` is used.

NOTE: `close()` method deletes the address of the file present in `fos` object and thus releases the file from our programs.

(P.T.O)

30.0

Reading data from a file using a byte oriented class

```
import java.io.*;
class Readfile1 {  
    public static void main (String args[]) {  
        try {  
            FileInputStream fis = new FileInputStream ("File1.txt");  
            int size = fis.available();  
            byte ba[] = new byte [size];  
            fis.read (ba);  
            String data = new String (ba);  
            System.out.println (data);  
            fis.close (); } catch (Exception e) { System.out.println (e); }  
    }  
}
```

Explanation: we have different methods to read data from a file. `read()` method reads the data in the form of bytes i.e. it reads byte-by-byte.

- `read(byte array)`: This read method takes a byte array as an argument and it reads all the bytes at once from the file into the array.
- So first of all we need to create a byte array and its argument this should be passed as an argument to the read method.

- In order to create the array, we need to mention the size of the array. Here the value of size is nothing but, the size of the file.
- How can we know the size of the file? For that we have a function by name `available()`. This method returns the size of the file whose address is present in the object on which this method is called.
- After reading all the data from the file, it should be converted into its equivalent string format.

NOTE : constructor of `FileInputStream` class is defined to accept name of the file as the argument. When we create object of `FileInputStream` class, the constructor is executed. It opens the file in the read mode and the file pointer points to the zeroth location.

Reading the name of the file from the console and displaying the contents of the file on the console

```

import java.io.*;
class Readfile2      Readfile2.java
{
    public static void main (String args[])
    {
        try
        {
            BufferedReader br = new BufferedReader (new InputStreamReader
                (System.in));
        }
    }
}

```

```

s.o.p ("enter the name of the file:");
String fname = br.readLine().trim();
fileInputStream fis = new FileInputStream (fname);
int size = fis.available();
byte ba [] = new byte [size];
fis.read(ba);
String fdata = new String (ba);
s.o.p (data);
fis.close();
}
catch (Exception e) { s.o.p (e); }
}
}

c:\> javac Readfile2.java
c:\> java Readfile2
enter the name of the file: Readfile2.java

```

NOTE: we can enter name of any file which is present in the harddisk. we can even enter Readfile2.class because the .class file will also be present in the harddisk.

- Till now we have operated on the files using the byte oriented classes. Now, we use the character oriented classes.

writing data into a file using character oriented classes

```

import java.io.*;
class Writecharfile           Writecharfile.java
{
    public static void main (String args[])
    {
        try
        {
            String data = "I am Sandeep Dev";
            FileWriter fw = new FileWriter ("File2.txt", true);
            char ch[] = data.toCharArray ();
            fw.write (ch);
            fw.close ();
        }
        catch (Exception e) { s.o.p (e); }
    }
}

```

Explanation:

- By using `FileWriter` class we write data into a file.
- To the constructor of `FileWriter` class we should pass name of the file and boolean value `true` as arguments.

This makes the file to be opened in the write mode and keeps the filepointer at the end of file location so that existing data will not be over-written with the new data that we write into the file.

- we can read the data from the console and that data can be written in the file. But here, for simplicity, we have considered a string class object which already contains some data.
- we have to write the data in to the file in the form of characters, but we have it in the string format. So we need to convert it into character format.
- when we convert a string into character format, we get an array of characters.
- Therefore, we call the `toCharArray()` method on the string class object. This returns an array of characters and these are stored in the character array `ch`.
- Now, this character array is passed as an argument to the `write()` method.
- In this way, we write data into a file using character-oriented classes.

NOTE: whenever a file is created, the last character in the file will be a special character which indicates the end of the file.

ASCII value of end of file character is -1.

Reading data from a file using character oriented classes..

NOTE : To read data from a file, we ~~were~~ ^{passed} an array object as an argument to the `read()` method in the byte oriented classes. We were able to create the array because, we could get the size of the file using a method by name `available()`.

- But in the character oriented classes we don't have any methods using which we can get the size of the file. So the above mentioned logic will not work here.
- Therefore, we have to read each character from the file, convert it into its equivalent string.

```
import java.io.*;  
class Readcharfile {  
    public static void main (String args[])  
    {  
        try  
        {  
            FileReader fr = new FileReader ("fik2.txt");  
            String data = "";  
            int i = fr.read();  
            while (i != -1)  
            {  
                char ch = (char) i;  
                data = data + ch; // concatenating each string equivalent  
                i = fr.read(); // character  
            }  
        }  
    }  
}
```

```
s.o.p(data);  
fr.close();  
}  
catch (Exception e) { s.o.p(e);}  
}  
}
```

Explanation: we have to read the contents of the file as character by character till the end of file is reached.

- `read()` method of `FileReader` class reads the character and returns its integer value.
- In the while loop we check `!= -1` whether that integer value is `-1` or not i.e. checking for end of file.
- If it is not end of file, then typecast the integer value into its equivalent character and append this character to the already read characters which are present in the `String` class object.
- Finally print the data on the console.
- This is how we read data from a file using character oriented class.

- Till now we have been opening the files either in the read mode or in the write mode using the respective classes.
- We have a class which allows to open a file in read-write mode so that we can perform the read and write operations using the same object.

The class is RandomAccessFile class.

- RAF class supports all the functionalities using which we can read or write data from or into a file in the form of their primitive datatypes.
Some of such functions are `writeInt()`, `writeFloat()` etc.
- RAF class has some functionalities which support the manipulation of file pointer.

Ex:

`seek()`: Moves the file pointer to a specific location.

`length()`: gives the length of the file.

`getFilePointer()`: gives the current location of file pointer.

NOTE: RAF class is a byte oriented class.

It reads or writes data internally in the form of bytes.

- Constructor of RAF is defined to accept two arguments

contd... (P.T.O)

(i) name of the file - a string class object

(ii) mode - a string class object.

mode: specifies whether the file has to be opened in the read mode ("r") or readwrite mode ("rw");

use of RAF:

whenever we need to store structured data in a file, then we use Random access file.

NOTE: In older days, RAF was the key issue in developing the project.

Even today, we use RAF in some projects where we want to read the data from the files and store it into the database.

31.08.2006

NOTE:

when the constructor of a class is assumed to throw checked exceptions, then we can't create object of the class directly at the place of declaring the instance variables. Instead, we define the references of that class as instance variables.

- A program using Random Access File.

```
import java.io.*;
class RandomDemo
{
    int idno;
    String name;
    float salary;
    BufferedReader br;
    RandomAccessFile raf;
    RandomDemo()
    {
        try
        {
            br = new BufferedReader(new InputStreamReader(System.in));
            raf = new RandomAccessFile("emp.dat", "rw");
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }

    public void readconcede()
    {
        try
        {
            s.o.p("enter the emp id no : ");
            idno = Integer.parseInt(br.readLine().trim());
            s.o.p(" enter the name of the employee : ");
            name = br.readLine().trim();
            s.o.p(" enter the salary of the employee : ");
            salary = Float.parseFloat(br.readLine().trim());
        }
    }
}
```

```
        catch (Exception e)
    {
        System.out.println (e);
    }

} // end function readconsoledata()

public void writeRecord()
{
    try
    {
        String choice = "yes";
        int size = raf.length();
        raf.seek (size);
        while (choice.equals ("yes"))
        {
            readconsoledata();
            raf.writeInt (idno);
            raf.writeUTF (name);
            raf.writeFloat (salary);
            s.o.p ("More records? (Yes/no) ");
            choice = br.readLine().trim();
        }
    }
    catch (Exception e)
    {
        System.out.println (e);
    }
} // end function writeRecord()
```

```

public void readRecords()
{
    try
    {
        raf.seek(0);
        int size = raf.length();
        while (raf.getFilePointer() < size)
        {
            int idno = raf.readInt();
            String name = raf.readUTF();
            float salary = raf.readFloat();
            System.out.println("Id no:" + idno + "Name :" + name + "Salary : " + salary);
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
} //end function readRecords()

public static void main (String args[])
{
    RandomDemo rd = new RandomDemo();
    rd.writeRecord();
    rd.readRecords();
    rd.raf.close();
} //main()

} //class

```

understanding the program:

- we know that RandomAccessfile class allows us to open a file in read-write mode.
- Now, our aim is to open a file in read-write mode and accept the employee information from the console and then write this information into the file.
- After all the records have been entered and written into the file, now we should read all the records from the file and display them on the console.
- In this program, we have altogether five instance variables. Three of them are related to employee information. Other two are class references.
- In the constructor of the class, we create object of RandomAccessFile class and BufferedReader class.
- we have three functions:
 - (i) readconsoledata(): This function is used to receive the data ^(record) entered by the user in the console.
 - (ii) writeRecord(): This function writes the records into the file. If there are more records, again readconsoledata() function is called. Thus, we call (i) from (ii).

(iii) readRecords(): After all the records are entered into the file, then each record is read from the file and displayed on the screen.

NOTE: The logic is implemented in such a way that, while writing a record into the file, the file pointer will be at the end of file position. While reading the records, the file pointer points to the zeroth location.

Serialization: It is the concept of transferring the objects (data) of classes from the RAM to any other location outside the RAM.

Ex: (i) Transferring state of an object from RAM to harddisk.
(ii) Transferring the state of an object from the RAM to a storage device in another system with the help of network.

01.09.2006

- we can't transfer objects of all classes from RAM to other locations outside RAM.
- There is an interface by name Serializable. It is a zero-body interface i.e. it has no function definitions.
- Objects of classes which implement Serializable interface only, are eligible for serialization.
- ObjectOutputStream class helps us in transferring the objects of classes from our program (i.e. RAM) to other locations.

(P.T.O)

- Similarly, ObjectInputStream class helps us in transferring the objects of classes from other locations to our program (i.e RAM).
- ObjectOutputStream class has a method by name writeObject(). It takes, object of a class which implements serializable interface, as an argument or in other words, it takes, the Serializable interface reference which points to the object of a class that implements this interface, as an argument.
- The object of ObjectOutputStream class on which the writeObject() method is called holds the address of the destination location to which the object is to be transferred.

Ex:

```
FileOutputStream fos = new FileOutputStream ("file1.obj");
ObjectOutputStream oos = new ObjectOutputStream (fos);
oos.writeObject (Serializable s) { A a1 = new A();}
```

on the above example, object of FileOutputStream, fos contains the address of the file which is present in the harddisk. fos is made available to object of ObjectOutputStream class oos. Now on 'oos' we call writeObject() method. In this way address of the destination is made available to writeObject() method in to which the object is to be written.

kring
program

- Interestingly, without knowing the serialization concept, we have made use of it in the RandomAccessfile program.
we have been saving . idno, name and salary of the employee into a file in the harddisk. Note that, name is a string class object. In this way we have saved the object of a class.
- This was possible because String class implements Serializable interface.

NOTE : All the wrapper classes implement Serializable interface.

- A program on serialization

//this program serializes (in this case transfers into harddisk)
an object of a class A. class A should definitely
implement Serializable interface.

```
import java.io.*;  
  
class A implements Serializable  
{  
    int i, j;  
    public void funA()  
    {  
        s.o.p(i);  
        s.o.p(j);  
        s.o.p("Inside funA()");  
    }  
}
```

contd... (P.T.O)

```

import java.io.*;
class WriteObj {  

    public static void main (String args[])
    {
        FileOutputStream fos = FileOutputStream ("file1.obj", true);
        ObjectOutputStream oos = new ObjectOutputStream (fos);
        A a1 = new A ();
        a1.i = 437;
        a1.j = 440;
        oos.writeObject (a1);
        a1.i = 18;
        a1.j = 22;
        a1.funA ();
        Integer in = new Integer (9);
        String s = "Sandeep Dev";
        oos.writeObject (in);
        oos.writeObject (s);
        oos.close ();
    }
}

```

NOTE: In the above program, after serializing the object i.e. oos.writeObject (a1), we are modifying the values of i and j. These are modified in the RAM but not in the harddisk into which we have serialized the object.

- As already mentioned, with the help of ObjectInputStream class, we can read back the objects which we have serialized. This is known as de-serialization.
- we have a function by name readObject() in the ObjectInputStream class. This function reads the objects in the form of objects of Object class.
- To get back the objects we have to typecast the objects of Object class back into the objects of the original classes.
- for this, we need to know the order (hierarchy), in which the objects were serialized.
- The following program de-serializes the objects which we have serialized into file1.obj in the above program writeObj.java.

```

import java.io.*;
class ReadObj           ReadObj.java
{
    public static void main (String args[])
    {
        FileInputStream fis = new FileInputStream ("file1.obj");
        ObjectInputStream ois = new ObjectInputStream (fis);
        Object o = ois.readObject ();
        A a1 = (A) o; // typecasting into the original form.
        a1.funA(); // i=437 ; j=440 will be printed.
    }
}
  
```

contd..(P.T.O)

```

Integer in = (Integer) ois.readObject();
int i = in.intValue();
s.o.p(i); // i=9 will be printed.

String s = (String) ois.readObject();
s.o.p(s); // Sandeep Dev is printed.

ois.close();
}

}

```

File class :

- file class is used to know the characteristics of something present in the harddisk.
- file class has many functions which are used in identifying the features of a file or a folder present in a given location.

Ex: length(): It returns the length of the file.

canWrite(): returns true if the file is of type read-write
returns false if the file is of type read only.

canRead(): returns true if the contents of file can be
read by anyone. It returns false, if the
access to file is restricted.

isDirectory(): returns true if the path specified, leads
to a folder. Else it returns false.

list(): returns an array of strings which would be
containing names of the files present in the
folder specified.

`isfile()`: returns true if the path specified, leads to a file.
else it returns false.

- The path should be specified in the form of a string class object.
- we create object of file class, by passing the path (a string class object) as an argument to its constructor.

Ex : `File f1 = new File("c:\sandip Dev\file2.txt");`
`FileOutputStream fos = new FileOutputStream(f1);`

NOTE : constructor of file class accepts the path (in the form of string) as an argument. After that it checks whether the file or folder is present in the specified path or not. If not, it creates a new file in the path specified.

(P.T.O)

04-09-2006

NETWORKING

Networking: It is the concept of transferring data from one program under execution in one machine to another program under execution in another machine.



Question: How can you transfer the data from one machine to another machine in a network?

Ans: It is possible, if the sending machine and the receiving machine follow a protocol.

Question: What is a protocol?

Ans: Protocol:

- It is a program
- It represents highlevel data in the form of its equivalent machine level data and is responsible for transmitting this data from one domain to another domain (i.e. from one machine to another machine)
- At the receiving end, another protocol program receives this machine level data and represents it in the form of its equivalent highlevel data.
- protocol program is also responsible for controlling the transmission.

Question: why do people say that protocol is an agreed set of rules?

Ans:

- Protocol programs are supplied by 'n' no. of vendors.
- The protocol program, at the sending end (machine) and at the receiving end (machine) need not be from the same vendor.
- Now, if each vendor follows or adapts his own style of representing the data, then it should be compulsory that the protocol programs in the sending machine and receiving machine are from the same vendor. This may not be possible in all the cases.
- So, to avoid this, all the vendors should follow some predefined standard set of rules to represent the data.
- Thus, protocol program is a program which has been developed according to some agreed set of rules.

Classification of protocol programs:

The protocol programs are classified into two categories.

- (i) Connection-oriented protocol program.
- (ii) Connectionless protocol program.

NOTE: Of the above two protocol programs, each has its own advantages and disadvantages.

But, most of the times in the real-time scenario, we make use of connectionless protocol programs.

- Connection oriented protocol programs:
- consider a network comprising of six systems as shown in figure.
- Let us assume that, we need to transfer data from System -1 to System -4.
- The path to transfer data from one system to another system (in this case from S-1 to S-4) in a network would be decided on based on two criteria.
 - (i) Shortest path
 - (ii) Network traffic
- After deciding the path, the data would be transferred in a continuous stream of bytes.

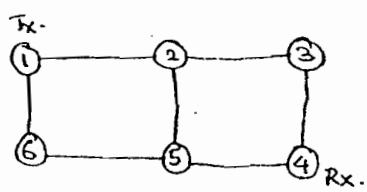
NOTE: This is the reason why we make use of streams in connection oriented protocols to transfer the data.

Advantage: Data, reaching the destination is guaranteed.

Disadvantage: Once the path is decided and data is being transferred, the path can not be changed in between just because, there is heavy traffic burst in that path.

Instead, it waits till the traffic is cleared.

So, in this case, the data is guaranteed to reach the destination but we can't say when the data reaches the destination.



- Connectionless protocol programs:

consider the same example that we have considered in the previous case.

- The path is not pre-determined. It is decided dynamically.
 - Here, after the data being represented in its machine level equivalent, will be broken or divided into groups known as packets.
 - Thus, data is transmitted in the form of packets.
 - Now, each packet may follow its own path in the network, to reach the destination.
 - Each packet will consist of the data, address of the destination and some information regarding the hierarchy of the packet in the original data. This information (i.e. data about data) is known as Metadata.
 - At the destination, each packet is buffered and once all the packets reach the destination, they will be regrouped according to the metadata and then converted into the required format.
- Advantage: Sudden burst of traffic will not affect the transmission because packets have the flexibility to take different paths in the network.
- Disadvantage: Data, reaching the destination is not guaranteed.

Question: How can we identify a particular program to which the data is to be sent, on in the receiving machine?

Ans: This is done with the help of port number.

Port number: It is the logical address of a program which is under execution.

NOTE: whenever a program is under execution, then definitely it would be loaded into a specific location in to the RAM.

Now, the local operating system, maintains a table which has two entries

| Port no | Address |
|---------|---------|
| ... | ... |
| ... | ... |
| ... | ... |

- (i) Port number (ii) Address of the location in the RAM in which the program is under execution.

When the sender sends the data, he mentions the IP address of the machine along with the port no. Local OS maps this port no. to the one present in the table it maintains. When it finds one, there will be a corresponding address of the location of the RAM in which the program is present.

In this way, the data reaches the actual program in the receiving machine.

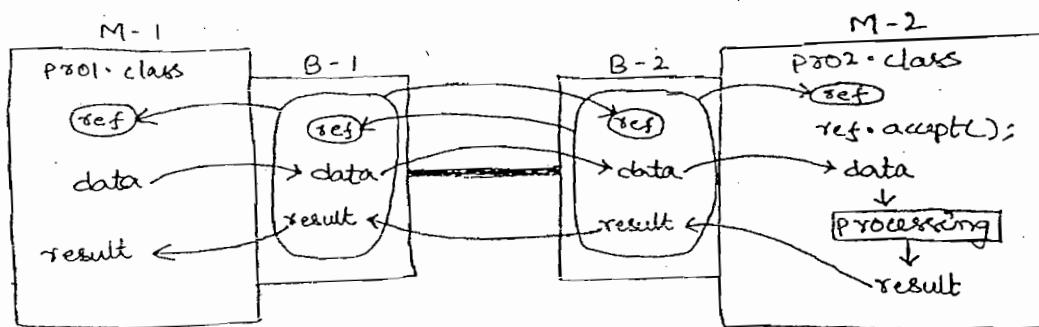
05.09.2006

The program which is supposed to receive the data should be executed first before executing the program which is supposed to send the data.

Question: Explain the process of transferring the data.

Every system will have a buffer memory. The data which is supposed to enter into the system or leave the system will pass through the buffers only.

The following figure illustrates the transfer of data from one machine to another machine.



For convenience, let us assume we have two machines M-1 and M-2 between which data is to be transferred.

Let B-1 be the buffer memory of M-1 and B-2 be the buffer memory of M-2.

For better understanding, let us assume only PRO1-class file is in M-1 and only PRO2-class file is present in M-2.

M-1 and M-2 are connected through the network.

World... (P.T.O)

- Since pro2.class file is supposed to receive the data it should be executed first.
- Now, we create an object of a class in B-2 of M-2
- The reference of let us call this object as Obj-2
- The reference of Obj-2 should be present in pro2.class file. Let this reference be ref-2.
- Now, in pro2.class file, we call the accept() method on this reference i.e. ref-2 = accept().
- The basic functionality of this accept() method is to wait for the connection. Since pro1.class file is not yet under execution, accept() method makes pro2.class file to wait for the data to be received.
- Coming to the transmitting side, we need to do the following.
 - create an object of some class in the buffer B-1 of M-1
 - Let us call this object as Obj-1
- The reference of Obj-1 should be present in pro1.class file. Let this reference be ref-1
- Another important point is, Obj-2 should contain ref-1 and Obj-1 should contain ref-2.
- Once this process is finished, now the data transmission can be achieved.

a

M-2

us

- Let us assume that pro1.class file has to send some data to pro2.class file. Now pro2.class file has to receive this data, process the data and generate the result. This result has to be sent back to pro1.class file.

- To achieve the above goal, we do the following.

Step-1: Using ref-1 place the data in obj-1 of B-1.

Step-2: Already obj-1 contains ref-2 which points to obj-2 so, using ref-2 place the data or move the data from obj-1 to obj-2.

Step-3: Call accept() method on ref-2. With this, data present in obj-2 will be sent to pro2.class file.

Step-4: Process the data and prepare the result. Now place this result in obj-2 using ref-2.

Step-5: obj-2 already contains ref-1 which points to obj-1. So, using ref-1 present in obj-2, transfer the result from obj-2 to obj-1.

Step-6: We know that ref-1 which points to obj-1 is present in pro1.class file. Therefore, using ref-1 collect the result from obj-1 and make it available to pro1.class file.

- In this way we can achieve data transfer between two machines.

NOTE: The objects which we create in the buffers using which we send/receive data to/from the systems across the networks are known as SOCKETS.

Thus, sockets are nothing but objects.

- The transfer of data between the programs which are being executed in different systems is done through the sockets. Hence, this is known as socket programming.

- In the above example, we have considered only one program under execution in either side. But in real-time, we have many no. of programs executing simultaneously.

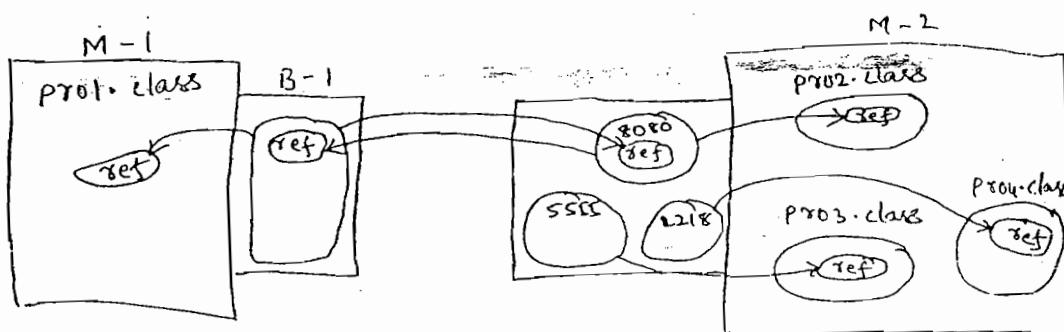
Question: In those many no. of programs, how can we locate the program to which we need to send the data.

(Or)

How can you locate the socket corresponding to a program in the buffer of the receiving machine?

Ans: In order to locate a socket, corresponding to a program in the buffer of the receiving machine, we need to mention the port no. (in which the program is under execution) while creating the in the socket object, while creating the socket object in the buffer of the receiving machine.

- This is illustrated in the following figure.



Server: It is a network related program.

client: It is also a network related program.

Question: What are responsibilities of server and client?

Server:

- Server is always designed to wait for the data first.
- First, it should receive the data.
- Identify the address of the client from where the data has come
- process the data that it has received.
- Send the result of the processed data to the client.

client:

- connects itself to the server program based on the port no. and machine address mentioned.
- Now client sends the data first.
- After that, client receives the processed data from the server.

In a nut-shell, server receives the data first and sends the data next, whereas client sends the data first and receives the data next.

Ex. of Server: Apache Tomcat, weblogic, IIS etc.

Ex. of client: Browser.

- Request: It is the data going from the client program to the server program.
- Response: It is the processed data coming from the server program to the client program.

- Service: The way in which, the data coming from the client, is processed in the server program is known as service.
- Server Socket: Server socket has to wait for the connection. It has to be created with the port number.
- Client Socket: Client socket gets itself connected to the server socket. No port number is required.
- All the classes which support socket programming are present in `java.net` package.
- We create client socket by using `Socket` class.
- Similarly, we create server socket by using the `ServerSocket` class.

Contd... BOOK-3

08.09.2006

The following is an example of socket programming, using connection oriented protocol.

```
import java.io.*;
import java.net.*;
```

```
class ServerDemo1
```

ServerDemo1.java

```
{
    public static void main (String args[])
    {
        /* - Create ServerSocket by mentioning the port number
         * - Make ServerSocket to wait for the connection from the client
         * - As soon as client connects itself to the server, then
         *   read the data from the client address [InputStream]
         * - Now process the data
         * - Send the result of the process back to the client [OutputStream]
        */
        try
        {
            ServerSocket ss = new ServerSocket (5555);
            Socket sc = ss.accept (); // sc contains address of the
                                      // client socket.
            InputStream is = sc.getInputStream ();
            BufferedReader br= new BufferedReader(new InputStreamReader(is));
            String data= br.readLine ();
            System.out.println (data);
            data = "Hai from Server" + data;
            OutputStream os = sc.getOutputStream ();
            PrintStream out = new PrintStream (os);
            out.println (data);
        } catch (Exception e) { s.o.p (e); }
    } //main.
} //class
```

```
import java.io.*;
import java.net.*;
class clientDemo1
{
    public static void main (String args[])
    {
        try
        {
            Socket sc = new Socket ("nit9", 5555); // sc contains address
                                              // of ServerSocket
            String data = "Sandeep Dev";
            OutputStream os = sc.getOutputStream();
            PrintStream out = new PrintStream (os);
            out.println (data);
            InputStream is = sc.getInputStream();
            BufferedReader br= new BufferedReader(new InputStreamReader(is));
            String si= br.readLine();
            System.out.println (si);
        }
        catch (Exception e)
        {
            System.out.println (e);
        }
    } // main
} // class
```

clientDemo1.java

Explanation:

- The program ServerDemo1.java is executed first. Now the accept() method makes this program to wait for some client getting connected to it.
- Once a client gets connected to it, it reads the data from the client, processes the data and then sends it back to the client.
- On the other hand, clients connects itself to a server by mentioning the address of the server machine and port number.
- After this, client sends data to the server and it receives the processed data from the server.
- As, we already mentioned, this is an example of connection oriented protocol. so, the data transfer is in the form of streams.

NOTE: we know that whenever we need to read or write data, we need to mention the address of the source or destination in the form of an InputStream class object or OutputStream class object respectively.

In this example, the source and destinations are objects of Socket class and ServerSocket class. But, the objects of these classes can not be directly represented in the form of InputStream or OutputStream class objects.

Hence we call getInputStream() and getOutputStream() methods on the Server class object and ServerSocket class objects (depending upon the situation) to represent them in the form of InputStream and OutputStream class objects.

Echo Server: client sends some data to the server and if the server sends the same data back to the client, then we call that server as Echo Server.

09.09.2006

NOTE: The server program in the above example is capable of serving only one request.

- To make it handle multiple requests, put the logic inside an infinite loop (i.e. `while(true)`).
- Now, the server program will be capable of handling more than one request. But, this does not solve the problem completely, because the server is now handling multiple requests but it is not able to handle multiple requests simultaneously.
- In real time, we need to develop servers in such a way that they should be capable of handling multiple requests simultaneously. To achieve this goal we need to make use of the concept of multithreading.
- The following program is the same server program in the above example, but with an upgradation i.e. it implements the concept of multithreading.
- Here, we have divided the program into two programs. The first program simply receives the request. The second program is a thread program and the logic of reading the data from the client, processing it and sending it back to the client is incorporated in this program.
- For multiple requests, those many threads will be invoked. In this way, the server will be capable of handling multiple requests simultaneously.

The program is as shown below.

```
- import java.net.*;  
class Server  
{ public static void main (String args)  
{ boolean flag = true;  
ServerSocket ss = new ServerSocket (5555);  
while (flag)  
{ Socket sc = ss.accept();  
Process p1 = new Process (sc);  
p1.start();  
}  
}  
}  
}
```

```
- import java.io.*;  
import java.net.*;  
class Process extends Thread  
{  
Socket sc;  
Process (Socket sc)  
{  
this.sc = sc;  
}  
}
```

contd... (P.T.O)

```

public void run()
{
    InputStream is = sc.getInputStream();
    BufferedReader br = new BufferedReader(new InputStreamReader(is));
    String data = br.readLine();
    data = "Hello, from Server" + data;
    OutputStream os = sc.getOutputStream();
    PrintStream out = new PrintStream(os);
    out.println(data);
} //run
} // class

```

Explanation:

- In the program Server.java, as usual, we create the object of ServerSocket by passing the port no. as an argument.
- On that object, we call the accept method which returns the client socket object.
- Now we pass the object of the Socket class (i.e. ClientSocket) as an argument to the constructor of Process class and create an object of Process class. Note that Process class extends Thread class.
- Now, call the start() method on the object of the Process class. This handles the run() method of the Process class to the local OS for scheduling.

- Thus, a thread is invoked for a request that hasn't come to the server from the client.
- In the run() method, we read the data from the client, perform the necessary processing on that data and then send the result back to the client.
- Meanwhile, when another request comes to the server from another client, then a new thread will be invoked and the above process repeats. In this way, the server handles multiple requests simultaneously.
- Transferring data using Connectionless protocol programming
 - In this, we need to create a socket (i.e. an object) which can transmit data in the form of packets.
 - For this, we make use of DatagramSocket class.
 - To transmit the data in the form of packets, first of all the data should be represented in the form of packets and each packet follows a random path to reach the destination. This indicates that each packet should have the address of the destination in it.
 - So, the constructor of the DatagramPacket class is defined to accept four arguments as shown below.

DatagramPacket (byte ba[], int baLength, InetAddress ia, int portNo);

- The first argument is the data represented in the form of bytes.
- The second argument is the total length of the bytes (data)
- The third argument is the IP address of the destination (Server)
- The fourth argument is the port no. in which the server program is running.

NOTE: In the connection oriented protocol programming, we had directly mentioned the address of the server systems (i.e. "nit-9") while creating the socket object.

But in connectionless protocol programming we need to mention the IP address. The IP address is obtained in the form of an object of the InetAddress class, as shown below.

```
InetAddress ia = InetAddress.getByName("nit-9");
```

NOTE: getByName() is a factory method, because it returns object of the class in which it is present.

- we make use of the send() method of the DatagramSocket class to insert or release the packets into the network.

The following program is an example of connectionless protocol programming.

```
import java.net.*;  
class DatagramClient {  
    public static void main (String args[]) {  
        String s1 = "I am Sandeep Dev";  
        DatagramSocket ds = new DatagramSocket(5555);  
        InetAddress ia = InetAddress.getByName("nit9");  
        byte bai[] = s1.getBytes();  
        DatagramPacket dpl = new DatagramPacket(bai, bai.length,  
                                              ia, 4444);  
    }  
}
```

DatagramClient.java

```
ds.send(dp1);

/* once the send() method is executed, the packets
   are sent. Now we have to receive the packets
   that are sent by the server */

byte ba2 = new byte [100];
DatagramPacket dp2 = new DatagramPacket (ba2, 100);
ds.receive(dp2);
String s2 = new String (ba2);
System.out.println ("data received from the server is " + s2);
} //main

} // class.
```

```
import java.net.*;
class DatagramServer
{
    public static void main (String args[])
    {
        DatagramSocket ds = new DatagramSocket (6666);
        byte ba1[] = new byte [100];
        DatagramPacket dp1 = new DatagramPacket (ba1, 100);
        ds.receive(dp1);
        String data = new String (ba1);
        System.out.println ("Data received from the client is " + data);
        data = "Hai from Server" + data;
        byte ba2[] = data.getBytes ();
        DatagramPacket dp2 = new DatagramPacket (ba2, ba2.length,
                                                dp1.getAddress (), dp1.getPort ());
        ds.send(dp2);
    }
}
```

DatagramServer.java

Explanation of the above program:

- In the client program, we create object of the DatagramSocket class by mentioning the port no. on which it is running.
- Represent the data in the form of bytes by calling the getBytes() method.
- Create object of the InetAddress class by passing the system address ~~port number~~ of the server program as an argument to its constructor. This object represents the IP address of the server machine.
- Now represent the data (bytes) in the form of packets by creating object of DatagramPacket class. Remember that as mentioned earlier, the constructor of this class takes four arguments - byte array (i.e. the data in the form of bytes), array length, IP address of server machine and port no. of the server program.
- call the send() method on the object of the DatagramSocket class by passing the object of the DatagramPacket class (created in the above step) as an argument to this method.
- After sending the packets to the server, it will process them and sends the data back in the form of packets.
- Declare a byte array of some size.
- create another object of the DatagramPacket class by passing the byte array object and its size as the arguments to the constructor.

NOTE: There is a constructor in the DatagramPacket class which accepts only two arguments - object of ~~an array~~ byte array and its size.

- call the `receive()` method on the object of the `DatagramSocket` class created earlier by passing the object of the `DatagramPacket` class (just created in the above step) as an argument to this method.
- The functionality of this method is to receive the packets one-by-one and retrieve the data (bytes) from them and store the data (bytes) in the array.
- convert these bytes into strings.
- In the server program, the above mentioned process only takes place, but in reverse order i.e. first the packets are received, converted into string format, processed and again converted into bytes and then represented in the form of packets and then sent back to the client.

Question: How can the server know, from which client it has received the data and how can it send back the response to that particular client only?

- we know that, to receive the data, we call the `receive()` method on the object of the `DatagramSocket` class created in the server program.

NOTE: Here, to the constructor of the `DatagramSocket` class we pass the port no. of the server program on which it is running.

- To the `receive()` method, we pass the object of the `DatagramPacket` class (which is created to accommodate the received packets) as an argument

- This object of the DatagramPacket class holds the information of the client (which it gets from the packet received).

NOTE: This indicates that, the packets, along with holding the metadata and address & port no. of the server machine and server program, also holds the details of the client.

- Now, on the object of this DatagramPacket class, if we call getAddress() and getPort() methods, then these two methods return the IP address of the client and port no. on which the client program is running.
- To send the data back to the client we need to call the send() method on the object of the DatagramSocket class. The argument to this method is object of the DatagramPacket class.
- We know that the DatagramPacket class has a constructor which takes four arguments — byte array object, length of the array, IP address of the destination (in this case, the destination is the client) and the port no. The IP address and port no. of the client program are obtained by calling the methods — getAddress() & getPort().
- This is how, the server recognizes the client after receiving the data from it, processes the data and sends the processed data back to the client.

10/9/2006

10/9/2006

- whenever we do not know the port number of the server program, then, for the client, to get connected to the server, we make use of the URL (Universal Resource Locator) of the server program.
- we have two classes - URL and URLConnection in the java.net package.
- URL class is used to represent the url in the form of an object.
- openConnection is a non-static method of URL class which returns object of the URLConnection class.
- now by calling the functionalities present in the URLConnection class, on the object obtained in the above step, we can exactly establish the connection.

NOTE: Establishing the connection, using the URL of the server program comes under connection-oriented protocol programming.

- So, here also we need to use the streams in order to send or receive data.
- Before establishing the connection we need to set some properties for the object of the URLConnection class i.e. we need to mention whether we want to send the data or receive the data using that object.

The rest of the programming part is similar to the one which we use in the connection-oriented protocol programming.

The following is a simple program in which the client gets connected to the server using the URL of the ServerProgram.

```
import java.net.*;
import java.io.*;

class UrlDemo
{
    public static void main (String args[])
    {
        URL u = new URL ("http://www.yahoo.com");
        URLConnection uc = u.openConnection();
        uc.setDoOutput (true); // setting the properties
        uc.setDoInput (true);
        uc.connect(); // Now the connection will be established.

        InputStream is = uc.getInputStream();
        BufferedReader br = new BufferedReader (new InputStreamReader (is));
        String data = br.readLine();
        s.o.p (data);
    }
}
```

This is how we get connected to the server program, when we do not know the port number on which the server program is running.

AWT

AWT is the acronym of Abstract Window Toolkit

Introduction:

GUI (Graphical User Interface) is a mediator (translator) between the end user and the program.

Form: It is a collection of GUIs designed for one specific purpose.

- console is a program, provided by the local os, which supports only text output.
- In order to display GUI output, we have another program (similar to the console) provided by the local os. This program is window
- All the java classes, which are used to create GUIs are present in the package `java.awt`.
- The theory behind the creation of GUIs is as follows.
 - The functionality of the constructor of the class, whose GUI is to be created, is to create that particular GUI.
 - Strictly speaking, this is not the functionality of the constructor of that class
 - In fact, the constructor has some statements which are nothing but, a call to a specific file in the local os. This particular file will be responsible in creating the corresponding GUI. In the foreground, it seems as if the GUI is created just because of the constructor.

- whatever language we use to create the GUI's, (for example: java, html, vb etc.) in all these cases, only the files present in the local os will be responsible for creating the corresponding GUI's.
- Because of this, the GUI's created in all these cases will look alike.
- Let us take java in particular.
When we want to create a GUI, we need to create object of that class which represents the corresponding GUI. Let us assume we need to create a GUI - a button. Then we need to write the following code, i.e.

```
Button b1 = new Button();
```

- So, to create the button GUI, JVM executes the constructor of the Button class. Inside the constructor we have statements which execute the files present in the local os to create the GUI.

** Definitely, That means, we are making JVM to execute the files present in the local operating system.

But remember that, JVM can execute only byte code, which ~~is~~ generated from the .class file and this .class file is resulted from a .java file.

- The files present in the local operating system need not be in java and definitely they will not be in java.
- They will be in some other native language.

Question: Then, how can JVM execute the files present in the local OS to create the GUIs?

Ans: JVM depends upon some classes known as Peer classes to execute the files present in the local OS.

** Thus, every awt component depends upon the files of the local OS to create GUIs, making JVM to execute those files which are non-java files. This increases the overhead on the JVM. This is the reason why we call awt classes (components) as heavy-weight Components.

- From the above discussion, it is clear that each OS will have its own files for creating the GUIs.
- Now, if we create a GUI using java on one OS, then the look and feel of this GUI will be completely different when we execute the same java program on another OS.

** Question: Java is not purely platform independent. Justify.

Ans: Because of the ~~awt~~ dependency of awt programs on the local OS in creating GUIs, the outputs of these programs vary from one OS to another OS. i.e. the

output of the AWT programs are platform dependent. This makes Java to be platform-dependent. This is the reason, why we say that Java is not purely platform-independent.

Disadvantages of AWT:

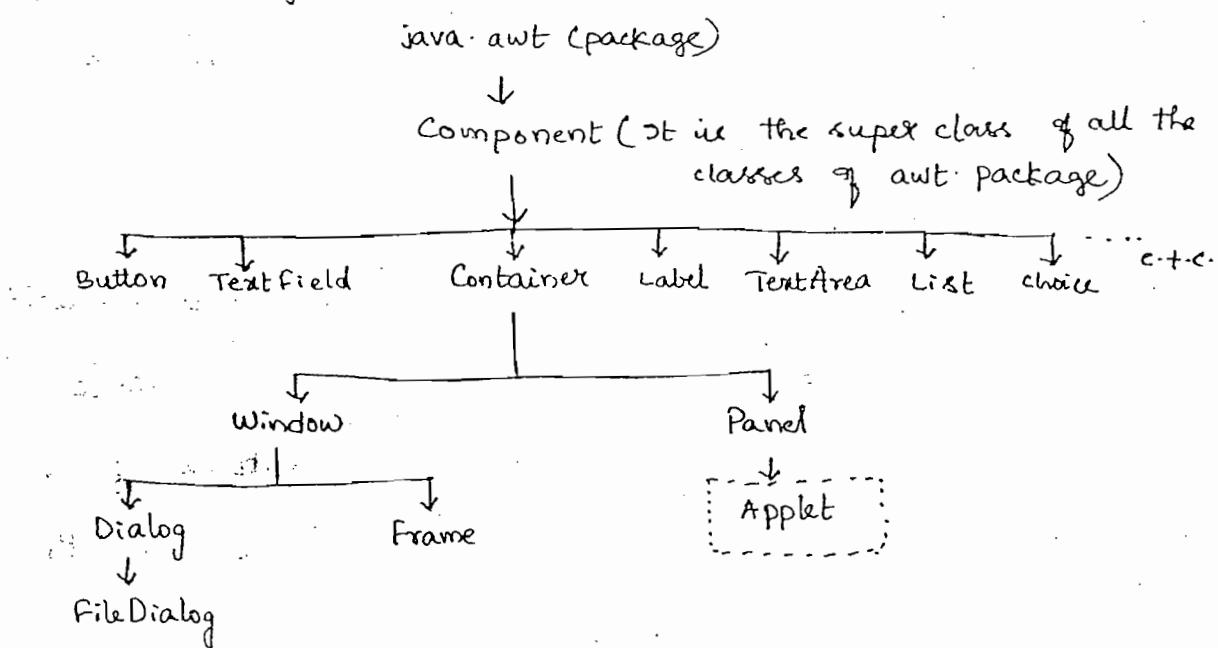
- (i) The AWT programs increase the overhead on the JVM.
- (ii) They make Java to be platform dependent.

The above disadvantages are overcome in Swing.

- Swing is an extension of AWT. The swing components do not depend upon the files of the local OS in creating the GUIs. And thereby, they decrease the overhead on the JVM.
- Thus, swing are known as Light-weight components.

12.09.2006

The following is the class hierarchy of java.awt package.



NOTE: Applet class does not belong to java.awt package.

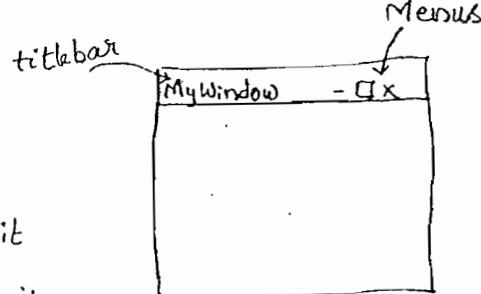
Applet class belongs to java.applet package.

- we know that window is used for displaying the GUI.
- A blind rule is that, when we want to add a component to another component then the later component should have the properties of a container
- whenever we want to view a GUI output, we add all the required components to the window and then they are displayed. That means, window has the properties of a container.
- from the class hierarchy, it is clear that window is the sub-class of container class. Hence it has all the properties of a container.
- if we want to add a button to another button, it is not possible because, button is not a sub-class of container and it does not have the properties of a container.
- same is the case with the other classes present in that level of hierarchy.
- similarly, we can not think of adding Textfield to a Label, TextArea to a List etc.

** Thus, any component which has the properties of container is eligible to accommodate other components in it.

- It is clear from the class hierarchy that Panel is a sub-class of container and Window is at the same level of hierarchy.
- Panel has the properties of container and any no. of components can be added to the Panel. To make them visible on the screen, we again need window, but observe that Panel does not have the properties of a window.
- Therefore, panel should be added to the window to make the components visible.
- Similarly, we can create a frame. Frame has the properties of window as well as container.
- As Button, TextArea e.t.c. have some properties, Window also has some properties.

- It has a title bar
- It has some system Menus
- We should be able to drag it
- We should be able to resize it
- e.t.c.



- Some of the best examples of the window application are Editplus Editor, Notepad, I.E. browser e.t.c.

NOTE: As console cannot display GUIs, Window cannot display text directly.

- In order to display text on the window, we need to represent the text in the form of a GUI component and then add it to the window. For this, we make use of the Label class.
- ** we always prefer to create a window through Frame because Window alone has no feature of boundaries. The facility of having boundaries for a window is possible with Frame. This is the reason why we create Window through Frame.
- Using the functions of the Frame class we can set the dimensions (size) for the window.

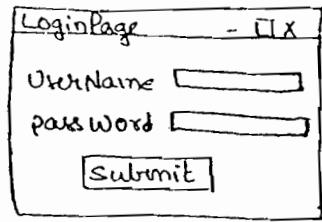
Ex : f.setSize (400, 400);

NOTE : By default, window would always be created in the invisible mode. To make the window visible on the VDU, we should explicitly set the visibility property of the window to the boolean value 'true'.

Ex : f.setVisible (true);

- A constructor defined in the Frame class takes a string class object as an argument. Thus, whatever the text we want to see in the title bar of the window, we need to pass it as an argument (in the form of a string class object) to the constructor of the Frame class.

Let us develop a simple program which displays a GUI as shown aside.



Hint: This GUI consists of six components

- two labels + two textfields + one button + one frame (window)

```
import java.awt.*;
```

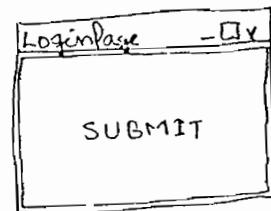
```
class Form1
```

Form1.java

```
{ public static void main (String args[])
{
    Frame f1 = new Frame (">Login Page");
    Label l1 = new Label ("UserName");
    Label l2 = new Label ("password");
    TextField tf1 = new TextField ();
    TextField tf2 = new TextField ();
    Button b1 = new Button ("submit");

    f1.add (l1);
    f1.add (l2);
    f1.add (tf1);
    f1.add (tf2);
    f1.add (b1);
    f1.setSize (437, 437);
    f1.setVisible (true);
}
```

O/P



- when we execute the above program, we can see only the button on the window. This is because, all the components have been added one upon another. The last component added is the button and hence we see only the button.
- This resulted because we have not mentioned, to which part of the window, the components have to be added. Hence they have occupied the entire window.
- so, to make all the components visible we have to align them properly on the window.
- we have some standard procedures using which we can align the components in the frame (container)
- The standard procedures which exactly specify the way in which we align align the components on the container are known as Layout Managers
- we have five layout Managers in the awt package, where each Layout Manager is the name of a class.
 - (i) Border Layout
 - (ii) FlowLayout
 - (iii) GridLayout
 - (iv) GridBagLayout
 - (v) CardLayout
- Therefore, we should mention the layout according to which the components are to be aligned on the container.

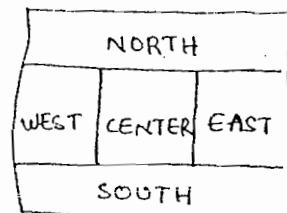
(P.T.O)

- we can mention the layout (specification or procedure) by using the `setLayout()` method of the Container class. This method is a non-static method.
- The `setLayout()` method takes the object of any of the above five layouts as an argument, according to which the components would be aligned (arranged) on the container.

13.09.2006

Border Layout:

- According to the specifications of the BorderLayout, the container area will be divided into five parts as shown in the fig.
 - The BorderLayout class has some static constants to refer to the areas divided on the container according to border layout
- They are NORTH, EAST, WEST, SOUTH, CENTER
we refer to these areas as BorderLayout.SOUTH e.t.c.
- The default layout associated with the Frame class object is BorderLayout. The default location associated with the BorderLayout is CENTER
 - Therefore, when we call `add()` method on the frame class object (i.e. `f.add(xxx)`), it takes the layout to be BorderLayout and adds the component to the centre of it.



- An important property of the BorderLayout is, when a component is added to the CENTER, then the component not only occupies the CENTER but also stretches to the other four areas, if there are no components in those areas.
- for example, if we add a button to the center, then it occupies the entire frame.
if we add two buttons - one to the NORTH and one to the center, then the button which is added to the CENTER also stretches to the EAST, WEST and SOUTH areas.

Flow Layout:

- According to this layout, the container will be divided into rows and columns depending upon the components and their size.
- when we start adding components to the frame according to flowLayout, then, the components would be added row-wise i.e. first row → column-1, column-2, e.t.c. if the row is filled, then the next component would be added to first column of second row and so on.

- NOTE: when we specify the layout as Flow layout, then, initially it assumes the entire space as 1row & 1column.
- when we add a component it would be added in the centre at the top (because initially it is 1row & 1column)

contd... (P.T.O)

- Now, when we add another component, control checks whether this component can be accommodated to the right of the existing component.
- If yes, it will be added, thus creating a second column.
- If no, the component would be added in the next row.
- Hence, size of the column completely depends upon the size of the component being added.
- Therefore, it is obvious that, when we add components to the container (frame) according to Flow layout, then the columns will be dynamically generated.

NOTE: The no. of rows and columns may or may not be equal.
i.e. the no. of columns may vary from one row to another row, depending upon the size of the components being added.

NOTE: When we resize the frame (on which the components are arranged according to flow layout) then the columns are going to get adjusted accordingly and even the components are going to get adjusted accordingly.

Ex:

| Before Resizing | After Resizing |
|--|--|
| - X Username <input type="text"/> Password <input type="password"/> Submit <input type="button" value="Submit"/> sports <input type="checkbox"/> music <input type="checkbox"/> <input type="button" value="TextField"/> | - X Username <input type="text"/> password <input type="password"/> Submit <input type="button" value="Submit"/> sports <input type="checkbox"/> music <input type="checkbox"/> <input type="text" value="TextField"/> |

// modifying the program Form1.java

```
import java.awt.*;
class Form2
{
    Frame f1 = new Frame ("LoginPage");
    Label l1 = new Label ("UserName");
    Label l2 = new Label ("Password");
    TextField tf1 = new TextField();
    TextField tf2 = new TextField();
    Button b1 = new Button ("SUBMIT");
    Form2()
    {
        f1.setLayout (new FlowLayout ());
        f1.add (l1); f1.add (l2);
        f1.add (tf1); f1.add (tf2);
        f1.add (b1);
        f1.setSize (437, 440);
        f1.setVisible (true);
    }
    public static void main (String args[])
    {
        Form2 f2 = new Form2 ();
    }
}
```

(P.T.O)

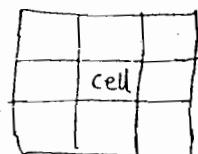
NOTE

- In the GUI programs which we have written till now, we have been creating objects of the TextField class wherever necessary. Instead whether it is for a Username label or a password label, the TextField is same. So, instead of creating another textField, can't we add the textField (which was created once) again and again where necessary?

NOTE: we can not reuse the GUI components once created in a container (frame) i.e. for example, if we have a textField already and if we need one more, then we can't add the one created earlier. Instead, we should create another textField and add it.

GridLayout :

- According to the specifications of GridLayout, the container will be divided into specific no. of rows and columns of equal size.
- each division is called a cell or a grid.
- In the GridLayout, only one component can be added to a grid, and the component occupies the entire grid.
- The components will be added to the grids in a serial fashion and we can not skip the grids in between. i.e. a component will not be added to grid-3, unless



grid-2 is filled provided grid-1 is already occupied.

- it is obvious that the constructor of Grid Layout class takes two integer arguments - row & column.

Ex : f1.setLayout(new GridLayout(4,4));

NOTE : when we use the GridLayout, it is compulsory that all the grids should be occupied with the components. No grid should be left free.

Panel : Panel is a component which has the properties of container. It does not have the properties of a window. So, panel has to be added to the window (frame) to make it visible.

- Since panel has the properties of a container, we can add any no. of components to the panel and now the panel will be treated as a single component.

NOTE : If we want to have more than one component in a grid, then we make use of the panel i.e. add all the required components to the panel and add this panel (which is a single component) to the grid.

- If we want to have an empty grid, then add an empty panel (without any components on the panel) to the grid. This makes the grid look empty.

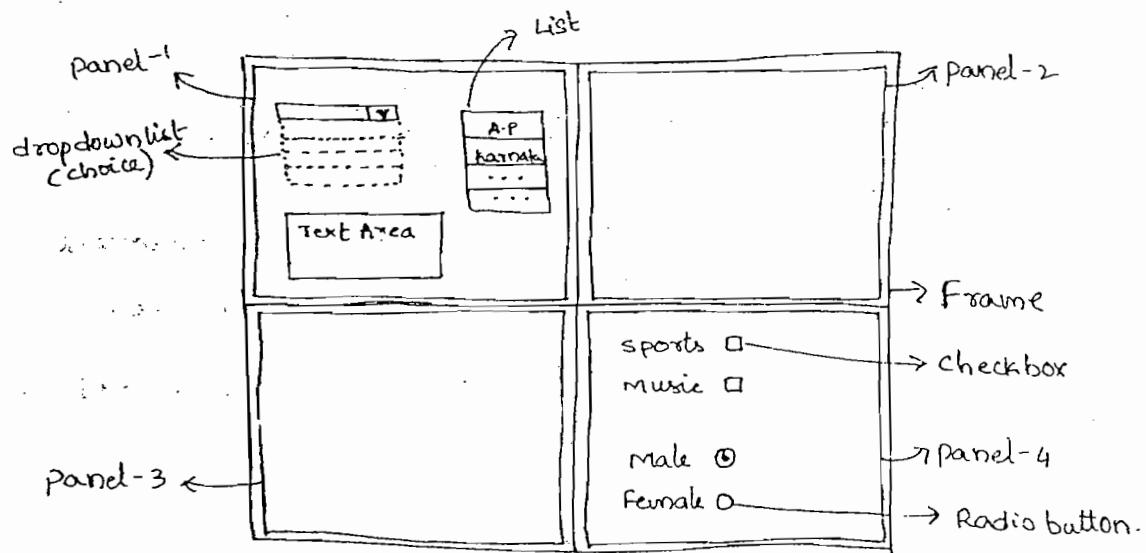
- we do not have any separate class to create radio buttons. we create radio buttons using checkbox class
- After creating the checkboxes, if we group them, they will act as radio buttons, else they will act as checkboxes.

14.09.2006

Question: Develop a GUI which meets the following requirements

- It should have four panels.
- panel-1 should have a dropdown list and an ordinary list and also a text-area.
- panel-2 and panel-3 are empty.
- panel-4 should have two checkboxes and two radio buttons.

Ans: The GUI would be something as shown below-



Hint: Here, we have altogether 12 components as listed below:

Frame - 1 choice - 1 TextArea - 1

panels - 4 List - 1 Checkboxes - 2

Radio buttons - 2

```
import java.awt.*;
class GridDemo1
{
    Frame f1 = new Frame (" Grid - Demo ");
    Panel P1 = new Panel ();
    Panel P2 = new Panel ();
    Panel P3 = new Panel ();
    Panel P4 = new Panel ();
    Choice ch = new Choice ();
    List lst = new List ();
    TextArea ta = new TextArea (9, 40);
    Checkbox cb1 = new Checkbox (" sports ");
    Checkbox cb2 = new Checkbox (" Music ");
    Checkbox ct1, ct2;
}
```

GridDemo1

```
{
    f1.setLayout (new GridLayout (2,2));
    ch.add (" Andhra Pradesh ");
    ch.add (" Karnataka ");
    ch.add (" Tamilnadu ");
    ch.add (" Kerala ");
    lst.add (" Hyderabad ");
    lst.add (" Bangalore ");
    lst.add (" Chennai ");
    lst.add (" Trivendrum ");
    P1.add (ch);
    P1.add (lst);
    P1.add (ta);
```

contd... (P.T.O)

```
p4.setLayout (new GridLayout (4,1));  
p4.add (cb1);  
p4.add (cb2);  
checkboxGroup cbg = new checkboxGroup ();  
r1 = new checkbox ("Male", true, cbg);  
r2 = new checkbox ("Female", false, cbg);  
p4.add (r1);  
p4.add (r2);  
f1.add (p1);  
f1.add (p2);  
f1.add (p3);  
f1.add (p4);  
f1.setSize (437, 437);  
f1.setVisible (true);  
}  
public static void main (String args [])  
{  
    GridDemo1 gd = new GridDemo1 ();  
}  
}
```

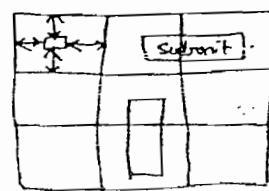
NOTE: The constructor of the checkbox class takes three arguments - string object (i.e. name), boolean value (to keep select or deselect the given radio button) , the checkboxGroup class object.

GridBagLayout :

The GridBagLayout is similar to the GridLayout, but with some additional features.

- in the grid bag layout, we have the facility of adding the components to a specific part of the grid (by mentioning the co-ordinates on all the four sides).
- we can stretch the components between the grids, either horizontally or vertically.

NOTE: Even though, GridBagLayout is advantageous, programmers generally do not prefer it because it increases the complexity of the program.

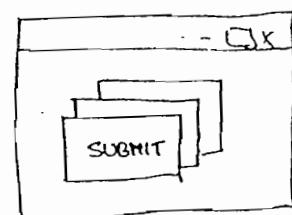


- we have a layout by name BoxLayout in the Swings package. we use this layout in order to achieve all that we do using GridBagLayout and with less complexity.

Card Layout :

The card layout specifications help us to add the components to a frame w.r.t. z-axis i.e. one component behind the other as shown in the figure.

- Card Layout is closely related to event-handling.



NOTE : Actually `setLayout()` method is defined to accept, an object of an interface, as an argument. The interface is Layout Manager.

All the layout classes implements the Layout Manager interface. Thus, object of any class, which implements Layout Manager interface, can be passed as an argument to the `setLayout()` method.

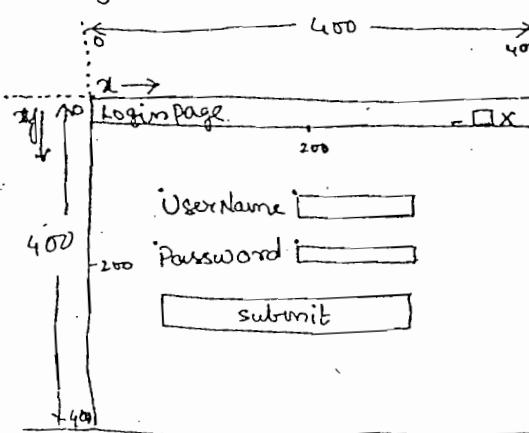
- Obviously, "null" can also be passed as an argument to the `setLayout` method.

15.09.2006

Null Layout : Strictly speaking null layout is not a layout. When we pass "null" as an argument to the `setLayout()` method, it nullifies all the standard layouts i.e. the default layouts will not be set for the corresponding components.

Now, we have the flexibility to add the components according to our specifications.

We can add the components to any part of the frame we like consider the following:



we need to create a login page as shown in the above figure.
The components should be placed, as they are shown in the fig.

First of all we need to calculate the bound properties of the components, as shown below

Username (Label l1)

x = 110 , y = 125 , width = 80 , height = 30

Textfield (tf1)

x = 200 , y = 125 , width = 110 , height = 30

Password (Label l2)

x = 110 y = 170 width = 80 height = 30

Textfield (tf2)

x = 200 y = 170 width = 110 height = 30

Button (b1)

x = 110 y = 220 width = 200 height = 30

we have a function setBounds() in the Component class.

The setBounds() method takes the bound properties as the arguments and aligns the components on the frame according to the specified bound-property values.

Following is the program to display the above login page.

```
import java.awt.*;
class LoginDemo
{
    Frame f1 = new Frame ("Login Page");
    Label l1 = new Label ("Username");
    Label l2 = new Label ("Password");
```

```
TextField tf1 = new TextField();
TextField tf2 = new TextField();
Button b1 = new Button("SUBMIT");
LoginDemo()
{
    f1.setLayout(null);
    l1.setBounds(110, 125, 80, 30);
    l2.setBounds(110, 170, 80, 30);
    tf1.setBounds(200, 125, 110, 30);
    tf2.setBounds(200, 170, 110, 30);
    b1.setBounds(110, 220, 200, 30);
    f1.add(l1);
    f1.add(tf1);
    f1.add(l2);
    f1.add(tf2);
    f1.add(b1);
    f1.setSize(400, 400);
    f1.setVisible(true);
}
public static void main (String args[])
{
    LoginDemo ld = new LoginDemo();
}
```

NOTE: Till now, we have been creating ~~an object~~ of the Frame class. But according to existing ~~abundance~~, the user-defined class extends Frame class as shown in the below example. Not only with Frame, but also the same is the case with all other components.

```
import java.awt.*;
class FormDemo extends Frame
{
    Label l1 = new Label ("username");
    Label l2 = new Label ("password");
    TextField tf1 = new TextField ();
    TextField tf2 = new TextField ();
    Button b1 = new Button ("SUBMIT");
    FormDemo()
    {
        setLayout (new FlowLayout ());
        add (l1);
        add (tf1);
        add (l2);
        add (tf2);
        add (b1);
        setSize (437, 400);
        setVisible (true);
    }
    public static void main (String args[])
    {
        FormDemo f = new FormDemo ();
    }
}
```

Event Handling

Event: Event is an object of a predefined class created by JVM to represent the corresponding action performed on the component.

- when an action is performed on a component (for example clicking of a button), then in order to represent that action, JVM creates an object of the ActionEvent class.
- This is the reason, why, events are always created in relation with actions.
- The list of all the event classes is present in the java.awt.event package.
- Registration methods: The functions which recognize the events generated in the components are known as registration methods.
- Event handling: catching the event class objects, which are generated because of an action performed on a component, and assigning them to the corresponding references is known as event handling. (i.e. avoiding the event class objects to reach JVM).
∴ we handle the events using the registration methods.

Hint: The event handling concept is very much similar to the exception handling concept.

16.09.2006

Listeners: Listeners are interfaces which define some function definitions. These function definitions bodies which of these function definitions will be executed by the registration methods whenever the registration methods identify event class objects being generated in the components because of actions performed on the components.

- The signature of the registration methods would be something like:

add <---> Listener()

Ex: addActionListener()

addTextListener(), addItemListener() etc..

NOTE: Every registration method is defined to accept object of a Listener interface (i.e. object of a class which implements the Listener interface).

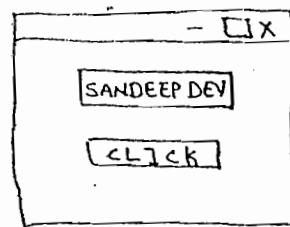
Ex: addActionListener(ai) . here ai, is an object of a class which implements ActionListener interface.

- addActionListener is a non-static method defined in the Button class.

NOTE: If there is a specific functionality to get executed because of an action being performed on a component, then we are supposed to handle the events in the corresponding component.

Question: Develop a GUI as shown in the figure.

Requirement: whenever you click on the button CLICK, then your name should appear in the textfield.



```
import java.awt.*;
class EventDemo1
{
    Frame f1 = new Frame();
    Button b1 = new Button("click");
    TextField tf1 = new TextField();
    EventDemo1()
    {
        f1.setLayout(new FlowLayout());
        f1.add(b1);
        f1.add(tf1);
        ActionListener ai = new A(tf1);
        b1.addActionListener(ai);
        f1.setSize(437, 440);
        f1.setVisible(true);
    }
    public static void main(String args[])
    {
        EventDemo1 ed = new EventDemo1();
    }
}
```

```

import java.awt.*;
import java.awt.event.*;
class A implements ActionListener
{
    TextField tf1;
    A(TextField tf1)
    {
        this.tf1 = tf1
    }
    public void actionPerformed(ActionEvent e)
    {
        tf1.setText("sandeepr Dev");
    }
}

```

Explanation:

The functionality of the registration method (in this example it is add ActionListener()) would be as follows:

- It waits for an event to occur in the component (in our example the component is the button b1)
- If an event occurs (i.e. if the button is clicked), then it catches that event, assigns the event to the corresponding reference. Now this reference (in fact it is the event object) will be sent as an argument to the specific function (in this case it is the actionPerformed()) and that method will be invoked. whatever the functionality

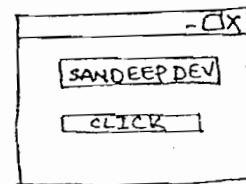
or logic we need to get executed when the action is performed on the component, should be written in this method.

- In the above example, our requirement was to print the name in the textfield when the button is clicked.
- The logic of printing the name in the textfield is written in the actionPerformed() method and this method is incorporated in a user defined class, class A where class A implements the ActionListener interface.

18-09-2006

In the above example, we have used two classes in accomplishing the task. we can achieve it using a single class as shown below.

```
import java.awt.*;
import java.awt.event.*;
class EventDemo2 implements ActionListener
{
    Frame f1 = new Frame();
    Button b1 = new Button("CLICK");
    Textfield tf1 = new Textfield();
    EventDemo2()
    {
        f1.setLayout(new FlowLayout());
        f1.add(b1); f1.add(tf1);
        b1.addActionListener(this);
        f1.setSize(437, 440);
        f1.setVisible(true);
    }
}
```



```

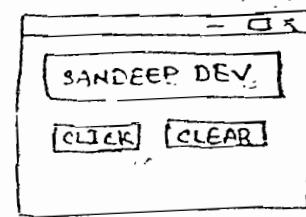
public void actionPerformed (ActionEvent e)
{
    tf1.setText ("Sandeep Dev");
}

public static void main (String args[])
{
    EventDemo2 ed = new EventDemo2 ();
}
}

```

Question: Display a GUI as shown

Requirement: whenever we click the button CLICK, your name should be displayed in the textfield. when CLEAR is clicked, the textfield should be cleared.



```

import java.awt.*;
import java.awt.event.*;
class EventDemo3 implements ActionListener
{
    Frame f1 = new Frame ();
    Button b1 = new Button ("CLICK");
    Button b2 = new Button ("CLEAR");
    TextField tf1 = new TextField ();
    EventDemo3 ()
    {
        f1.setLayout (new FlowLayout ());
        f1.add (b1);
        f1.add (b2);
        f1.add (tf1);
    }
}

```

contd... (P.T.O)

```

        b1.addActionListener(this);
        b2.addActionListener(this);
        f1.setSize(437, 440);
        f1.setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
        String s1 = e.getActionCommand();
        if(s1.equals("CLICK"))
            tf1.setText("Sanadeep Dev");
        if(s1.equals("CLEAR"))
            tf1.setText("");
    }

    public static void main(String args[])
    {
        EventDemo3 ed = new EventDemo3();
    }
}

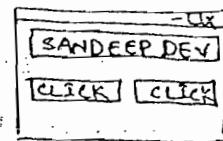
```

Explanation:

- In this example, we have two buttons - CLICK and CLEAR.
- An event object will be generated when either of the buttons is clicked. Based on this object, the actionPerformed() method will be executed.
- Now, it is our duty to identify, because of which button the event object has been generated and execute the corresponding functionality.

- There is a non-static function by name, `getActionCommand()` in the `ActionEvent` class. This method returns a `String` class object which is associated with the component that is responsible for generating the object of the `ActionEvent` class.
- we have used this method in our program to identify the button on which the action has been performed and generated the corresponding event class object.

Question: what is the case, if both the buttons have the same name as shown in the fig.



- The requirement is same as that in the above example.
- There is a method by name `getSource()` which returns an object of the component (in the form of an object of `Object` class) which is otherwise responsible in creating the object of `ActionEvent` class.
- The object returned by `getSource()` method will be in the form of object of `Object` class. we need to perform the necessary typecasting.
- The following program illustrates the use of `getSource()` method. The program generates a form shown in the above question.

contd... (P.T.O)

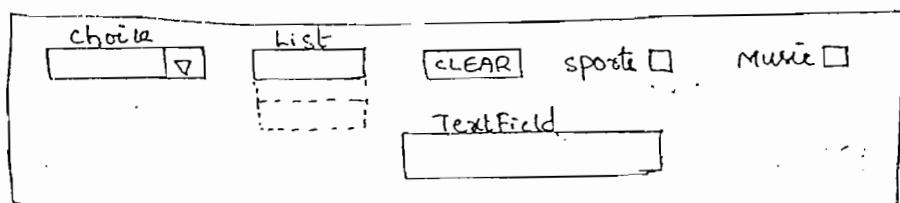
```
import java.awt.*;
import java.awt.event.*;
class EventDemo4 implements ActionListener
{
    Frame f1 = new Frame();
    Button b1 = new Button("CLICK");
    Button b2 = new Button ("CLICK");
    TextField tf1 = new TextField();
    EventDemo4()
    {
        f1.setLayout(new FlowLayout());
        f1.add(b1);
        f1.add(b2);
        f1.add(tf1);
        b1.addActionListener(this);
        b2.addActionListener(this);
        f1.setSize(437, 440);
        f1.setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        Object o = e.getSource();
        Button b = (Button)o;
        if (b==b1) tf1.setText("Sandeep Dev");
        if (b==b2) tf1.setText("");
    }
}
```

```

public static void main (String args [ ] )
{
    EventDemo4 cd = new EventDemo4 ();
}
}

```

Question: create a GUI as shown below.



Requirement: Initially, the list box should be empty. When an item is selected in the choice, it should be added to the list box. When we click the button CLEAR, the list box should once again become empty. When we select a checkbox, then the corresponding label should appear in the text field.

```

import java.awt.*;
import java.awt.event.*;

class EventDemos implements ActionListener, ItemListener
{
    Frame f1 = new Frame();
    Choice ch1 = new Choice();
    Button b1 = new Button ("CLEAR");
    Textfield tf1 = new Textfield ();
    List lst = new List();
    Checkbox sports = new Checkbox ("sports");
    Checkbox music = new Checkbox ("music");

```

contd... (P.T.O)

EventDemo5()

```
{ s1.setLayout (new FlowLayout());  
    ch1.add ("HYDERABAD");  
    ch1.add ("CHENNAI");  
    ch1.add ("BANGALORE");  
    fl.add (ch1);  
    fl.add (lst);  
    fl.add (b1); fl.add (f1);  
    fl.add (sports);  
    fl.add (music);  
    fl.add (tf1);  
    ch1.addItemListener (this);  
    b1.addActionListener (this);  
    sports.addItemListener (this);  
    music.addItemListener (this);  
    fl.setSize (437, 440);  
    fl.setVisible (true);  
}
```

19-09-2006

```
public void itemStateChanged (ItemEvent e)  
{  
    Object o = e.getSource();  
    try {  
        Checkbox cb = (Checkbox) o;  
        if (cb == sports)  
        {  
            boolean flag1 = sports.getState();  
        }  
    }  
}
```

```

    if (flag1 == true)
        tf1.setText ("sport");
    }

    if (cb == music)
    {
        boolean flag2 = music.getState ();
        if (flag2 == true)
            tf1.setText ("music");
    }

} //try

catch (Exception e)
{
    string s1 = ch1.getSelectedItem ();
    if (s1 != null)
        lst.add (s1);
}

}

public void actionPerformed (ActionEvent e)
{
    lst.removeAll ();
}

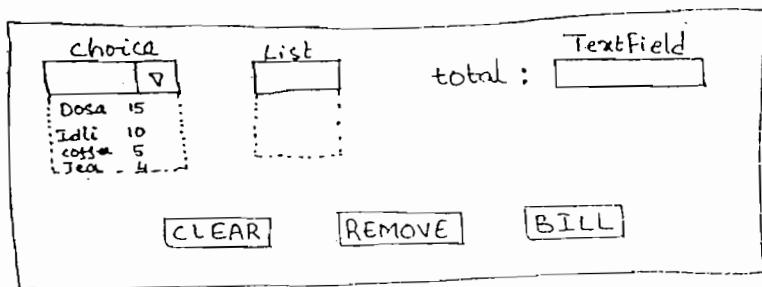
public static void main (String args[])
{
    EventDemo5 ed = new EventDemo5 ();
}

} //class

```

NOTE: we can use try - catch as if - else when necessary.
 In the above program we have done the same in the itemStateChanged () method.

Question: create a GUI as shown below.



Requirement:

Selected items in the choice should be added to the list.
Any unwanted item in the list should be removed. when BILL is clicked, the total amount of the items in the list should appear in the textField. when CLEAR is clicked both, the list and textField should be cleared.

- Till now we have created many windows (frames), but we were not able to close them using the system menu (i.e. by clicking the \times button on the top-right corner).

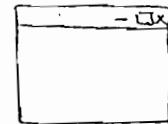
The reason is simple. clicking the \times button (system menu) is an event action on the window (frame) and till now we did not handle the event generated because of the action.

If we handle that event through a registration method, then it would be possible for us to meet our requirement.

- dispose() is a non-static function of the window class, using which we can destroy the window object.

- `System.exit()` is another function of the `System` class using which we can destroy the window object. But in this case, along with the window, the application will also be terminated.

Question: create a GUI as shown.



Requirement: whenever user clicks the button the window (frame) should be closed.

```
import java.awt.*;
import java.awt.event.*;

class WindowEventDemo1 implements WindowListener
{
    Frame f1 = new Frame();

    WindowEventDemo1()
    {
        f1.addWindowListener(this);
        f1.setSize(437, 440);
        f1.setVisible(true);
    }

    public void windowActivated(WindowEvent e) { }

    public void windowClosed(WindowEvent e) { }

    public void windowClosing(WindowEvent e)
    {
        f1.dispose();
        System.exit(0);
    }
}
```

contd... (P.T.O)

```

public void windowDeactivated (WindowEvent e) { }
public void windowDeiconified (WindowEvent e) { }
public void windowIconified (WindowEvent e) { }
public void windowOpened (WindowEvent e) { }
public static void main (String args[])
{
    WindowEventDemo1 wd = new WindowEventDemo1 ();
}
} // class.

```

Explanation:

In the above example, we are required to create a GUI and when we click the button, the window (frame) should be closed.

- We know that is a system menu present on the window (frame). whenever we click it, an event object will be generated. we need to handle this event.
- That means, we should find out the registration method using which we can handle the event object generated in the window (frame).
- we know that every registration method is defined to accept the reference of an interface as an argument.
- we should see that our class implements this interface and now we need to provide bodies for all the methods present in the interface.

- we have done the same in the above programs.
we have identified the registration method in the window class (i.e. addWindowListener()) and we have called it on the frame class object f.
- The addWindowListener() method takes the reference of windowListener interface.
- Inside the windowListener interface, many methods are declared for which we need to provide the bodies.
- Out of all those methods, we will use those methods with which we can perform the required functionality.
For the other methods, we just provide empty bodies.

Question: create your own text editor, with some properties, as shown below.

| MyEditor | | -File | |
|----------|--------|--------|--|
| File | color | Font | |
| save | Green | bold | |
| open | white | italic | |
| Exit | Orange | | |

NOTE: Menubars would never be added. They can only be set.
Menubars can be placed only at the top of the frame. They can't be placed anywhere else on the frame.

- we can set the Menubars on the frame using the setMenuBar() method.

(P.T.O)

- The `setMenuBar()` is defined to accept object of `MenuBar`.
- It is a non-static function present in the `Frame` class.
- We know that `Panel` is neither a sub-class nor a super class of `Frame`. Therefore `setMenuBar()` is not available to the `Panel` class.

This is the reason why we can set menus only to `Frames` but not `Panels`.

FileDialog: It is a component using which we can view all the contents (files) present in the harddisk.

```

import java.awt.*;
import java.awt.event.*;
class MyEditor implements
{
    Frame f1 = new Frame ("My Editor");
    TextArea ta1 = new TextArea (18, 40);
    MenuBar mb = new MenuBar ();
    Menu file = new Menu ("File");
    Menu color = new Menu ("Color");
    Menu font = new Menu ("Font");
    MenuItem save = new MenuItem ("Save");
    MenuItem open = new MenuItem ("Open");
    MenuItem exit = new MenuItem ("Exit");
    MenuItem green = new MenuItem ("Green");
    MenuItem blue = new MenuItem ("Blue");
}
  
```

```
28. MenuItem orange = new MenuItem ("Orange");  
29. MenuItem bold = new MenuItem ("Bold");  
30. MenuItem italic = new MenuItem ("Italic");
```

MyEditor ()

```
{  
    file.add (save);  
    file.add (open);  
    file.add (exit);  
    color.add (green);  
    color.add (blue);  
    color.add (orange);  
    font.add (bold);  
    font.add (italic);  
    mb.add (file);  
    mb.add (color);  
    mb.add (font);  
    fl.setMenuBar (mb);  
    fl.add (ta1);
```

20.9.06

```
save.addActionListener (this);  
open.addActionListener (this);  
exit.addActionListener (this);  
green.addActionListener (this);  
blue.addActionListener (this);  
orange.addActionListener (this);  
bold.addActionListener (this);  
italic.addActionListener (this);  
fl.setSize (437, 440);  
fl.setVisible (true);
```

```
} // constructor
```

contd... (P.T.O)

```
public void actionPerformed (ActionEvent e)
{
    String s1 = e.getActionCommand ();
    if (s1.equals ("exit"))
        System.exit (0);
    if (s1.equals ("green"))
        tai.setBackground (Color.GREEN);
    if (s1.equals ("blue"))
        tai.setBackground (Color.BLUE);
    if (s1.equals ("orange"))
        tai.setBackground (Color.ORANGE);
    if (s1.equals ("Bold"))
    {
        Font f01 = new Font ("Serif", Font.BOLD, 18);
        tai.setFont (f01);
    }
    if (s1.equals ("Italic"))
    {
        Font f02 = new Font ("Serif", Font.ITALIC, 18);
        tai.setFont (f02);
    }
    if (s1.equals ("save"))
    {
        FileDialog fdl = new FileDialog (f1, "save", FileDialog.SAVE);
        fdl.show ();
        String dir = fdl.getDirectory ();
    }
}
```

worlde... (T.O)

```
        string fname = fd1.getFile();
        string path = dir + "/" + fname;
        saveFile(path);
    }

    if (s1.equals("open"))
    {
        fileDialog fd2 = new fileDialog (f1, "open", FileDialog.LOAD);
        fd2.show();
        string dir1 = fd2.getDirectory();
        string fn = fd2.getFile();
        string path = dir1 + "/" + fn;
        openFile(path);
    }
}
```

//actionPerformed.

```
public void savefile (String path)
{
    string data = ta1.getText();
    FileOutputStream fos = new FileOutputStream (path);
    byte ba[] = data.getBytes();
    fos.write (ba);
    fos.close();
}
```

contd... (P.T.O)

```
public void openFile (String path)
{
    FileInputStream fis = new FileInputStream (path);
    int size = fis.available ();
    byte ba2[] = new byte [size];
    fis.read (ba2);
    String data = new String (ba2);
    t1.setText (data);
    fis.close ();
}

public static void main (String args[])
{
    MyEditor med = new MyEditor ();
}

}//class.
```

Explanation :

In the above program, we have handled the events generated by all the eight MenuItem's which we have considered in the GUI

Except for the MenuItem's - open and save , handling the events in the rest of the MenuItem's is same as the process which we followed in the previous examples.

Even for 'open' & 'save' also the process is same but with some extra functionality which needs additional logic to be written.

- Dialog is a window.
- whenever we need to display a window on another window with a relation between those two windows, then we make use of the Dialog.
- The relation between the windows as mentioned above is nothing but, as long as the child window is alive, the parent window can not be closed.
- Thus, if a Dialog has to be created, then definitely we need to pass the object of its parent window as an argument to the constructor of the Dialog class.
- As already mentioned, FileDialog is one of the Dialog.
- A FileDialog can be created in two modes as given below:
 - Load mode - using which we can display the file.
 - Save mode - using which we can save the file to a location.
- Thus, the constructor of FileDialog is defined to accept three arguments.

`FileDialog (f1, "save", FileDialog::SAVE);`

`f1` → The parent window object on which the FileDialog has to be displayed

`"save"` → The string which is to be displayed on the titlebar.

`FileDialog::SAVE`
`(or)`
`FileDialog::LOAD`

→ the mode in which the FileDialog has to be opened.

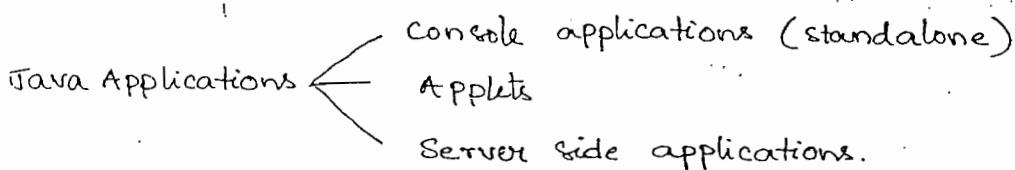
contd... (P.T.O)

- we use the setVisible(true) function to make the window visible on the screen.
- In a similar manner, when we "click" 'save' or 'open' in the MenuItems, then the FileDialog should pop-up on the existing frame (window). For this purpose we call the show() method on the object of the FileDialog.
- The functions savefile() and openfile() are user defined functions, written to perform the corresponding functionalities which their names suggest.
- This is the process of creating a simple text editor. We can make more enhancements to this program to make it an efficient program editor.

21-09-2006

Applets

- Applet: An applet is a java program which can be executed on a java enabled browser.
- Java applications can be classified as shown below



- Life cycle: It is the procedure followed by JVM to execute the programs.

NOTE: Life cycle differs from one category to another category of applications.

For example:

In console applications, the life cycle starts in the `main()` method and also ends in the `main()` method.

In applet, the life cycle starts in the `init()` method and ends in the `destroy` method.

On this way, it differs from one kind of application to another.

- We know that to execute an applet, a java enabled browser is needed. Also we know that only JVM can deal with java programs (i.e. .class files). Then

Then how can a browser execute an applet which is in turn a java program?

contd..(P.T.O)

- This would be possible because, every Java enabled browser has an in-built JVM in it which would deal with the applets (class files).
- Browser is a window and it understands only HTML. How can browser understand the Java programs (applets)? Since browser understands only HTML, using HTML we inform the browser about the applet.
- There is a tag by name <applet>. The <applet> tag has three attributes, as given below
 - code - Mentions the name of the .java file which is to be executed as an applet.
 - height, width - These two attributes describe the dimensions of the applet.
- whenever the browser encounters the <applet> tag, then it hands over the file name present in the 'code' attribute to the in-built JVM. Now, this JVM starts executing the program.
- Using applets we can display any GUI and the same GUI can be displayed even by using HTML in a more simpler way.

Question: why should we use applets, when we have an easy alternative (HTML)?

Ans: whenever it is required to hide the source code,

then we prefer applets to display the GUI's rather than using html to display the same GUI.

In other words, the disadvantage with HTML is that, in the browser, if we click the 'view source' option then the entire html code will be visible to the user which would be sometimes very dangerous.

If we use applet, then this possibility would not be there.

Life cycle of an applet

- As soon as browser encounters the `<applet>` tag in the html file, the browser hands over the .class file to its JVM mentioned in the 'code' attribute.
- JVM creates an object of that class and this object would be assigned to a reference of the Applet class.
- Now, JVM calls `init()` method and after that it calls `start()` method on the reference of the Applet class.
- If control has to come out of the applet program, then JVM calls `stop()` method and then the control is relinquished.
- Suppose if control comes back to the html page on which the applet program was running, then once again the html code will be executed and in this process, `<applet>` tag will be encountered and once again the .class file would be handed over to the JVM.

- Now, JVM will not create another object, since the object has already been created earlier.
- Since the object is not created (for the second time) `init()` method will not be executed. Instead, JVM executes the `start()` method.

NOTE: `init()` method would be called only once in the lifecycle of an applet

- If the browser is closed, then before deleting the object of the .class file, JVM calls the `destroy()` method on the reference of the Applet class
- The four functions (`init()`, `start()`, `stop()`, `destroy()`) through which JVM executes the applet program are known as Life Cycle Methods.
- These life cycle methods are defined in Applet class of `java.applet` package.
- `java.applet` package is one of the smallest package in the entire JDK. It has only one class (`Applet`) and some interfaces.
- In the present version, `destroy()` is a deprecated method.

NOTE: `paint()` is not a life cycle method.

Moreover `paint()` method is present in the Component class

Question: Develop a simple applet program which displays a button "Hello" and a text field. When you click the button, your name should be displayed in the textfield.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class HelloApplet1 extends Applet implements ActionListener
{
    Button bl;
    Textfield tf1;

    public void init()
    {
        bl = new Button("Hello");
        tf1 = new Textfield();
    }

    public void start()
    {
        add(bl);
        add(tf1);
        bl.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        tf1.setText("SANDEEP DEV");
    }
}
```

contd ... (P-I-O)

```
<html>  
    <body>  
        <H3> This is my first applet program </H3> <BR>  
        <applet code = "HelloApplet1" width = "437" height = "440">  
        </applet>  
    </body>  
</html>
```

Hello1.html

Question: can we define a constructor in an applet class?

Ans: Yes, we can define a constructor in an applet class
provided, the constructor should be a no-argument constructor.

Image class:

- The Image class is used to represent the images.
 - we can not directly create object of the Image class because it is an abstract class.
- ∴ getImage() function (a non-static function of Applet class) creates ^{object} of the Image class, representing the contents of the gif file that we pass as an argument to the getimage() method.

NOTE: getimage() method takes two arguments,

(i) name of the gif file - a string class object

(ii) the path to the gif file in the form of a URL class object.

22.09.2006

To display the images on the browser,

- (i) create Image class object which represents the contents of the gif file
- (ii) display the Image object on the applet.

Hint : (i) getCodeBase() function which belongs to Applet class

returns object of the URL class. which represents the path/url of the current applet .class file.

(ii) getDocumentBase() function returns object of the URL class which represents the path/url of the current html file.

Ex: Image img = getImage(getCodeBase(), "tiger.gif");

- To display the image on the applet, we use a function by name drawImage(). It is a non-static function present in Graphics class.
- drawImage() method is defined to accept four arguments.
 - (i) Image class object representing the image (gif file)
 - (ii) X- coordinate value
 - (iii) Y- coordinate value
 - (iv) the object of the container on which the image has to be displayed.
- drawImage() method is a non-static method of Graphics class. So, to call drawImage() method we need object of Graphics class, but Graphics class is an abstract class.

Question: How can we get the object of Graphics class?

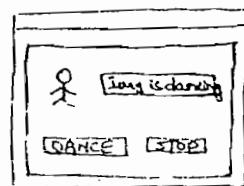
(P.T.O)

- `paint()` is a method present in the `Component` class and it is defined to accept object of `Graphics` class.

i.e. `paint(new Graphics() { ... })`;

- An anonymous inner class extends the `Graphics` class and this returns the object of the `Graphics` class.
- Now, we can call `drawImage()` method on the object of the `Graphics` class.

Question: create an applet GUI as shown



- Requirement: Just display the image, buttons and text field, on the applet. Do not handle any events.

```

import java.awt.*;
import java.applet.*;

public class ImageApplet1 extends Applet
{
    Button b1, b2;
    Textfield tf1;
    Image img;
    public void init()
    {
        b1 = new Button("DANCE");
        b2 = new Button("STOP");
        tf1 = new Textfield();
        img = getImage(getCodeBase(), "tiger.gif");
    }
}
  
```

ImageApplet1.java

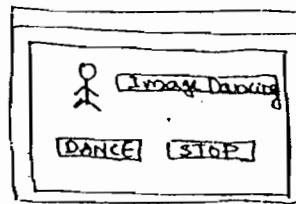
```
public void start ()  
{  
    add (b1);  
    add (b2);  
    add (tf1);  
}  
public void paint (Graphics g)  
{  
    g. drawImage (timg, 22, 40, this);  
}
```

```
<! -- ImageDemo.html --> ImageDemo.html  
<html>  
<body>  
    <h3> My First Image Application </h3> <br>  
    <applet code = "ImageApplet1" width = "437" height = "440">  
    </applet>  
</body>  
</html>
```

In the above program, we used an image by name tiger.gif. we can use any of the image files with .gif extension, to be displayed on the applet.

NOTE: Before executing the above example, we should see that, ImageApplet1.java, ImageDemo.html, tiger.gif - all lie in the same path.

Question: Develop a GUI as shown.



Requirement: when we click the button "DANCE", then the image should start dancing and in the textfield it should display "Image is dancing" on a green background.
when we click the button "STOP", then the image should stop dancing and in the textfield it should display "Image stopped dancing" on a red background.

Hints: - To meet the above requirement, first we need to have different postures of an image i.e. different gif files of the same image with slight differences between them.

- we have to make use of a non-static function repaint() of the Component class.
- The functionality of the repaint method is to call the paint() method of the current object on which we call the repaint() method.

NOTE: To execute the repaint() method, the paint() method has to be called atleast once.

25.09.2006

UTIL PACKAGE

→ The Collections Framework

— Collections Overview

- The java collections framework standardizes the way in which groups of objects are handled by our programs.
- Prior to Java 2, java provided ad hoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme.
- The collections framework was designed to meet several goals:
 - (i) The framework had to be high-performance
 - (ii) The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability
 - (iii) Extending / Adapting a collection class had to be easy.
- Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class.
- Algorithms provide a standard means of manipulating collections.
- Another item created by the collections framework is the Iterator interface. It gives us a general-purpose, standardized way of accessing the elements within a collection, one at a time.

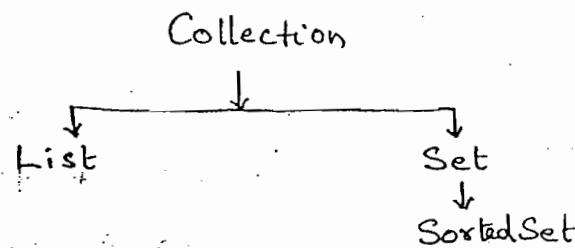
- Thus, an iterator provides a means of enumerating the contents of a collection. Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator.
- In addition to collections, the framework defines several map interfaces and classes.
- Maps store key-value pairs.

NOTE: Although maps are not "collections" in the proper use of the term, they are fully integrated with collections.

— java.util defines the following interfaces:

| | |
|---------------|--------------|
| Collection | Map |
| Comparator | Map.Entry |
| Enumeration | Observer |
| EventListener | RandomAccess |
| Iterator | Set |
| List | SortedMap |
| ListIterator | SortedSet |

The interfaces that underpin collections are listed below w.r.t to their hierarchy.



- The Collection Interface:

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have.

NOTE: Most of the methods throw an `UnsupportedOperationException` if a collection can not be modified.

A `ClassCastException` is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection.

The following are the methods defined in the Collection interface.

- boolean add (Object obj) : Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
- boolean addAll (Collection c) : Adds all the elements of c to the invoking collection. Returns true if the elements were added. Otherwise, returns false.
- void clear () : Removes all the elements from the invoking collection
- boolean contains (Object obj) : Returns true if obj is an element of the invoking collection. Otherwise returns false.
- boolean containsAll (Collection c) : Returns true if the invoking collection contains all elements of c. Otherwise returns false.
- boolean equals (Object obj) : Returns true if the invoking collection and obj are equal. Otherwise returns false.

Contd....(P.T.O)

- int hashCode(): Returns the hashCode for the invoking collection.
- boolean isEmpty(): Returns true if the invoking collection is empty. Otherwise, returns false.
- Iterator iterator(): Returns an iterator for the invoking collection.
- boolean remove(Object obj): Removes one instance of obj from the invoking collection and returns true if the element was removed, else returns false.
- boolean removeAll(Collection c): Removes all elements of c from the invoking collection. Returns true if the elements were removed. Otherwise, returns false.
- boolean retainAll(Collection c): Removes all elements from the invoking collection except those in c. Returns true, if the elements were removed. otherwise returns false.
- int size(): Returns the no. of. elements held in the invoking collection.
- Object[] toArray(): Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
- Object[] toArray(Object array[]): Returns an array containing only those collection elements whose type matches that of array. If the size of array equals the no. of matching elements, these are returned in array.
If the size is less than the ^{no. of} matching elements, a new array of the necessary size is allocated and returned. If the size of the array is greater than the no. of matching elements, the array element following the last collection element is set to null.

ion.

NOTE: On the above method, an ArrayStoreException is thrown if any collection element has a type that is not a subtype of array.

ction • The List Interface:

The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

NOTE: A list may contain duplicate elements.

In addition to the methods defined by Collection interface, the List interface defines some of its own methods, as shown below.

- void add (int index, Object obj): Inserts obj into the invoking list at the ^{value} index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
- boolean addAll (int index, Collection c): Inserts all elements of c into the invoking list at the location passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
- Object get (int index): Returns the object stored at the specified index within the invoking collection.

contd....(P.T.O)

- int indexOf (Object obj): Returns the index of the first instance (since list may contain duplicates) of obj in the invoking list. If obj is not element of the list, then -1 is returned.
- int lastIndexOf (Object obj): Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, then -1 is returned.
- ListIterator listIterator(): Returns an iterator to the start of the invoking list.
- ListIterator listIterator (int index): Returns an iterator to the invoking list that begins at the specified index.
- Object remove (int index): Removes the element present at the location specified by index from the invoking list and returns the deleted element. The resulting list (after deletion) is compacted i.e. the indexes of the subsequent elements are decremented by one.
- Object set (int index, Object obj): Assigns obj to the location specified by index within the invoking list.
- List subList (int start, int end): Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

- The ArrayList class:

- The ArrayList class extends AbstractList and implements the List interface.
- ArrayList supports dynamic arrays that can grow as needed.
- ArrayList is a variable-length array of object references i.e. an ArrayList can dynamically increase or decrease its size.
- ArrayLists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.
- The ArrayList class has three constructors as shown below
 - (i) ArrayList(): Builds an empty array list
 - (ii) ArrayList(Collection c): Builds an array list that is initialized with the elements of the collection c
 - (iii) ArrayList(int capacity): Builds an array list that has the specified initial capacity. The capacity grows automatically as more elements are added to the array list.
- The ArrayList class is not synchronized.
- To make the ArrayList class synchronized, we follow a different approach (discussed later).
- The following program demonstrates a simple use of the ArrayList. An array list is created and string objects are added to it. The list is then displayed. Some modifications are done on the list and again the list is displayed.

```
import java.util.*;
class ArrayList Demo
{
    public static void main (String args[])
    {
        ArrayList al = new ArrayList(); // creates an empty list
        System.out.println("Initial size is :" + al.size());
        al.add ("IND");
        al.add ("USA");
        al.add ("UK");
        al.add ("CHN");
        al.add ("SA");
        al.add ("SL");
        al.add ("KEN");
        System.out.println("Size after additions : " + al.size());
        System.out.println("Contents of list : " + al);
        al.add (2, "AUS");
        System.out.println("After addition, contents are : " + al);
        al.remove ("KEN");
        al.remove (3);
        System.out.println("Size of list after deletions : " + al.size());
        System.out.println("Contents of the list : " + al);
    }
}
```

The output of the above program is as shown below:

Initial size is : 0

Size after additions : 7

contents of list : IND USA UK CHN SA SL KEN

After addition, contents are : IND USA AUS UK CHN SA SL KEN

Size of list after deletions : 6

Contents of the list : IND USA AUS CHN SA SL

NOTE: Although the capacity of an ArrayList object increases automatically as objects are stored in it, we can increase the capacity of an ArrayList object manually by calling ensureCapacity() method.

Conversely, if we want to reduce the size of the array that underlies an ArrayList object so that it is precisely as large as the no. of items that it is currently holding, we call trimToSize() method.

The signatures of the above two functions are as shown below.

- void ensureCapacity (int cap)
- void trimToSize ()

- Sometimes, we may need to obtain an actual array that contains the contents of the list. we can achieve this by calling the toArray() method.

contd.... (p.T.O)

we may be required to do so in the following situations:

- To obtain faster processing times for certain operations.
 - To pass an array to a method that is not overloaded to accept a collection.
 - To integrate our newer, collection-based code with legacy code that does not understand collections.
- The following program illustrates the concept of converting an ArrayList into an array.

consider the ArrayList created in the previous example

```
import java.util.*;  
  
class ArrayListToArray  
{  
    public static void main (String args[]){  
          
        ArrayList al = new ArrayList();  
        al.add ("IND");  
          
        ;  
        ;  
        al.add ("KEN");  
          
        System.out.println ("contents of the list : " + al);  
          
        Object sa[] = al.toArray();  
          
        for (int i=0; i < sa.length; i++)  
        {  
            String s = (String) sa[i];  
            System.out.println (s);  
        }  
    }  
}
```

- The LinkedList class

- The LinkedList class extends AbstractSequentialList and implements the List interface.
- It provides a linked-list data structure.
- It has two constructors as shown below
 - LinkedList(): It builds an empty linked-list
 - LinkedList(Collection c): This constructor builds a linked list that is initialized with the elements of the collection c
- In addition to the methods that it inherits, the LinkedList class defines some useful methods of its own, as shown below, for manipulating and accessing lists.
- void addFirst(Object obj): Adds elements to the start of the list.
- void addLast(Object obj): Adds elements to the last of the list.
- Object getFirst(): Returns the first element present in the list.
- Object getLast(): Retrieves the last element from the list.
- Object removeFirst(): Removes or deletes the first element from the list.
- Object removeLast(): Removes the last element from the list.

(P.T.O)

The following program illustrates several of the methods supported by the LinkedList.

```
import java.util.*;  
  
class LinkedListDemo  
{  
    public static void main (String args[])  
    {  
        LinkedList ll = new LinkedList ();  
  
        ll.add ("MSD");  
        ll.add ("SSA");  
        ll.add ("MGR");  
        ll.add ("PV");  
        ll.add ("PS");  
        ll.addLast ("MSR");  
        ll.addFirst ("MSS");  
        ll.add (3, "MSR");  
  
        System.out.println ("Content of the list : "+ll);  
  
        ll.remove ("PV");  
        ll.remove (4);  
  
        System.out.println ("Content of the list , after deletions : "+ll);  
  
        ll.removeFirst ();  
        ll.removeLast ();  
  
        System.out.println ("After removing first & last elements : "+ll);  
  
        Object value = ll.get (1);  
        ll.set (1, (String) value + " MSD");  
  
        System.out.println ("List , after modification : "+ll);  
    }  
}
```

The output of the above program is as shown below:

contents of the list: MSS MSD SSA MSR MGR PV PS MSK

contents of the list after deletions: MSS MSD SSA MSR PS MSK

After removing first & last elements: MSD SSA MSR PS

List after modification: MSD SSA MSD MSR PS

- The HashSet class:

- HashSet extends AbstractSet and implements the Set interface.
- It creates a collection that uses a hash table for storage.
- A hashtable stores information by using a mechanism called hashing.
- In hashing, the informational content of a key is used to determine a unique value, called its hash code.
- The hash code is then used as the index at which the data associated with the key is stored.
- The advantage of hashing is that it allows the execution time of basic operations, such as add(), contains(), remove() and size() to remain constant even for large sets.
- The following constructors are defined in the HashSet class.
 - (i) HashSet(): It constructs a default HashSet
 - (ii) HashSet(Collection c): This initializes the HashSet by using the elements of c
 - (iii) HashSet(int capacity): This constructor initializes the capacity of the HashSet to the value specified by capacity.

contd....(P.T.O)

(ii) `HashSet<int capacity, float fillRatio>`: This initializes both the capacity and the fillRatio (also called loadFactor) of the HashSet from its arguments.

- The fillRatio must be between 0.0 and 1.0, and it determines how full the HashSet can be before it is resized upward.
- Specifically, when the no. of elements is greater than, the capacity of the HashSet multiplied by its fill ratio, then the HashSet is said to be expanded

NOTE: For constructors that do not take a fillRatio, the default value of 0.75 is used.

- HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

~~** NOTE : HashSet does not define any additional methods~~

HashSet does not guarantee the order of its elements, because the process of hashing does not usually lend itself to the creation of sorted storage.

If we need a sorted storage, then another collection, such as TreeSet is a better choice.

- The following program demonstrates a HashSet

```
import java.util.*;  
class HashSetDemo  
{  
    public static void main (String args[])  
    {  
        HashSet hs = new HashSet();
```

```

        hs.add ("B");
        hs.add ("A");
        hs.add ("D");
        hs.add ("E");
        hs.add ("C");
        hs.add ("F");
        System.out.println ("The set is : " + hs);
    }
}

```

A possible output of the above program may be

The set is : A F E D C B.

NOTE: As explained, the elements are not stored in sorted order and the precise output may vary after each execution of the program.

- The LinkedHashSet class:
- This class extends HashSet, but adds no members of its own.
- LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted.
- When cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
- In the above program, if we use LinkedHashSet in place of HashSet then the output will be as follows:

The set is : B A D E C F

(P.T.O)

- The TreeSet class:

- TreeSet provides an implementation of the Set interface that uses a tree for storage.
- Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
- The following are the constructors defined in this class

TreeSet(): constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

TreeSet(Collection c): Builds a tree that contains the elements of c

TreeSet(Comparator comp): constructs an empty tree set that will be sorted according to the comparator specified by comp.

TreeSet(SortedSet ss): Builds a tree set that contains the elements of ss.

The following program demonstrates a TreeSet

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

```
class TreeSetDemo
```

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

```
{ TreeSet ts = new TreeSet(); }
```

```

    ts.add ("c");
    ts.add ("A");
    ts.add ("B");
    ts.add ("E");
    ts.add ("F");
    ts.add ("D");
    s.o.p(ts);
}
}

```

The output of this program will be: A B C D E F

NOTE: As explained, because TreeSet stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

- The Iterator interface & ListIterator interface

Many a times we may need to cycle through the elements in a collection. For example, we may want to display each element.

- The easiest way to do this is to employ an iterator.
- An iterator is an object that implements either the Iterator or the ListIterator interface.
- Iterator enables us to cycle through a collection, obtaining or removing elements.
- ListIterator extends Iterator to allow bidirectional traversal of a list and modification of the elements.

(P.T.O)

The methods declared by the Iterator interface are:

- boolean hasNext(): Returns true if there are more elements. otherwise returns false.
- Object next(): Returns the next element. Throws NoSuchElementException if there is not a next element.
- void remove(): Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next().

The methods declared by the ListIterator interface are:

- void add(Object obj): Inserts obj into the list in front of the element that will be returned by the next call to next().
- boolean hasNext(): Returns true if there is a next element. otherwise returns false.
- boolean hasPrevious(): Returns true if there is a previous element. otherwise returns false.
- Object next(): Returns the next element. A NoSuchElementException is thrown if there is not a next element.
- int nextIndex(): Returns the index of the next element. If there is not a next element, returns the size of the list.
- Object previous(): Returns the previous element. A NoSuchElementException exception is thrown if there is not a previous element.

- int hashCode(): Returns the hash code of the current element. Returns -1 if there is no a present element.
 - void remove(): Removes the current element from the list. An IllegalStateException is thrown if remove() is called before next() or previous() is invoked.
 - void set (Object obj): Assigns obj to the current element. This is the element last returned by a call to either next() or previous().
 - In general, to use an iterator to cycle through the contents of a collection, we follow the below given steps.
 - (i) obtain an iterator to the start of the collection by calling the collection's iterator() method.
 - (ii) Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext() returns true.
- The following program demonstrates Iterator and ListIterator.

```

import java.util.*;
class IteratorDemo
{
    public static void main (String args[])
    {
        ArrayList al = new ArrayList();
        al.add ("C");
        al.add ("A");
        al.add ("E");
        al.add ("B");
        al.add ("D");
        al.add ("F");
    }
}
  
```

Ans
contd... (P.T.O.)

```
s.o.p (" original contents of al : ");
Iterator itr = al.iterator();
while (itr.hasNext())
{
    Object element = itr.next();
    s.o.print (element + " ");
}
s.o.p ();
ListIterator litr = al.listIterator();
while (litr.hasNext())
{
    Object element = litr.next();
    litr.set (element + "+");
}
s.o.p (" modified contents of al : ");
itr = al.iterator();
while (itr.hasNext())
{
    Object element = itr.next();
    s.o.print (element + " ");
}
s.o.p ();
s.o.p (" modified list backwards : ");
while (itr.hasPrevious())
{
    Object element = itr.previous();
    s.o.print (element + " ");
}
s.o.p ();
}
} //main
} // class
```

The output of the above program will be:

Original contents of al: C A E B D F

Modified contents of al: C+ A+ E+ B+ D+ F+

Modified list backwards: F+ D+ B+ E+ A+ C+

- The RandomAccess interface

- This interface contains no members
- By implementing this interface, a collection signals that it supports efficient random access to its elements.
- RandomAccess is implemented by ArrayList and by the legacy Vector class

- MAPS:

- A map is an object that stores associations between keys and values, or key/value pairs.
- Given a key, we can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps accept a null key and null values whereas others do not.
- The following interfaces support maps:
 - (i) Map
 - (ii) Map.Entry
 - (iii) SortedMap

(P.T.O)

- The Map interface:

- The Map interface maps unique keys to values.
- A key is an object that we use to retrieve a value at a later date.
- Given a key and a value, we can store the value in a Map object.
- After the value is stored, we can retrieve it by using its key.
- The following are the methods declared in the Map interface.

void clear()

boolean containsKey(Object k)

boolean containsValue(Object v)

Set entrySet()

boolean equals(Object obj)

Object get(Object k)

int hashCode()

boolean isEmpty()

Set keySet()

Object put(Object k, Object v)

void putAll(Map m)

Object remove(Object k)

int size()

Collection values()

- Maps revolve around two basic operations
 - `get()` and `put()`
- To put a value into a map, use `put()`, specifying the key and the value.
- To obtain a value call `get()`, passing the key as an argument. The value will be returned.

NOTE: maps are not collections because they do not implement the collection interface, but we can obtain a collection-view of a map. To do this, we can use the `entrySet()` method. It returns a Set that contains the elements in a map.

- To obtain a collection-view of the keys, use `keySet()`.
- To get a collection-view of the values, use `values()`.
- Collection-views are the means by which maps are integrated into the collections framework.
- The SortedMap interface:

- The `SortedMap` interface extends `Map`.
- It ensures that the entries are maintained in ascending key order.
- Sorted maps allow very efficient manipulations of submaps (i.e., a subset of a map).
- To obtain a submap, use `headMap()`, `tailMap()` or `subMap()`.
- To get the first key in the set call `firstKey()` and to get the last key call `lastKey()`.

contd... (P.T.O)

- The methods defined by SortedMap interface are shown below

Comparator comparator()

Object firstKey()

SortedMap headMap(Object end)

Object lastKey()

SortedMap subMap(Object start, Object end)

SortedMap tailMap(Object start)

- The Map.Entry interface:

- The Map.Entry interface enables us to work with a map entry.
- we know that entrySet() method declared by the Map interface returns a Set containing the map entries. each of these set elements is a Map.Entry object.
- The following are the methods declared by this interface.

boolean equals(Object obj)

Object getKey()

Object getValue()

int hashCode()

Object setValue()

The Map classes

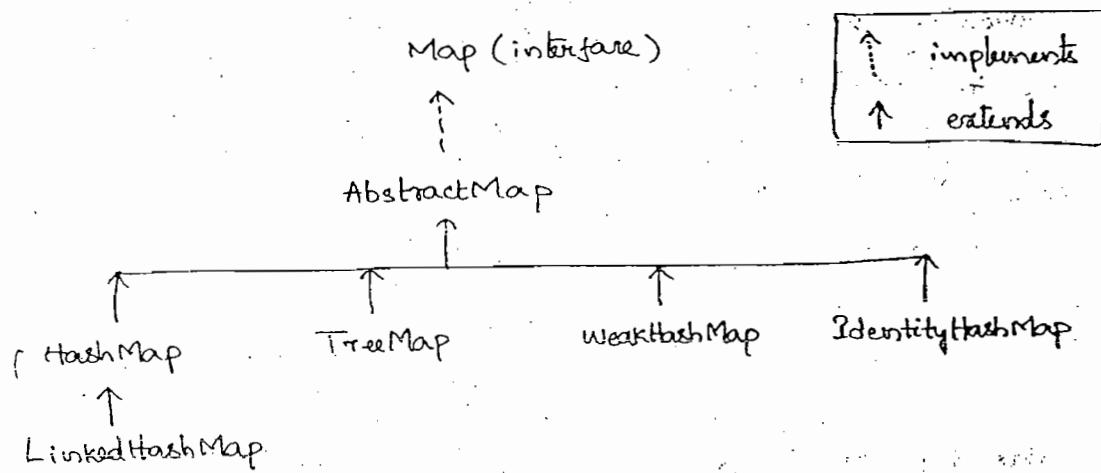
Several classes provide implementations of the map interfaces.

The classes that can be used for maps are as shown below

AbstractMap, HashMap, TreeMap, WeakHashMap,

LinkedHashMap, IdentityHashMap

The class hierarchy is,



The HashMap class

- The HashMap class uses a hash table to implement the Map interface
- This allows the execution time of basic operations, such as `get()` and `put()`, to remain constant even for large sets.
- The following are the constructors defined in `HashMap` class

`HashMap()`

`HashMap(Map m)`

`HashMap(int capacity)`

`HashMap(int capacity, float fillRatio)`

- HashMap implements Map and extends AbstractMap. It does not add any methods of its own.

NOTE: A hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates a HashMap. It maps names to account balances.

```

import java.util.*;
class HashMapDemo
{
    public static void main (String args[])
    {
        HashMap hm = new HashMap();
        hm.put ("MSD", new Double (2218.40));
        hm.put ("SSA", new Double (8122.40));
        hm.put ("MSS", new Double (4037.22));
        hm.put ("MSR", new Double (3740.18));
        hm.put ("SRK", new Double (4730.18));

        Set set = hm.entrySet();
        Iterator i = set.iterator();
        while (i.hasNext())
        {
            Map.Entry me = (Map.Entry) i.next();
            System.out.print (me.getKey () + ":");
            System.out.println (me.getValue ());
        }
    }
}

```

contd... (P.T.O)

```

    // deposit 2218 into SSA's account
    double balance = ((Double) hm.get("SSA")).doubleValue();
    hm.put("SSA", new Double(balance + 2218));
    System.out.println("SSA's new balance: " + hm.get("SSA"));
}

}

```

Explanation:

- The program begins by creating a hash map and then adds mapping of names to balances by ~~key-value~~ associations.
- Next the contents of the map are displayed by using a set-view obtained by calling entrySet().
- The keys and values are displayed by calling the getKey() and getValue() methods that are defined by Map.Entry.

NOTE: The put() method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after SSA's account is updated, the hash map will still contain just one "SSA" account.

• The TreeMap class

- The TreeMap class implements the Map interface by using a tree.
- A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- A tree map guarantees that its elements will be sorted in ascending key order.

4.11. (P.T.O)

4.12. (P.T.O)

- The following constructors are defined in TreeMap class.

TreeMap()

TreeMap(Comparator comp)

TreeMap(Map m)

TreeMap(SortedMap m)

- TreeMap implements SortedMap and extends AbstractMap.

It does not define any additional methods of its own.

- The following program illustrates TreeMap.

Consider the previous example (HashMapDemo)

```
import java.util.*;  
  
class TreeMapDemo  
{  
    public static void main (String args[])  
{  
        TreeMap tm = new TreeMap();  
  
        tm.put("MSD", new Double(2218.40));  
        :  
        tm.put("SRK", new Double(4730.18));  
  
        Set set = tm.entrySet();  
        Iterator i = set.iterator();  
  
        while (i.hasNext())  
        {  
            Map.Entry me = (Map.Entry)i.next();  
            System.out.print(me.getKey() + ":" );  
            System.out.println(me.getValue());  
        }  
    }  
}
```

The output of the above program is:

MSD : 2218.40

MSR : 3740.18

MSS : 4037.22

SRK : 4730.18

SSA : 8122.40

- As mentioned earlier, the elements are sorted in ascending key order.

NOTE: TreeMap sorts the keys. However in this case, they are sorted by first letter (in fact first name). We can even make the TreeMap to sort the keys by last name (provided the key has first name and last name). This can be done by specifying a comparator when the map is created.

- The LinkedHashMap class:

- This class extends HashMap
- LinkedHashMap maintains a linked list of the entries in the map, in the order in which they were inserted. Thus, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted.
- we can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.
- LinkedHashMap defines the following constructors.

LinkedHashMap(): constructs a default LinkedHashMap

LinkedHashMap(Map m): initializes the LinkedHashMap with the elements from m.

`LinkedHashMap (int capacity)` : initializes the LinkedHashMap with the capacity specified.

`LinkedHashMap (int capacity, float fillratio)` : initializes with both - capacity and fillratio.

`LinkedHashMap (int capacity, float fillratio, boolean order)` :

This form allows us to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If order is true, then access order is used. If order is false, then insertion order is used.

- `LinkedHashMap` adds only one method to those defined by `HashMap`.

`protected boolean removeEldestEntry (Map.Entry e)`

- The IdentityHashMap class :

- This class ^{extends} implements `HashMap`
- It is similar to `HashMap` except that it uses reference equality when comparing elements.

NOTE: The Java 2 documentation explicitly states that

`IdentityHashMap` is not for general use.

Comparators:

- Both TreeSet and TreeMap store elements in sorted order.
- By default, these classes store their elements by using the natural ordering which we usually expect.
- If we want to order the elements in a different way, then specify a Comparator object either when we construct the set or map. Doing so gives us the ability to govern precisely how elements are stored within sorted collections and maps.
- The Comparator interface defines two methods:
 - compare() and equals().
- int compare(Object obj1, Object obj2)
 - obj1 and obj2 are objects to be compared.
 - This method returns zero if the objects are equal.
 - It returns a positive value if obj1 is greater than obj2 otherwise a negative value is returned.
- By overriding compare(), we can alter the way that objects are ordered.
- For example, to sort in reverse order, we can create a comparator that reverses the outcome of a comparison.
- boolean equals(Object obj)
 - This method tests whether an object equals the invoking comparator.

NOTE: overriding equals() is unnecessary

The following example demonstrates the use of a comparator.
It causes a tree set to be stored in reverse order.

```
import java.util.*;  
  
class MyComp implements Comparator  
{  
    public int compare (Object a, Object b)  
    {  
        String astr = (String)a;  
        String bstr = (String)b;  
        return bstr.compareTo(astr);  
    }  
}
```

```
class CompDemo  
{  
    public static void main (String args [])  
    {  
        TreeSet ts = new TreeSet (new MyComp());  
        ts.add ("c");  
        ts.add ("A");  
        ts.add ("B");  
        ts.add ("E");  
        ts.add ("F");  
        ts.add ("D");  
        Iterator i = ts.iterator();
```

contd....(P.T.O)

```

    {
        Object element = i.next();
        System.out.print(element + " ");
    }
    System.out.println();
}
}

Output: F E D C B A

```

Explanation: On the compare() method, the String method compareTo() compares the two strings. However bstr invokes compareTo() rather than astr. This causes the outcome of the comparison to be reversed.

The Collection Algorithms:

- The collections framework defines several algorithms that can be applied to collections and maps.
- These algorithms are defined as static methods within the Collections class.
- We know that none of the standard collection implementations are synchronized.
- To obtain synchronized (thread-safe) copies of the various collections we use synchronizedList() and synchronizedSet().
- The set of methods that begins with unmodifiable returns views of the various collections that cannot be modified. These will be useful when we want to grant some process read-but-not-write-capabilities on a collection.

- The Legacy classes and Interfaces
- The original version of java.util did not include the collections framework.
- Instead, it defined several classes and an interface that provided an ad hoc method of storing objects.
- with the addition of collections by Java 2, several of the original classes were reengineered to support the collection interfaces. Thus, they are fully compatible with the framework.

**
NOTE: None of the collection classes are synchronized, but all the legacy classes are synchronized.

- The legacy classes defined by java.util are
Dictionary HashTable Properties Stack Vector
- one legacy interface is Enumeration.
- The Enumeration Interface:
- The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- This legacy interface has been superceded by Iterator.
- Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as Vector & Properties).

- Enumeration specifies the following two methods.
 - boolean hasMoreElements(): Returns true while there are still more elements to extract and returns false when all the elements have been enumerated.
 - nextElement(): Returns the next object in the enumeration as a generic object reference i.e. each call to nextElement() obtains the next object in the enumeration. The calling routine must cast that object into the object type held in the enumeration.

- The Vector class:

- vector implements the concept of dynamic array.
- it is similar to ArrayList but with two differences:
 - Vector is synchronized, and it contains many legacy methods that are not part of the collections framework.
 - with the release of Java 2, vector was reengineered to extend AbstractList and implement the List interface, so it is now fully compatible with collections.
- The following are the constructors of Vector class.

Vector<>

Vector<int size>

Vector<int size, int incr>

Vector<collection c>

(P.T.O)

- The first constructor creates a default vector, which has an initial size of 10.
- The second constructor creates a vector whose initial capacity is specified by size.
- The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the no. of elements to allocate each time that a vector is resized upward.
- The fourth constructor creates a vector that contains the elements of collection c.
- All vectors start with an initial capacity. After this initial capacity is reached, the next time when we attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects.
- By allocating more than just the required memory, the vector reduces the no. of allocations that must take place.
- This reduction is important, because all allocations are costly in terms of time.
- The amount of extra space allocated during each reallocation is determined by the increment that you specify when we create the vector.
- If we don't specify an increment, the vector's size is doubled by each allocation cycle.

- vector defines the following protected data members:
 - int capacityIncrement : The increment value is stored in capacityIncrement
 - int elementCount : The no. of elements currently present in the vector is stored in elementCount.
 - Object elementData [] : The array that holds the vector is stored in elementData.
- In addition to the collections methods defined by List, vector defines several legacy methods as shown below.

void addElement (Object element)

int capacity ()

Object clone ()

boolean contains (Object element).

void copyInto (Object array [])

Object elementAt (int index)

Enumeration elements ()

void ensureCapacity (int size)

Object firstElement ()

int indexOf (Object element)

int indexOf (Object element, int start)

void insertElementAt (Object element, int index)

boolean isEmpty ()

Object lastElement ()

contd....(P.T.O)

```
int lastIndexOf (Object element, int start)  
void removeAllElements ()  
boolean removeElement (Object element)  
void removeElementAt (int index)  
void setElementAt (Object element, int index)  
void setSize (int size)  
int size ()  
String toString ()  
void trimToSize ()
```

- The following program uses a vector to store various types of numeric objects. It demonstrates several of the legacy methods defined by Vector. It also demonstrates the Enumeration interface.

```
import java.util.*;  
  
class VectorDemo  
{  
    public static void main (String args [])  
    {  
        Vector v = new Vector (3, 2); // Initial size is 3 and  
                                     // increment is 2  
  
        System.out.println ("Initial size : " + v.size ());  
        System.out.println ("Initial capacity : " + v.capacity ());  
    }  
}
```

contd....(P.T.O)

```

v.addElement (new Integer (1));
v.addElement (new Integer (2));
v.addElement (new Integer (3));
v.addElement (new Integer (4));
s.o.p (" capacity after four additions: " + v.capacity ());
v.addElement (new Double (5.45));
s.o.p (" current capacity: " + v.capacity ());
v.addElement (new Double (6.08));
v.addElement (new Integer (7));
s.o.p (" current capacity: " + v.capacity ());
v.addElement (new Float (9.4));
v.addElement (new Integer (10));
s.o.p (" current capacity: " + v.capacity ());
v.addElement (new Integer (11));
v.addElement (new Integer (12));
s.o.p (" first element: " + (Integer) v.firstElement ());
s.o.p (" last element: " + (Integer) v.lastElement ());
if (v.contains (new Integer (3)))
    s.o.p (" vector contains 3 ");
Enumeration vEnum = v.elements ();
s.o.p (" \n Elements in vector: ");
while (vEnum.hasMoreElements ())
    s.o.print (vEnum.nextElement () + " ");
s.o.p ();
}

//main
}

//class.

```

Initial size : 0

Initial capacity: 3

capacity after four additions: 5

current capacity : 5

current capacity : 7

current capacity : 9

first element : 1

Last element : 12

vector contains 3

Elements in vector:

1 2 3 4 5.45 6.08 7 9.4 10 11 12

NOTE: with the release of java2, vector added support for iterators. Instead of relying on an enumeration object to cycle through the objects, we can now use an iterator.

- Because enumerations are not recommended for the new code, we will usually use an iterator to enumerate the contents of a vector. Fortunately, enumerations and iterators work in nearly the same manner.

- The Stack class:

- Stack is a subclass of Vector that implements a standard last-in, first-out (LIFO) stack.
- Stack only defines the default constructor, which creates an empty stack.
- Stack includes all the methods defined by Vector and in addition to those, defines the following methods.

boolean empty()

Object peek(): Returns the element on the top of the stack, but does not remove it.

Object pop()

Object push(Object element)

int search(Object element): Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise -1 is returned.

- The following program creates a stack, pushes several Integer objects onto it and then pops them off again.

```
import java.util.*;

class StackDemo
{
    static void showpush(Stack st, int a)
    {
        st.push(new Integer(a));
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }
}
```

```
static void showpop (Stack st)
{
    s.o.print (" pop → ");
    Integer a = (Integer) st.pop ();
    s.o.p (a);
    s.o.p (" stack : " + st);
}

public static void main (String args[])
{
    Stack st = new Stack ();
    s.o.p (" stack : " + st);
    showpush (st, 42);
    showpush (st, 66);
    showpush (st, 99);
    showpop (st);
    showpop (st);
    showpop (st);

    try
    {
        showpop (st);
    }

    catch (EmptyStackException e)
    {
        s.o.p (" empty stack ");
    }
}
```

- The output of the above programs is

stack: []

push (42)

stack: [42]

push (66)

stack: [42, 66]

push (99)

stack: [42, 66, 99]

pop → 99

stack: [42, 66]

pop → 66

stack: [42]

pop → 42

stack: []

pop → empty stack.

- The Dictionary class:

- Dictionary is an abstract class that represents a key/value storage repository and operates much like a Map.
- Given a key and a value, we can store the value in a Dictionary object. Once the value is stored, we can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.
- Although not actually deprecated by Java 2, Dictionary is classified as obsolete, because it is superseded by Map.

(P.T.O)

- The following are the abstract methods defined by Dictionary.

Enumeration elements()

Object get(Object key)

boolean isEmpty()

Enumeration keys()

Object put(Object key, Object value)

Object remove(Object key)

int size()

NOTE: The Dictionary class is obsolete. we should implement the Map interface to obtain key/value storage functionality.

- The HashTable class:

- HashTable was part of the original java.util and is a concrete implementation of a Dictionary.
- However, Java2 reengineered HashTable so that it also implements the Map interface. Thus HashTable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.
- Like HashMap, Hashtable stores key/value pairs in a hash table. When using a hashtable, we specify an object that is used as a key and the value that we want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

- The following are the hashtable constructors defined in hashtable:

hashtable()

hashtable(int size)

hashtable(int size, float fillRatio)

hashtable(Map m)

- In addition to the methods defined by the Map interface, which hashtable now implements, Hashtable defines the following legacy methods.

void clear()

Object clone()

boolean contains(Object value)

boolean containsKey(Object key)

boolean containsValue(Object value)

Enumeration elements()

Object get(Object key)

boolean isEmpty()

Enumeration keys()

Object put(Object key, Object value)

void rehash()

Object remove(Object key)

int size()

String toString()

(P.T.O)

- The following program

```
import java.util.*;

class hashtableDemo {

    public static void main (String args)

    {
        hashtable bal = new hashtable();

        Enumeration names;
        String str;
        double balance;

        bal.put ("MSD", new Double (9999.00));
        bal.put ("SSA", new Double (18999.40));
        bal.put ("MSS", new Double (1822.35));
        bal.put ("SSD", new Double (2218.40));

        names = bal.keys();
        while (names.hasMoreElements ())
        {
            str = (String) names.nextElement ();
            s.o.p (str + ":" + bal.get (str));
        }
        s.o.p ();

        balance = (Double) bal.get ("MSD").doubleValue ();
        bal.put ("MSD", new Double (balance + 10000));
        s.o.p ("MSD's new balance: " + bal.get ("MSD"));

    }
}
```

The output of the above program is

SSA : 18999.40

MSD : 9999.00

MSS : 1822.35

SSD : 2218.40

MSD's new balance : 19999.00

NOTE: Like the map classes, Hashtable does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of bal. However, we can obtain set-views of the hash table, which permits the use of iterators. To do so, we simply use one of the collection-view methods defined by Map, such as entrySet() or keySet(). For example, we can obtain a set-view of the keys and iterate through them.

- The Properties class:

- Properties is a subclass of Hashtable.
- It is used to maintain lists of values in which the key is a string and the value is also a string.
- The Properties class is used by many other java classes. For example, it is the type of object returned by System.getProperty() when obtaining environmental ~~variables~~ ^{values}.
- Properties class defines the following instance variable:
Properties defaults; This variable holds a default property list associated with a Properties object.

- The Properties class defines the following constructors:

Properties(): creates a Properties object that has no default values.

Properties(Properties propDefault): creates an object that uses propDefault for its default values.

NOTE: In both the above cases, the property list is empty.

- In addition to the methods that Properties class inherits from Hashtable, Properties class defines the following methods.

NOTE: Properties also contains one deprecated method i.e. save(). This was replaced by store() because save() did not handle errors correctly.

String getProperty(String key)

String getProperty(String key, String defaultValue)

void list(PrintStream streamOut)

void list(Writer streamOut)

void load(InputStream streamIn) throws IOException;

Enumeration propertyNames()

Object setProperty(String key, String value)

void store(OutputStream streamOut, String description).

- one useful capability of the Properties class is that we can specify a default property that will be returned if no value is associated with a certain key.

Q:

Ex: A default value can be specified along with the key in the getProperty() method - such as

```
getProperties ("name", "defaultValue");
```

If the "name" value is not found, then "defaultValue" is returned.

- The following example demonstrates Properties. It creates a property list in which the keys are the names of states and the values are the names of their capitals. An attempt to find the capital of West Bengal includes a default value, in this example.

```
import java.util.*;  
  
class PropDemo  
{  
    public static void main (String args[])  
    {  
        Properties capitals = new Properties();  
        Set states;  
        String str;  
        capitals.put ("AndhraPradesh", "Hyderabad");  
        capitals.put ("TamilNadu", "Chennai");  
        capitals.put ("Karnataka", "Bangalore");  
        capitals.put ("Kerala", "Trivandrum");  
        // Show all states and capitals in a hashtable  
        states = capitals.keySet(); // get set-view of keys.  
        Iterator itr = states.iterator();
```

wantd....(P.T.O)

```

while (itr.hasNext())
{
    str = (String) itr.next();
    s.o.p("The capital of " + str + " is " + capitals.getProperty(str));
}

s.o.p();
// look for a state, not in the list -- specify default.
str = capitals.getProperty("WestBengal", "Not Found");
s.o.p("The capital of WestBengal is " + str);
}
}

```

The output of the above program is,

The capital of Andhra Pradesh is Hyderabad

The capital of Tamilnadu is Chennai

The capital of Karnataka is Bangalore

The capital of Kerala is Trivandrum.

The capital of WestBengal is Not Found.

- using store() and load(). One of the most useful aspects of Properties is that the information contained in a Properties object can be easily stored to or loaded from a disk with the store() and load() methods. At anytime, we can write a Properties object to a stream or read it back. This makes property list especially convenient for implementing simple databases.

MORE UTILITY CLASSES

• The StringTokenizer class

- The processing of text often consists of parsing a formatted input string.
- Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning.
- The StringTokenizer class provides the facility of parsing the text. It is often called lexer (lexical analyzer) or scanner.
- StringTokenizer implements the Enumeration interface. Therefore, given an input string, we can enumerate the individual tokens contained in it using StringTokenizer.
- To use StringTokenizer, we can specify an input string and a string that contains delimiters.
- Delimiters are characters that separate tokens. Each character in the delimiter string is considered a valid delimiter.
- The default set of delimiters consists of whitespace characters such as space, tab, newline etc.
- In general, 'space' is used as the default delimiter.

(P.T.O)

- The following are the constructors in `StringTokenizer`:
`StringTokenizer(String str)`
`StringTokenizer(String str, String delimiter)`
`StringTokenizer(String str, String delimiters, boolean delimsAsTokens)`

- In all the three constructors mentioned above, `str` is the string that will be tokenized.
- In the first constructor, the default delimiter (space) is used.
- In the second and third constructors, `delimiters` is a string that specifies the delimiters.
- In the third constructor, if `delimsAsTokens` is true, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.
- The following are the methods defined in `StringTokenizer`.

```
int countTokens()
boolean hasMoreElements()
boolean hasMoreTokens()
Object nextElement()
String nextToken()
String nextToken(String delimiters)
```

- The following program demonstrates a StringTokenizer.

```
import java.util.StringTokenizer;  
class STDemo  
{  
    static String str = "This-is-my-core-java-notes";  
    public static void main (String args[])  
    {  
        StringTokenizer st = new StringTokenizer(str, "-");  
        while (st.hasMoreTokens ())  
        {  
            String s1 = st.nextToken();  
            System.out.println (s1);  
        }  
    }  
}
```

Output:

This is my core java notes ↓ (printed vertically).

- The Date class:

- The Date class encapsulates the current date and time.

- Date class supports the following constructors

Date () & Date (long millisec)

- The first constructor initializes the object with current system date and time.

- The second constructor accepts one argument that equals the no. of milliseconds that have elapsed since midnight, Jan 1, 1970.

- Date class also implements Comparable interface.

- The following are the non-deprecated methods defined in the Date class.

boolean after (Date date)

boolean before (Date date)

Object clone ()

int compareTo (Date date)

int compareTo (Object obj)

boolean equals (Object date)

long getTime ()

int hashCode ()

void setTime (long time)

String toString ()

- From the above methods, it is obvious that, the Date features do not allow us to obtain the individual components of the date or time. To obtain more-detailed information about the date and time, we use the Calendar class.

- The Calendar class:

- The `Calendar` class is abstract and provides a set of methods that allows us to convert a time in milliseconds into a no. of useful components such as year, month, day, hour, minute and second.

- It is intended that subclasses of `Calendar` will provide the specific functionality to interpret time information according to their own rules. An example of such a class is `GregorianCalendar`.

NOTE: `Calendar` provides no public constructors.

- The `GregorianCalendar` class:

- `GregorianCalendar` is a concrete implementation of a calendar.
- `GregorianCalendar` is a concrete implementation of a calendar. The `getInstance()` method of `Calendar` returns a `GregorianCalendar` initialized with the current date and time in the default locale and time zone.

- The following are the constructors defined in the `GregorianCalendar`:

`GregorianCalendar()`

`GregorianCalendar(int year, int month, int dayOfMonth)`

`GregorianCalendar(int y, int m, int dom, int hours, int minutes)`

`GregorianCalendar(int year, int month, int dayOfMonth,
int hours, int minutes, int seconds)`

- `GregorianCalendar` provides an implementation of all the abstract methods in `Calendar`. It also provides some useful and interesting methods.

Ex: `isLeapYear()` : returns true if the year is a leap year and false otherwise

The signature of this method is

`boolean isLeapYear(int year).`

- The TimeZone class

- This is another time-related class.
- The TimeZone class allows us to work with time zone offsets from Greenwich mean time (GMT).
- It also computes daylight saving time.
- TimeZone supplies only the default constructor.

- The SimpleTimeZone class:

- The SimpleTimeZone class is a convenient subclass of TimeZone.
- It provides bodies for the abstract methods defined in TimeZone class and allows us to work with time zones for a Gregorian calendar.
- It also computes daylight saving time.
- SimpleTimeZone defines four constructors with different arguments.

- The Locale class

- The Locale class is instantiated to produce objects that each describe a geographical cultural region.
- It is one of the several classes that provides us with the ability to write programs that can execute in several different international environments.

Ex: The formats used to display dates, times and numbers are different in various regions.

- The Locale class defines some constants that are useful for dealing with the most common locales.
Ex: The expression Locale.FRENCH represents the Locale object for French.
- The following are the constructors defined in Locale class.
 - Locale (String language)
 - Locale (String language, String country)
 - Locale (String language, String country, String data).
- The above constructors build a Locale object to represent a specific language and in the case of second and third constructors, also a specific country.
- Locale defines several methods, which return human-readable strings that can be used to display the name of the country, the name of the language and the complete description of the locale.

NOTE: Calendar and GregorianCalendar are examples of classes that operate in a locale-sensitive manner.
DateFormat and SimpleDateFormat also depend on the locale.

- The Random class:

- The Random class is a generator of pseudorandom numbers.
- These are called pseudorandom numbers because they are simply uniformly distributed sequences.

... contd ... (parto)

- The following are the constructors defined in Random class.

Random(): creates a number generator that uses the current time as the starting or seed value.

Random(long seed): This form allows us to specify a seed value manually.

- If we initialize a Random object with a seed, we define the starting point for the random sequence.
- If we use the same seed to initialize another Random object we will extract the same random sequence.
- If we want to generate different sequences, then we should specify different seed values. The easiest way to do this is to use the current time to seed a Random object. This approach reduces the possibility of getting repeated sequences.

- The Observable class:

- The Observable class is used to create subclasses that other parts of our program can observe.
- When an object of a such a subclass undergoes a change, observing classes are notified.
- Observing classes must implement the Observer interface, which defines the update method. The update() method is called when an observer is notified of a change in an observed object.

- An object that is being observed must follow two simple rules:
 - (i) First, if it has changed, it must call `setChanged()`.
 - (ii) When it is ready to notify observers of this change, it must call `notifyObservers()`. This causes `update()` method in the observing objects to be called.

NOTE: If the object calls `notifyObservers()` without having previously called `setChanged()`, no action will take place. The observed object must call both `setChanged()` and `notifyObservers()` before `update()` will be called.

- The Observer interface:

- To observe an observable object, we should implement the Observer interface.
- This interface defines only one method as shown below.

```
void update( Observable observable, Object arg)
```

Here `observable` is the object being observed, and `arg` is the value passed by `notifyObservers()`. The `update()` method is called when a change in the observed object takes place.

- Timer and TimerTask:

- Java 2, version 1.3 added an interesting and useful feature to `java.util` i.e. the ability to schedule a task for execution at some future time. The classes that support this feature are `Timer` and `TimerTask`.

(P.T.O)

- Using these three classes we can create a thread that runs in the background, waiting for a specific time. When the time arrives, the task linked to that thread is executed.
- Although it was always possible to manually create a task that would be executed at a specific time using the Thread class, Timer and TimerTask greatly simplify this process.
- Timer and TimerTask work together.
- Timer is the class that we will use to schedule a task for execution. The task being scheduled must be an instance of TimerTask.
- Thus, to schedule a task, we will first create a TimerTask object and then schedule it for execution using an instance of Timer.
- TimerTask implements the Runnable interface; thus it can be used to create a thread of execution.
- The easiest way to create a timer task is to extend TimerTask and override run().

- The Currency class:

- Java version 1.4 adds the Currency class.
- This class encapsulates information about a currency.
- It defines no constructors.