- LECTURER NOTES -

UNIT-I

OVERVIEW OF JAVA:

- Java is general purpose, object oriented language developed by Sun Microsystem of USA in 1991.
- Originally called as 'oak' by James Gosling, one of the inventors of language.
- Java was designed for development of sfw for consumer electronic devices like TV's, VCR's, toaster, & other electronic m/cs.
- Java is modelled on C & C++ but removed mol. of features of C & C++ that were considered as source of problem and making Java a really simple, reliable, Portable & Powerful lang.

JAVA FEATURES:

- Compiled & Interpreted:
- Platform Independent & Portable
- Object oriented
- Robust & Secure
- Distributed
- Simple, Small & Familiar
- Multithread & Interactive

- High Performance
- Dynamic & Extensible
- Scalability & Performance.

Compiled & Interpreted:

→ Java is both compiled & interpreted
→ Java compiler (javac) translates source code to bytecode instr.
→ Bytecode are not m/c instructions and in second stage Java interpreter (java) generates m/c code that can be directly executed by m/c running Java Pgm.

Platform independent & Portable:

→ Portability is the main feature of Java
→ Java programs can be easily moved from one computer to another
→ Changes & upgradation in os, processor & sys Resources will not force any changes in Java Pgm.
→ We can download Java applet from remote computer into our local system via internet & execute it locally.

Object oriented:

→ Java is true object oriented language (i.e) Everything in Java is object.
→ All Pgm & data reside within object & classes

Robust & Secure:

- Java is a robust language
- It has strict compile time & runtime checking for datatypes.
→ It is automatic garbage - collected language
→ It not only verify all memory access but also ensure that no virus can be communicated with an applet.
→ The absence of pointers in Java ensures that pgms cannot gain access to mem. locations without proper authorization.

Distributed:

→ Designed as distributed pgm. language for creating applns on n/w.
→ Has ability to share both data & pgm and Java applns can open & access remote objects on internet as they do in local sm.

Simple - Small & Familiar:

→ It is small & simple lang.
→ Java doesn't use pointer, preprocessor header files, goto stmts.
→ It also eliminates operator overloading & multiple inheritance.
→ It is familiar since it is modelled on C & C++

Multithreaded & Interactive:

→ Multithreaded means handling multiple task simultaneously
→ Java support multithread pgms (ie) no need to wait for applns to finish one task before begining another.
(eg) we can listen to audio clip when scrolling page

High Performance:

→ Java speed is comparable to native C/C++

→ The incorporation of multithreading enhances overall execution speed of Java Pgms.

Dynamic & Extensible:

1. Java is dynamic language & capable of linking new class libraries, methods & objects
2. Java Pgm support fns written in other Pgm. languages such as C/C++. These fns are called as 'native methods'.
3. Native methods are linked dynamically at runtime.

Scalability & Performance:

→ It is improved by ~~improved by~~ improving the startup time & reducing amount of memory used in Java 2 Runtime Environment

→ Memory utilization is reduced by sharing data in shared archive among multiple JVM Process.

Java & C++:

→ Java is true (pure) object oriented lang.

→ Doesn't support operator overloading

→ Doesn't have template classes as in C++.

→ Doesn't support multiple inheritance but it is accomplished by 'interface'.

→ Doesn't use pointers.

→ It has replaced destructor fn. with finalizer() fn.

JAVA ENVIRONMENT:

→ Includes large no. of development tools & hundreds of class & Methods.

→ The tools are Part of JDK (Java Development kit) & classes & methods are Part of JSL (Java Standard library) known as API (Application programming Interface).

Java Development kit:

→ consists of collection of tools used for developing & running Java Pgms.

(*) appletviewer (for viewing Java applets)
(*) Javac (Java compiler)
(*) Java (Java interpreter)
(*) javap (Java disassembler)
(*) javah (for c header files)
* Javadoc
* Jdb.

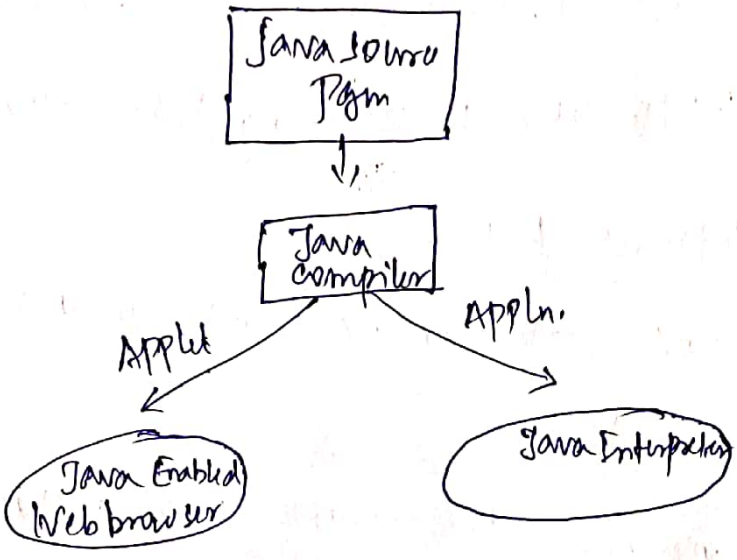
OVERVIEW OF JAVA:

→ Java is general Purpose, object oriented Programming lang.
→ we can develop 2 types of Pgms (i) Stand alone applns.
(ii) Web applets.
→ Executing Stand alone applications involves 2 steps
(1) Compiling source code to byte code using Java compiler
(2) Executing byte code Pgm using Java interpreter.

→ Applets are small Java Pgms developed for Internet applications.

→ Applets can do everything from simple animated graphics to complex games & utilities.

→ Applets can be embedded in HTML document & run inside webpage



SIMPLE JAVA PROGRAM:

```
Class SampleOne
{
Public static void main ( String args[])
  {
   System.out.println (" Java is better than C++ ");
  }
}
```

Public: Declares the main method as unprotected & making it accessible to all other classes. Similar to C++ Public modifier.

Static: Declares that this method belongs to entire class & not part of any object of class. It is declared

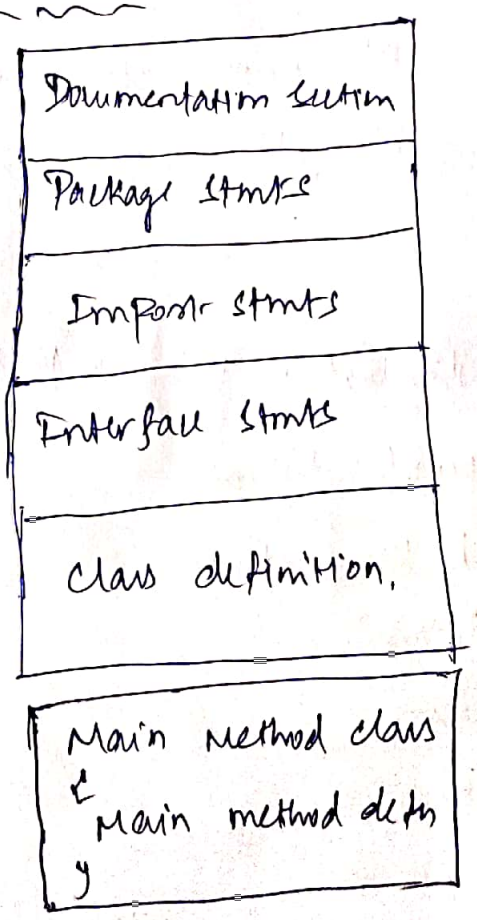
- as static since interpreter uses this method before any objects are created.

void - Main() method doesn't return any value.

Here String args[] declares Parameter named args which contain array of objects of class type 'String'.

→ println method is member of 'out' object which is static data member of 'System' class.

JAVA PROGRAM STRUCTURE:



Documentation section - comprises set of comment lines giving name of Pgm, author & other details

Package stmts - First stmt in Java is package stmt.

- Declares Package name & informs compiler that class defined here belongs to this Package
eg: Package student.

Import stmts - Similar to include stmt. & it instructs interpreter to load test-class contained in Package 'student'.

Interface stmts: like class but include group of method declarations.

Class Defn :- Contain multiple class defns.

Main method class - Starting point of execution. Creates Objs of various class & establishes communicat. btw them.

JAVA TOKENS:

→ Smallest individual unit in a Program are known as token

→ It has 5 types of tokens

(1) Reserved keywords
(2) Identifiers
(3) Literals
(4) Operators
(5) Separators.

Keywords:-

→ Java has 50 words as keywords.

→ They have specific meanings and can't use as name for class, variables, methods

→ All keywords are with lower case letters.

Identifiers:

→ They are used to name classes, methods, variables, Packages, Interfaces

→ Rules for identifiers are

(1) Have alphabets, digits, underscore & dollar sign characters

(2) Not begin with digit

(3) Upper & lower case are distinct

(4) Can be of any length.

Literals:

→ They are sequence of characters that represent constant values to be stored in variables.

⇒ 5 types of literals

(*) Integer literals

(*) Floating point literals

(*) Character literals

(*) String literals

(*) Boolean literals

(*) Null literals.

Operators:

→ They are symbols that take one (or) more operators & operate on them.

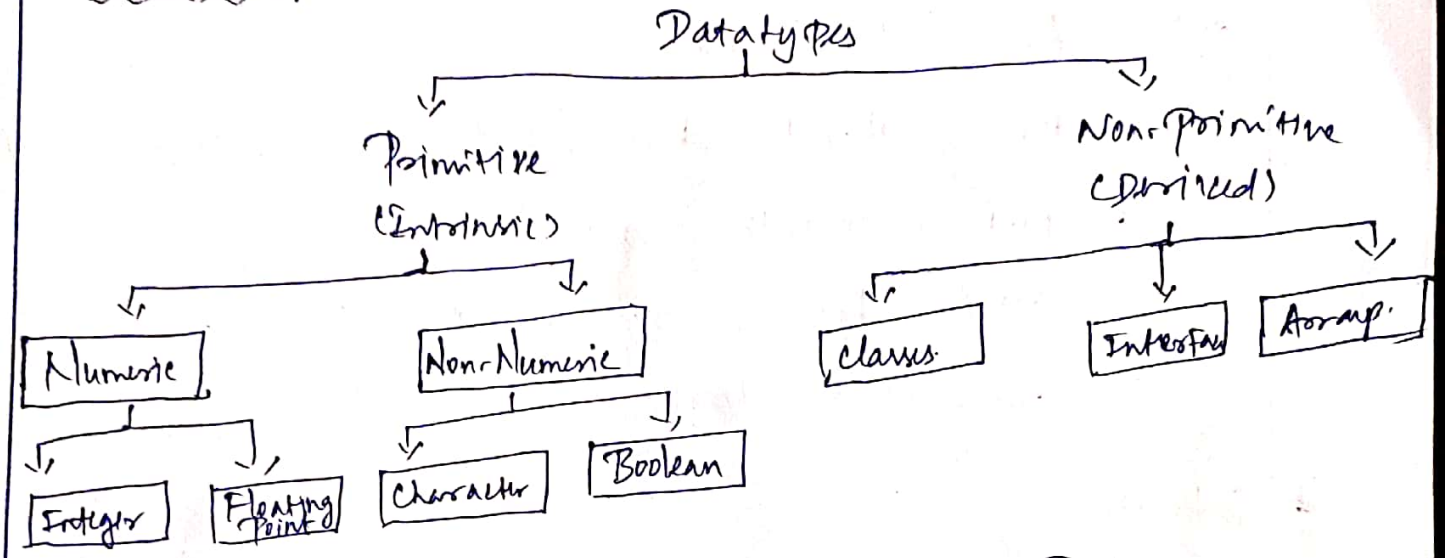
Separators:

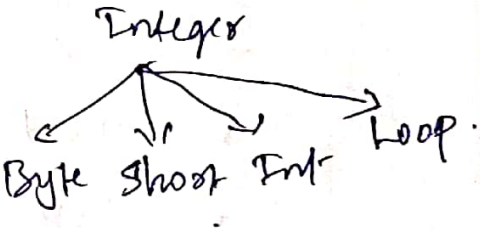
→ Symbols used to indicate where group of code are divided &

and arranged

-> Some of separators are (1) Parantheses - ( ) (2) braces { } (3) brackets [ ], (4) semicolon ; (5) comma (6) period .

DATATYPES:

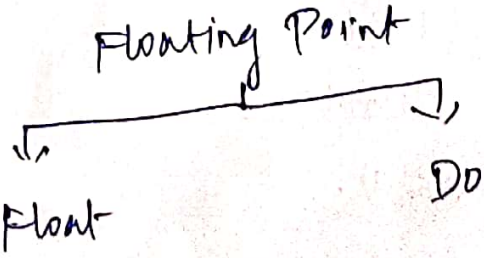




Integer Types:

byte - one byte
short - two bytes
int - four bytes
long - Eight bytes

Floating Point:



Float - 4 bytes.
Double - 8 bytes.

Character: size - 2 bytes.

Boolean: 1 bit.

SCOPE OF VARIABLES:

-> Java variables are classified as 3 kinds

(1) Instance Variables

(2) Class Variables

(3) Local Variables.

Instance Variables -> * Declared inside a class

(*) Created when the objects are created (instantiated) and hence they are associated with objects.

(*) They take diff. values for diff. objects

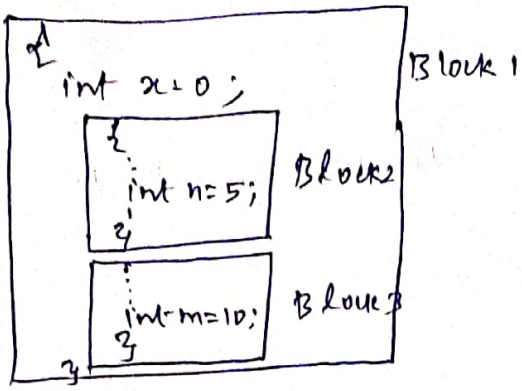
Class Variables:

-> They are global to a class & belong to entire set of objects that class creates.

-> Only one Memory location is created for each class Variable

Local Variables:

- Variables declared & used inside methods are local variables
- They are not available for use outside method defn.
- They are also declared inside pgm blocks that are blw { & }.
- The area of pgm where variable is accessible is called its Scope.



OPERATORS & EXPRESSIONS:

(*) Java has rich set of operators

(*) Operator is a symbol that tells computer to perform certain Mathematical (or) logical manipulations.

(*) Used in Pgms to manipulate data & variables.

(*) They are classified into

(1) Arithmetic operators
(2) Relational operators
(3) Logical operators
(4) Assignment operators
(5) Increment / decrement operators
(6) Conditional operators
(7) Bitwise operators
(8) Special operators.

Arithmetic operators:

→ Used to construct Mathematical expressions

→ Operates on any builtin numeric datatype.

Operator	Meaning
+	Addition / Unary plus
-	Subtraction / Unary Minus
*	Multiplication
/	Division
%	Modulo division.

Relational Operators:

→ used to compare 2 quantities depending on their relation take certain decision

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to.
==	Equal to
!=	Not equal to

Logical Operators:

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

→ used to form compound conditions by combining 2 (or) more relns

→ An expression of this kind which combines 2 or more relational expression is called logical expression

Assignment operator:

→ used to assign value of an expression to a variable.

→ It is '='.

v op= exp;

Increment & decrement operator:

→ ++ & --

m=5;
y=++m;

m=5
y=m++.

Conditional operator. (? :)

→ exp1? exp2 : exp3;

Exp1 is evaluated first if it is true, then exp2 will be evaluated else exp3 will be evaluated.

Bitwise operator:

& - bitwise AND
| - " OR
^ - " Exclusive OR
~ - ONE's complement.
<< - shift left
>> - shift right
>>> - shift right with zero fill

Instanceof operator:

Eg Person instanceof Student -> Return true if Person is object belong to Student class

VARIABLES:

.> Variables are name of Storage locations

-> After putting variable names, we have to declare variable to the compiler.

-> Declaration tells the compilers

(1) What the variable name is

(2) It specifies what type of data the variable will ho

(3) The Place of declaration decides scope of variable

-> Variable should be declared before it is used in Pgm

Eg:

type var1, var2, ... varN;

(Ex) int count;
float x, y;

Assignment Stmt:

var_name = value;

Also

type var_Name = value;

int count = 5;

CLASSES, OBJECTS, METHODS:

→ Java is an object oriented language, so everything should be enclosed within class.

→ Classes define state & behaviour of basic program component known as object.

→ Objects are instance of class & object uses 'methods' to communicate b/w them.

→ classes provide convenient method for packing together group of logical related data items & fns that work on them.

Defining classes:

→ Once class type has been defined we can create 'variables' of that type using declarations that are similar to basic type declarations.

→ In Java, these variables are termed as 'instance of classes' called as Object

Syntax:

```
class  <classname> [extends superclassname]
{
    [fields declaration;]
    [methods declaration;]
}
```

→ There is no semicolon when closing class structure braces.

Field declaration:

→ Data fields are placed inside body of class definition.

→ These Variables are called as 'Instance Variables' as they are created whenever the Object of the class is instantiated

```
Class Rectangle
{
  int length;
  int width;
}
```

→ These Variables are only declared & therefore no storage space has been created in the memory.

→ Instance Variables are also known as 'Member Variables'.

Method declaration:

→ Methods are declared inside body of class but immediately after declaration of instance Variables.

→ The general form is

```
type <methodname> (Parameter_list)
{
  method-body;
}
```

eg:

```
Class Rectangle
{
   int length;
   int width;
   void getdata (int x, int y)
    {
      length = x;
      width = y;
    }
}
```

(ii)

```
Class Rectangle
{
  int length, width;
  void getdata (int x, int y)
   {
     length = x;
     width = y;
   }
   int rectArea ()
    {
      int area = length * width;
      return (area);
    }
}
```

Creating objects:

→ Creating objects is also refered as 'instantiating'

an Object.

→ Objects are created using new operator.

→ The new operator creates an object of specified class & returns reference to that object.

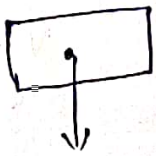
eg

```
Rectangle rect1;          // declare the object
rect1 = new Rectangle();  // instantiate the object
```

(or)

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
```

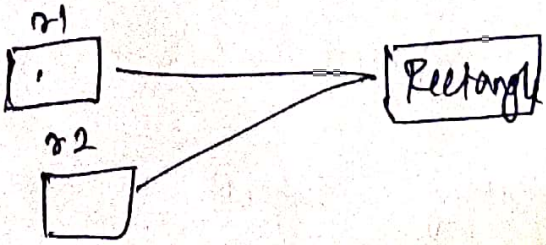
Action	Stmt	Result
Declare	Rectangle rect1;	null rect
Instantiate	rect1 = new Rectangle();	rect1



Rect1 is a Rectangle object reference to Rectangle object.

→ It is possible to create 2 or more Reference to same object.

```
Rectangle r1 = new Rectangle();
Rectangle r2 = r1;
```



→ Each object has its own copy of instance variable of its class

Accessing Members:

→ Each object has its own set of variables & we should assign value before we use it in Pgm.

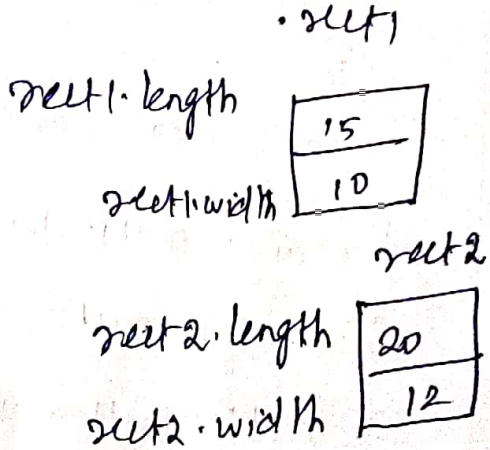
→ We have to access instance variables using concerned object & dot operator

(syntax)

```
Object_name . Variablename = Value;
Object name . method name (parameter_list);
```

(eg, Method 1):

```
rect1 . length = 15
rect1 . width = 10;
rect2 . length = 20;
rect2 . width = 12
```



Method 2:

```
Rectangle rect1 = new Rectangle()
rect1.getdata(15,10);
```

eg Pgm:

```
class Rectangle
{
  int length, width;
  void getdata (int x, int y)
  {
    length = x;
```

```
    width=y;
}
int rectArea()
  {
    int area = length * width;
    return(area);
  }
}
class RectArea
{
Public static Void main(String args[])
  {
    int area1, area2;
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
    rect1.length = 15;
    rect1.width = 10;
    area1 = rect1.length * rect1.width;
    rect2.getdata(20, 12);
    area2 = rect2.rectArea();
    System.out.println("Area1=" + area1);
    System.out.println("Area2=" + area2);
  }
}
```

ARRAYS:

→ An array is a group of contagious (or) related data items that share a common name.

eg: salary[10];

Define array name salary to represent salary of 10 employees.

→ Individual values are called as 'elements'

ONE-DIMENSIONAL ARRAYS:

→ A list of items can be given one variable name using only one subscript and such a variable is called "One-dimensional array".

Creation of array:

→ Arrays must be declared & created in computer memory before they are used.

→ Creation of array consists of

(1) Declaration of array

(2) Creating memory locations

(3) Putting values into memory locations.

Declaration of Array:

→ Array can be declared in

type arrayname[];

eg: int number[];

Creation of Arrays:

→ After declaring, we need to allocate memory for array using 'new operator'.

```
arrayname = new type[size];
```

(ie) `number = new int[5];`

→ The declaration & creation can be combined into

```
int number[] = new int[5];
```

Initialization of arrays:

→ Initializing values into array is 'Initialization'.

→ The general format is

```
arrayname[index] = value;
```

(ie) `number[0] = 15;`

(or)

```
type arrayname[] = {list of values};
int number[] = {35, 40, 20, 57, 19};
```

ASSIGNING ARRAYS:

```
int a[] = {1, 2, 3}
int b[];
b = a;
```

Array length:

```
int number = number.length;
```

Example Pgm using Arrays:

```
Class Numbersorting
{
Public static void main(String args[])
{
 int number[] = {55, 40, 80, 65, 71};
   int n = number.length;
 System.out.println("Given list: ");
 for(int i=0; i<n; i++)
 {
 System.out.println(" " +number[i]);
 }
 System.out.println("\n");
 for(int i=0; i<n; i++)
 {
  for(int j= i+1 ; j<n ; j++)
  {
   if (number[i] < number[j])
     {
        int temp = number[i];
        number[i]=number[j];
        number[j] = temp;
     }
  }
 System.out.println("Sorted list");
for(int i=0; i<n; i++)
  {
   System.out.println(" " +number[i]);
}
System.out.println(" ");
}
}
```

Two dimensional Arrays:

→ Two dimensional array creation follows same steps as creation of single dimensional arrays.

```
int myArray [] [];
myarray = new int [3][4];
```

Initialization:

→ Like 1-dimensional arrays, two dimensional arrays can be initialized as

```
int table[2][3] = {0,0,0,1,1,1};
```

(or)

```
int table[][] = {{0,0,0},{1,1,1}};
```

CONSTRUCTORS:

→ Specialised method used for initialising the objects.

→ It is special member fn, used to initialize the objects when it is created.

→ It is invoked whenever object of its associated class is created.

Characteristics of Constructor:

→ The constructor name should be same as Class name.

→ There should be not return type not even 'void'.

→ It is automatically invoked when object is created.

→ Java compiler provides default constructor if there is no constructor

→ It cannot be virtual.
→ Constructors cannot be inherited but derived invoke Constructor of base class.
→ Multiple constructor can be created in same class
→ Constructors can have default arguments.

Example: // JAVA CONSTRUCTOR PGM

```
class Rectangle
{
 int height;
  int width;
 Rectangle ()
 {
 height=10;
 width=20;
 }
 Rectangle (int h, int w)
 {
   height =h;
   width = w;
  }
 void area ()
 {
  System.out.println (" The An. called ");
   int result =height * width;
   System.out.println(" The area is" + result);
  }
}
```

```
Class Const_Demo
{
Public Static Void main (String args[])
{
 Rectangle r1= new Rectangle ();
    r1. area();
 Rectangle r2= new Rectangle (10,20);
    r2. area();
}
}
```

Exercise: write the o/p for the following code:

```
Class Student3
{
 int id;
 Strda String name;
void display ()
{
 System.out.Println (" Id " +id);
 System.out.Println ("Name " +name);
}
}
Class Student
{
Public Static Void main (String args[])
{
   Student3 s1= new Student3();
        s1.display();
}
}
```

INHERITANCE (IS-A)

-> The main purpose of inheritance is reusability.

-> It is process of creating new class from old class.

-> The new class is called as derived class (or) subclass & the old class is called as base class (or) superclass.

-> Inheritance make derived class to acquire some properties from base class.

-> Inheritance represents "IS-A relationship" which is a Parent child relationship.

General form of Inheritance:

```
Class <Subclass.name> extends Superclassname
  {
     =
  }
```

-> 'extends' keyword is used in inheritance.

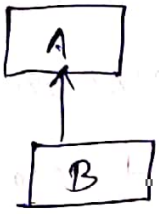
TYPES OF INHERITANCE:

-> In Java there are 3 types of inheritance 1) Single, 2) Multilevel 3) Hierarchical.

-> In Java Multiple inheritance is supported by means of interfaces only.

Single Inheritance:

→ It has one Parent and one derived class.
→ It is most common form of inheritance.



Base class / Super class

Derived class / Sub class.

Eg Prgm: → The Derived class will inherit the properties from base class.

General format:

```
Class A
{
  ==
}
Class B extends A
{
  ==
}
```

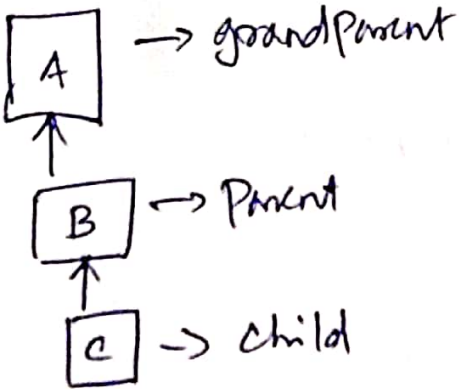
Example Prgm:

```
class A
{
  int a;
void set_a (int i)
{
  a=i;
}
```

```
void show_a()
{
System.out.println("The Value of a" + a);
}
}
class B extends A
{
 int b;
  void set_b(int i)
  {
    b=i;
 }
 void show_b()
  {
  System.out.println("The value of b" + b);
 }
 void mul()
  {
   int c;
     c = a*b;
  System.out.println("The Value of c" + c);
  }
 }
 class InhrnitDemo
{
Public static void main(String args[])
{
    A Obj_a = new A();
    B Obj_b = new B();
      Obj_b.set_a(10);
```

```
Obj_b.show_a();
Obj_b.show_b();
Obj_b.mul();
}
}
```

Multilevel Inheritance:



→ In this base class A is grandparent, from which subclass B is derived.

→ There is class C which is derived from class B.

→ In main function, we can create object of C & access any of field of class hierarchy.

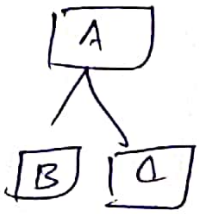
eg: Pgm

```
class A
{
int a;
void set_a(int i)
{
a = i;
}
void show_a()
{
```

```
System.out.println("The Value of a:" + a);
}
}
class B extends A
{
int b ;
void set_b(int i)
{
b= i;
}
void show_b()
{
System.out.println(" The Value of b" + b);
}
}
class C extends B
{
int c;
void set_c(int i)
{
c = i;
}
void show_c()
{
System.out.println("The Value of c" + c);
}
void mul()
{
int ans;
ans = a*b*c;
System.out.println("The value of ans" + ans)
}
class Example
{
public static void main
(String args[])
{
A obj_a = new A();
B obj_b = new B();
C obj_c = new C();
obj_c.set_a(10);
obj_c.set_b(20);
obj_c.set_b(30);
obj_c.show_a();
obj_c.show_b();
obj_c.show_c();
obj_c.mul();
}
}
```

HIERARCHICAL INHERITANCE:

--> It has one base class and many derived class.




```
Public class A
{
  ==
}
Public class B extends A
{
  ==
}
Public class C extends A
{
  ==
}
```

Exercise Pgm:

```
Class Animal
{
  Void eat()
  {
    System.out.println(" eating...");
  }
}
Class Dog extends Animal
{
  Void bark()
  {
    System.out.println(" barking");
  }
}
Class Cat extends Animal
{
  Void meow()
  {
    System.out.println(" meowing");
  }
}
Class Example
{
  Public static void main
      (String args[])
  {
    Cat c = new Cat();
    c.meow();
    c.eat();
    c.bark(); //error
  }
}
```

Method overriding:

→ Whenever same method is existing in both base class & derived class with same types of Parameters/or same order of Parameters is known as "Method overriding".

→ Method Overriding is used to provide specific implementation of a method in subclass which is already provided by its superclass.

→ It is used for runtime polymorphism.

Eg Pgm:

```
Class Animal
 {
 Void move()
 {
  System.out.println("Animals can move");
 }
}
class Dog extends Animal
  {
   Void move()
   {
  System.out.println("Dog can walk & run");
  }
}
class Example
{
Public static Void main(String args[])
```

```
{
Animal a = new Animal(); // Animal ref. & object
Animal b = new Dog(); // " " but dog object
    a.move(); // runs method in Animal class
    b.move(); // " " " Dog class.
}
}
```

Difference b/w overloading & overriding:

Overloading	Overriding
1) Used to increase readability of Program.	Used to provide specific implementation of method that is already provided by its superclass
2) Method overloading is performed within class.	It occurs in two classes (IS-A relationship) base & derived.
3) Parameters must be different	Parameters must be same including order.
4) It is example of compile time Polymorphism	Example of runtime Polymorphism
5) overloading can be done at both static & nonstatic method.	It can be done only at non-static method.
6) For overloading returntype may not be same	For overriding returntype should be same.
7) Overloading can be done for method and constructor level	Exhibited only at method level.

Q12: Overriding pgm.

```
class Animal
 {
Public void move()
{
System.out.println(" Animals can move ");
 }
}
Public class Dog extends Animal
  {
  Public void move()
    {
  System.out.println("Dogs can walk and run");
 }}
Public class TestDog
 {
Public static void main(String args[])
  {
     Animal a=new Animal();  // Animal reference type & Object
     Animal b= new Dog();   // Animal reference but Dog object
         a.move();  // runs method in Animal class
         b.move();  // run  method in Dog class.
  }}
```

→ In compile time, check is made on the reference type.

→ In run time JVM figures out Object type and call the method that belongs to Particular class for which it is created.

Super keyword:

→ It is a reference variable used to refer immediate Parent class object.

Usage of Super Keyword:

(1) Super can be used to refer immediate parent class instance variable

(2) " " " invoke immediate Parent class method.

(3) Super() can be used to invoke immediate super class constructor.

(i) Super used to refer immediate Parent class instance variable

```
class Employee
{
 float salary = 10000;
}
class HR extends Employee
{
 float salary = 20000;
 void display()
 {
  System.out.println("Salary" + super.salary); //prints base class salary
 }
}
class supervariable
{
public static void main(String args[])
 {
  HR obj = new HR();
  obj.display();
```

Super keyword at constructor level:

```
class Employee
{
  Employee()
  {
   System.out.println("Employee class constructor");
  }
}
class HR extends Employee
{
  HR()
  {
   super();
   System.out.println("HR class constructor");
  }
}
class supercons
{
public static void main(String args[])
{
  HR obj = new HR();
}
}
```

Super keyword at Method level:

-> It is used to invoke Parent Class method.

-> It is used in case of Method overriding.

eg:

```
class Student
{
 void message()
 {
  System.out.println("Good Morning Sir");
 }
}

class Faculty extends Student
 {
  void Message()
   {
    System.out.println("Goood Morning Students");
   }

  void display()
   {
    message();
    Super.message();
   }

  Public static void main(String args[])
   {
    Student s = new Student();
     s.display();
   }
```

Abstract classes:

-> A class which is declared with abstract keyword is known as "Abstract class"

-> It can have abstract and non-abstract method (method with body)

-> It needs to be extended and its method implemented.

-> It cannot be instantiated.

-> Abstraction is the process of hiding the implementation details and showing only functionality to the user. (i.e, Shows only essential details to user & hides the internal details.

eg:- If you send email, you just need to type content, address of receiver and click send, the protocol of your email server are hidden from user.

Rules of Abstract class

(1) An abstract class must be declared with an abstract keyword.

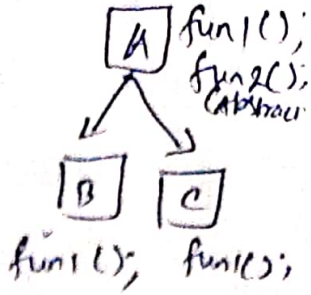
(2) It can have abstract & non abstract methods

(3) It cannot be instantiated.

(4) It can have constructors & static methods also.

(5) It can have final methods, which will force subclass not to change body of method.

Eg-Pgm




```
abstract class A
{
abstract void fun1();
void fun2()
 {
  System.out.println("A: In fun2");
}
}
class B extends A
{
void fun1()
 {
 System.out.println("B: In fun1");
}
}
class C extends A
 {
  void fun2()
  {
System.out.println("C: In fun2");
 }
}
Public class AbstractclsDemo
 {
 B b=new B();
 C c=new C();
  b.fun1();
  b.fun2();
    c.fun1();
    c.fun2(); }}
```

Abstract Methods & Abstract classes Properties:

(1) An Abstract method must be present in abstract class only It should not be in non-abstract class

(2) All the abstract methods must be implemented in non-abstract subclass

(3) Abstract class cannot be instantiated.

(4) A constructor of abstract class can be defined & invoked by subclass.

(5) A class that contain abstract method must be abstract but abstract class may not contain abstract method.

(6) A Superclass can be abstract but subclass can be concrete.

Diff. b/w Abstract class & Concrete class

Abstract class	Concrete class
1. The abstract keyword is used to define abstract class	The keyword class is used to define concrete class.
(2) It can't be instantiated.	It can be instantiated.
(3) It has Partial (or) no implementation details	It has data members & methods defined in it.
(4) It may contain abstract methods	It contains concrete methods (or) method with definition.
(5) Abstract class act as Parent class always.	It can be Parent (or) child class
(6) It must be extended.	It may (or) may not be extended

final class:

⇒ If we declare any class as final, no class can be derived from it.

Eg:

```
final class Test
{
void fun()
{
System.out.println("In In base class");
}}
class Test1 extends Test
{
final void fun()
{
System.out.print("In In Derived class");
}
}
```

Error Msg:

> 8: cannot inherit from final Test
class Test1 extends Test
^
1 error.

final Variable:

⇒ A variable can be declared as final, with keyword 'final'.
⇒ A final variable cannot be modified further. (ie, it is constant.
Eg) final int a = 10;

Final method:

→ The final keyword can also be applied to the method.
→ final method cannot be overridden.

eg:

```
class Test
{
final void fun()
{
System.out.println("Hello this is a final");
}
}
class Test1 extends Test
{
final void fun()
{
System.out.println("Another fn");
}
}
```

output:

```
>10: fun() in Test1 cannot override fun() in Test; overridden
                method is final
    final void fun()
               ^
1 error
```

METHOD OVERLOADING:

- Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**

Advantage of Method Overloading in Java:

- One significant advantage of method overriding is that a class can give its specific execution to an inherited method without having the modification in the parent class (base class).

Syntax

```
class  class_Name
{
Returntype  method()
{.....}
Returntype  method(datatype1 variable1)
{.....}
Returntype  method(datatype1 variable1, datatype2 variable2)
{.....}
Returntype  method(datatype2 variable2)
{.....}
Returntype  method(datatype2 variable2, datatype1 variable1)
{.....}
}
```

Different ways to overload the method

There are two ways to overload the method in java

- By changing number of arguments or parameters
- By changing the data type

By changing number of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

Example Method Overloading in Java

```
lass Addition
{
void sum(int a, int b)
{
```

```
System.out.println(a+b);
}
void sum(int a, int b, int c)
{
System.out.println(a+b+c);
}
public static void main(String args[])
{
Addition obj=new Addition();
obj.sum(10, 20);
obj.sum(10, 20, 30);
}
}
```

Output

```
30

60
```

By changing the data type

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

Example Method Overloading in Java

```
class Addition
{
void sum(int a, int b)
{
System.out.println(a+b);
}
void sum(float a, float b)
{
System.out.println(a+b);
}
public static void main(String args[])
{
Addition obj=new Addition();
obj.sum(10, 20);
obj.sum(10.05, 15.20);
}
```

```
}
```

Output

```
30

25.25
```

Why Method Overloading is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity.

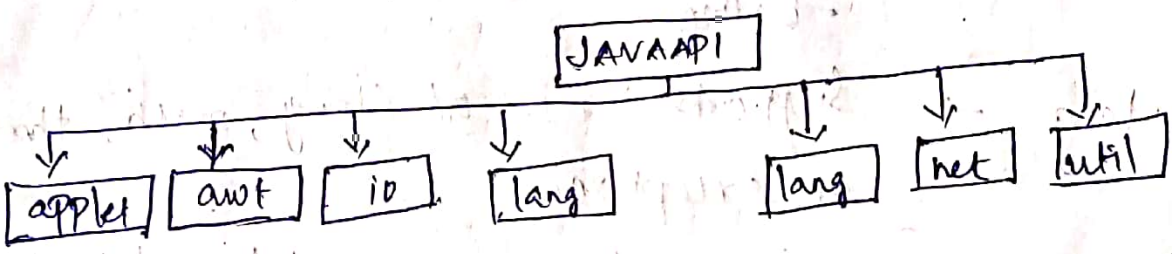
UNIT-II

PACKAGES:

→ It is the collection of similar type of classes, interfaces & subpackages

→ There are two types of Packages (1) Builtin Package (2) Userdefined Package

→ Some of the builtin Packages are



→ Packages are used to achieve code reusability (i.e) classes from other program can be used easily by a class without physically copying it to current location.

Benefits of Packages:

→ The classes defined in the package of other program can be easily reused

→ Two classes from two different packages can have same name. By using the package name, the particular class can be referred.

→ Using Packages, it is possible to hide the class. (i.e) it prevents other pgms to access the classes that are developed for internal purpose only.

→ It provide complete separation blw two phase (i) design phase & coding phase.

BUILT-IN PACKAGES:

Java.applet → Used for creating & implementing applet

java.awt → Used for graphical user interface (GUI). It can use components like button, textboxes, radio button

Java.lang → Supports the use of String, math, thread functions & exception.
It is always imported by default.

java.util → Various utility classes such as vectors, date, random numbers are used in this package.

Java.IO → Supports I/P & O/P operations.

Java.net → Support classes for networking.

Defining Package:

→ Package is the collection of similar type of classes, interfaces & subpackages

→ Packages are used to provide reusability across program.

-> There are two types of Packages

(1) Builtin Packages (2) Userdefined Packages.

Rules to Create Userdefined Package:

-> Package Stmt should be first Pgm Stmt of any Package Pgm.

-> Choose any class name (or) interface name with access specifier 'Public'.

-> Package Pgm Shouldnot contain any main Pgm.

-> Modifier of constructor which is present in class should be Public

-> Modifier of the class (or) interface which is present in the Package must be 'Public'.

-> Every Package Pgm should be saved either with Public class name (or) Public interface name.

Declaration of Package:

```
Package <Package_name>;
```

eg Package mypack;

Package Program:

```
Package mypack;
Public class A
{
 Public void show()
 {
  System.out.println("Sum method");
 }
}
```

compilation

> javac -d . A.java

Implementation of Package Pgm:

```
import mypack.A;
Public class Hello;
{
 Public static Void main(String args[])
 {
   A a = new A();
    a.show();
    System.out.println("show() class A");
 }
}
```

Interface:

→ Interface is similar to class, which is collection of (Public Static final Variable (Constants) & abstract Methods.

→ It is mechanism used to achieve fully abstraction in java.

→ There can only be abstract methods in the interface.

→ It is used to achieve fully abstraction & multiple inheritance in java.

Properties of Interface:

→ It is implicitly abstract, so we don't need to use abstract keyword when declaring interface.

→ Each method in an interface is also implicitly abstract, so abstract keyword is not needed.

→ Methods in interface are implicitly public.

→ All data members of interface are implicitly public static final.

BEHAVIOUR OF COMPILER WITH INTERFACE PLM:

```
interface Person
{
    int age = 19;
    void run();
}
```

Compile →

```
interface Person
{
    public static final
        int age=19;
    public abstract void
        run();
}
```

Declaring Interface:

→ The interface keyword is used to declare an interface.

```
Interface Person
{
 void run ();
}
class Employee implements Person
 {
 Public Void run ()
  {
  System. out. println ("Run fast");
 }
Public static Void main (String args[])
 {
```

```
Employee obj = new
        Employee();
 Obj.display();
 obj.run();
 }
```

Rules for implementing interface:

1) A class can implement more than one interface at a time.

2) A class can extend only one class, but implement many interfaces

3) An interface can extend another interface like class extending another class.

class	interface	interface
↑ extends	↑ implements	↑ extends.
class	class	interface.

Eg 3:

```
interface Printable
 {
 void print();
}
interface Showable
 {
 void show();
}
class A7 implements Printable, Showable
 {
Public void Print()
{
 System.out.println("Hello");
 }
Public void Show()
{
 System.out.println("Welcome");
}
Public static void main(String args[])
{
  A7  obj = new A7();
    Obj.print();
    Obj.show();
  }
}
```

Eg:2 Interface example:

```
interface Drawable
  {
  void draw();
}
class Rectangle implements Drawable
{
Public void draw()
{
  System.out.println("drawing Rectangle"); }
}
class Circle implements Drawable
{
Public void draw()
 {
  System.out.println("drawing circle");
 }
 class TestInterface
  {
Public static void main(String args[])
{
 Drawable d = new circle();
   d.draw();
   }
 }
```

Extending interface:

```
Interface Printable
{
 Void Print();
}
interface Showable extends Printable
{
  Void Show();
}
Class TestInterface4 implements Showable
{
 Public Void Print()
 {
  System.out.println("Hello");
}
Public Void Show()
 {
System.out.println("welcome");
}
Psvm static Void main(String args[])
 {
  TestInterface4 Obj = new TestInterface4();
       Obj.Print();
      Obj.Show();
 }}
```

NESTED INTERFACE:

→ An interface (or) declared within another interface or class is known as 'nested interface'.

→ The nested interface are used to group related interfaces so that they can be easy to maintain.

→ The nested interface must be referred by the outer interface (or) class.

→ It cannot be accessed directly

→ Nested interface must be public if it is declared inside the interface

→ It can have any access modifier if declared within the class.

→ Nested interfaces are declared static implicitly.

Syntax : Nested Interface declared within interface

```
interface interface_name
{
  ....
  interface nested_interface_name
  {...
  }
}
```

Syntax: Nested Interface declared within class:

```
class class_name
{
  interface nested_interface_name
  {
    . . .
  }
}
```

Eg: Pgm:

```
interface Showable
{
  void show();
}
interface Message
{
  void msg();
}
class TestNestedInterface implements Showable.Message
{
  public void msg()
  {
    System.out.println("Hello Nested Interface");
  }
public static void main(String args[])
  {
    Showable.Message message = new TestNestedInterface();
```

```
        message.msg();
    }
}
```

Difference Btw Abstract class & Interface:

Abstract class	Interface
(1) It is collection of abstract method & concrete method.	It is collection of abstract method
(2) It doesn't support multiple inheritance	It support multiple inheritance.
(3) It is preceded by 'abstract' keyword.	It is preceded by 'interface' keyword
(4) It contains either Variables/ constants.	It contains only constants.
(5) The class properties can be reused in other class using extends keyword.	Properties can be reused in any class using implements keyword.
(6) Inside abstract class we have constructor.	We cannot have any constructor.
(7) Initialization of Variables at the time of declaration is not mandatory.	Initialization of Variables at the time of declaration is Mandatory.
(8) Members of abstract class can have any modifier like Public, private, protected.	Members of Interfaces are Public by default.

CLASSPATH:

→ It is used by JRE to locate specific Package

→ It makes use of current working directory as its starting point.

→ If the Package is not in current working directory, you can specify the directory path setting CLASSPATH stmt.

(Ex) if package "MyPackage" is in D:\> then

```
set CLASSPATH = .;D:\;
D:\> cd MyPackage
D:\MyPackage>
```

Access Modifiers:

→ There are 4 types of Java access specifiers. They are

(1) Private (2) Default (3) Protected (4) Public.

Private: The access level of Private modifier is only within class. Cannot be accessed outside class.

Default: The access level of default modifier is only within Package. Cannot be Accessed from outside Package. By default, the access

Protected: The access level of Protected modifier is within Package & outside Package through child class. If you don't make

Child class, it cannot be accessed from outside package.

Public: The access level is everywhere. It can be accessed from within the class, outside the class, within package & outside package.

→ The non-access modifiers are static, abstract, native, synchronized, volatile, transient etc...

Access Modifier	within class	within package	outside package by subclass	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y.

Ex Program for Private access Specifier:

```
Class A
{
Private int data = 40;
Private void msg()
 {
   System.out.println("Hello java");
}
}
```

```
Public class Simple
{
Public Static Void main(String args[])
{
   A  obj = new A()
    obj.msg();   // C.T error
  System.out.println("Obj.data);
}
}
```

Eg: Pgm for default Access Specifier:

```
Package Pack;        ⇒ A.java
Class A
{
 Void msg()
{
  System.out.println("Hello");
}
```

```
Package mypack;
import Pack.*;
class B
{
Public Static Void main(String args[])
{
   A obj = new A();   // CT error
       obj.msg();     // CT error
```

Ex: Pgm for Protected Access Specifier:

```
Package Pack;          => A.java.
Public class A
 {
 Protected void msg()
{
  System.out.println("Hello");
}
}
```

```
Package mypack;
import Pack.*;          => B.java.
 class B extends A
 {
 Public static void main(String args[])
 {
    B obj=new B();                    O/p: Hello.
    obj.msg();
   }.
 }.
```

Ex. Pgm for Public Access Specifier:

```
Package pack;          => A.java,
Public class A
```

```
{
Public Void msg ()
 {
  System.out.println(" Hello");
 }
}
```

```
Package mypack;
import pack.*;

Class B
{
Public Static Void main (String args[])
{
   A obj = new A ();
      obj.msg ();
   }
}
```

STREAMS:

-> Input refers to flow of data into the program & output refers to flow of data out of a program.

-> The i/p source maybe keyboard, mouse, n/w, disk, memory

-> The o/p device maybe referred as Screen, Printer, Memory

-> Java uses the concept of streams to make i/o operation fast.

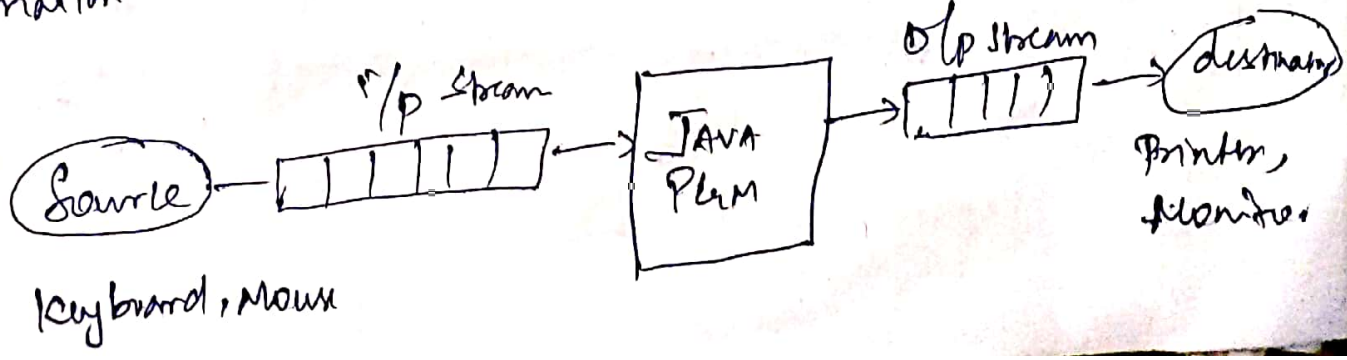
-> java.io package contains all classes required for i/p & o/p operations.

-> A stream is a sequence of data.

-> A stream is classified into (1) Input stream & (2) Output stream.

-> InputStream extracts Data from source (file) & send it to program

-> Output stream takes data from Pgm & sends (writes) to destination (file).



TYPES OF STREAMS:

→ In Java I/p & O/p operations are performed by means of streams.

→ There are two types of streams (1) Bytestream (2) Character stream.

Byte Stream:

→ Java bytestream are used to perform i/p & output of 8 bit bytes.

→ The classes associated with Input bytestream are (1) FileInputStream (2) ByteArrayInputStream (3) PipedInputStream (4) FilterInputStream (5) SequenceInputStream (6) StringBufferI/pStream.

→ The classes associated with outputstream are (1) Fileoutput Stream (2) Pipedoutput Stream (3) Filteroutputstream (4) ByteArrayoutput Stream.

→ The most commonly used input & output stream are FileInput Stream & Fileoutput Stream.

→ It uses read() & write() fns for I/O operations.

Characterstream:

→ Java Character Stream are used to perform input & output for 16 bit unicode.

(*) There are many classes associated with character stream to perform I/O operations. The most important are FileReader & FileWriter

(*) FileReader & FileWriter reads & writes two bytes at a time respectively.

(*) Two of the most important methods are read() & write().

Difference b/w bytestream & Character stream

	Byte Stream	CharacterStream.
(1)	It is used for inputting & outputting the bytes	It is used for I/P & O/P of characters.
(2)	There are 2 superclasses used in bytestream (i) InputStream (ii) ByteStream	The 2 superclasses used in CharacterStream & they are (i) Reader (ii) Writer.
(3)	A byte is 8 bit number type ranging from +127 to -127	It is 16 bit number type that represents unicode.
	Neither supports unicode characters.	It supports unicode characters.

STANDARD STREAMS:

→ All Programming lang support standard I/O where users can take input from keyboard & produce output to screen.

→ Java provides 3 standard streams

Standard input:

→ This is used to give input to user's program

→ Usually keyboard is used as standard input

→ It is represented by System.in, where System is class in java.lang.package and 'in' is the predefined stream variable

Standard output:

→ This is used to display the output generated by the user's program

→ It is commonly referred to 'computer screen'.

→ It is represented by System.out where System is class in java.lang Package & 'out' is the predefined stream variable.

Standard Error:

→ It is used to output the error produced by user's program & usually a computer program screen is used for standard error stream.

→ It is represented by System.err.

Example Program for Byte stream:

```
import java.io.*
```

```
Public class CopyFile
{
Public static void main(String args[]) throws IOException
{
  FileInputStream in = null;
  FileOutputStream out=null;
  try {
    in = new FileInputStream("input.txt");
    out= new FileOutputStream("output.txt");
    int c;
    while ((c= in.read()) !=-1)
    {
     out.write(c);
    }
     finally {
         if (in!=null)
         {
          in.close();
         }
        if (out!=null)
         {
          out.close();
         }
    }
}
```

Reading Console Input:

→ We can use Character Stream to perform console input operations (read).

→ The console i/p is referred as System.in.

→ To have Character based stream attached to the console, we should pass variable 'System.in' to BufferedReader.

```
BufferedReader br = new BufferedReader(new InputStreamReader
                                    (System.in));
```

→ InputStreamReader converts bytes to characters.

Eg:

```
import java.io.*;
Class BRRead {
Public static void main (String args[])
{
  Char c;
  BufferedReader br = new BufferedReader (new InputStreamReader
                                    (System.in));
  System.out.println (" Enter characters, 'q' to exit ");
  do
  {
    c = (char) br.read();
  System.out.println (c);
```

```
} while (c != 'q');
}
}
```

→ Here i/p operation is done by int read() throws IOException.

→ read() reads character from i/p stream & returns it as an integer value.

→ It returns -1 when end of stream is encountered.

Reading Strings:

→ To read string, we use the function readLine()

(or) String readLine() throws IOException.

```
class BRReadLines
{
public static void main(String args[]) throws IOException
{
BufferedReader br = new BufferedReader(new InputStreamReader
                                        (System.in));
String str;
System.out.println("Enter lines of text");
System.out.println("Enter stop to quit.");
do
{
  str = br.readline();
  System.out.println(str);
} while(!str.equals("stop"));
}
}
```

Writing Console output:

→ Console output can be easily done by means of Print() & Println() of PrintStream.

→ System.out is byte stream, using for simple pgm output.

→ Since PrintStream is an output stream derived from OutputStream, it also implements write(), to write to console.

Void write (int byteval)

writes the byte specified by byteval.

→ Although byteval is declared as integer only lower order eight bits are written.

Eg Pgm:

```
class writeDemo
  {
Public static Void main (String args[])
 {
   int b;
   b = 'A';
System.out.write (b);
System.out.write ("\n");
 }
}
```

OutputStream Class:

→ OutputStream class is an abstract class.
→ It is superclass of all classes representing output stream of bytes.
→ Some of the useful methods in OutputStream class are

(i) public void write(int) throws IOException
→ use to write a byte to current o/p stream

(ii) public void write(byte[]) throws IOException.
- used to write array of byte to current o/p stream

(iii) public void flush() throws IOException
- flushes the current o/p stream.

(iv) public void close() throws IOException
- used to close current output stream.

InputStream Class:

→ It is abstract class
→ It is superclass of all class representing input stream of bytes

(i) public abstract int read() throws IOException
- reads next byte of data from i/p stream. Returns -1 at end of file.

(ii) public int available() throws IOException. returns the estimate of no. of bytes that can read from current i/p stream.

Eg pgm for Character oriented I/O operation using Files:

```
import java.io.*;
Class copyCharacters
{
Public Static void main(String args[])
{
File inFile = new File("input.dat");
File outFile = new File("output.data");
FileReader ins = null;
FileWriter outs = null;
try
{
 ins = new FileReader(inFile);
 outs = new FileWriter(outFile);
 int ch;
while((ch=ins.read())!=-1)
  {
    outs.write(ch);
  }
}
catch(IOException e)
  {
    System.out.println(e);
  System.exit(-1);
```

```
finally
{
 try
 {
   ins.close();
   outs.close();
 }
 catch (IOException e)
 {
 }
}
}
```

RANDOM ACCESS FILE:

→ The RandomAccessFile class in Java enables us to read & write to a specific location at filePointer.

→ The java.io.RandomAccessFile.seek(long pos) method sets filePointer offset, which is set to beginning of this file at which next read (or) write occurs

Eg Pgm :

```
import java.io.*;
Public class RandomAccessFileDemo
{
Public static void main (String args[])
{
 RandomAccessFile file = null;
  try
  {
  file = new RandomAccessFile ("input.txt", "rw");
file.writeChar ('x');
file.writeInt (555);
file.writeDouble (3.14);
file.seek (0);
System.out.println (file.readChar());
System.out.println (file.readInt());
System.out.println (file.readDouble());
file.seek (2);
System.out.println (file.readInt());
 file.seek (file.length());
  file.writeBoolean (false);
  file.seek (4);
```

```
System.out.println (file.readBoolean ());
file.close ();
}
catch (IOException e)
{
System.out.println (e);
}
}
```

Append to End of File:

```
import java.io.*;
class RandomAccess
{
static public void main (String args[])
{
RandomAccessFile rfile;
try
{
rFile = new RandomAccessFile ("city.txt", "rw");
rFile.seek (rfile.length ());
rfile.writeBytes ("MUMBAI \n");
rfile.close ();
}
catch (IOException e)
{
System.out.println (e);
}
}
```

UNIT-3

Exception Handling

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

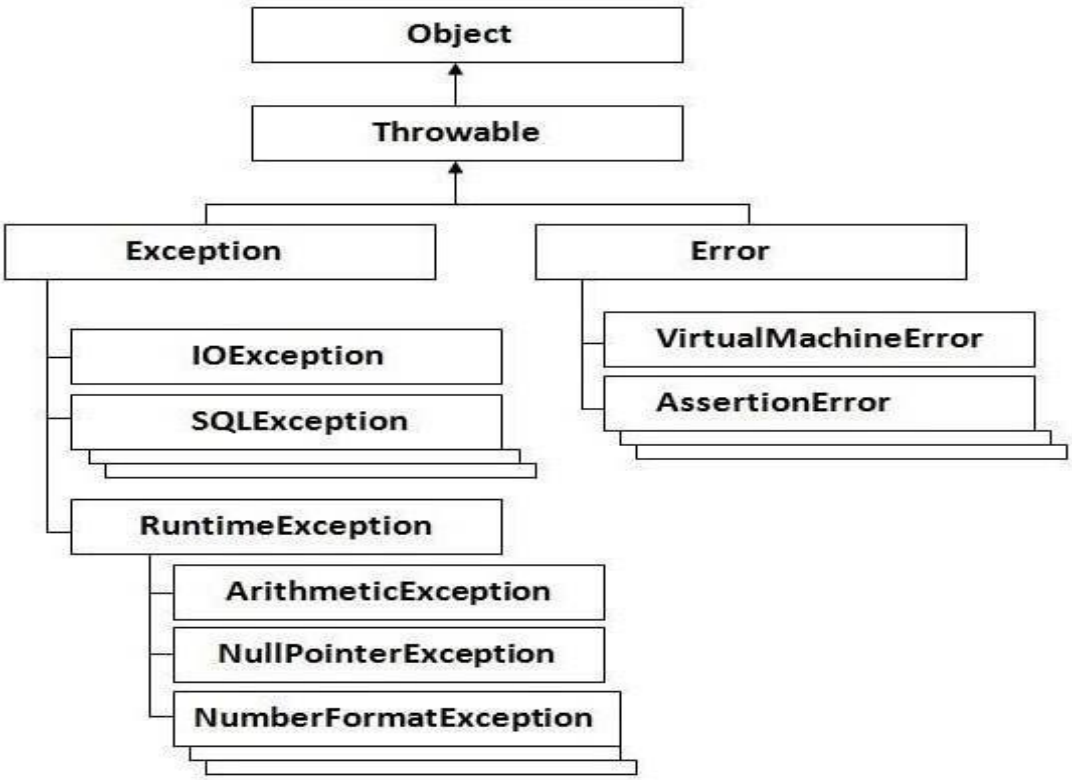
Difference between checked and unchecked exceptions

1) Checked Exception: The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception: The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error: Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionErroretc.

Hierarchy of Java Exception classes



Checked and UnChecked Exceptions

Checked Exceptions	Unchecked Exceptions
• Exception which are checked at Compile time called Checked Exception • If a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword	• Exceptions whose handling is NOT verified during Compile time. • These exceptions are handled at run-time i.e., by JVM after they occurred by using the try and catch block
• Examples: ○ IOException ○ SQLException ○ DataAccessException ○ ClassNotFoundException ○ InvocationTargetException ○ MalformedURLException	• Examples ○ NullPointerException ○ ArrayIndexOutOfBound ○ IllegalArgumentException ○ IllegalStateException

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
  public static void main(String args[]){
    int data=50/0;//may throw exception
    System.out.println("rest of the code...");
} }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
```

```
  public static void main(String args[]){
   try{
     int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
} }
```

1. Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1.  public class TestMultipleCatchBlock{
2.    public static void main(String args[]){
3.     try{
4.      int a[]=new int[5];
5.      a[5]=30/0;
6.     }
7.    catch(ArithmeticException e){System.out.println("task1 is completed");}
8.   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");
9. }
10.    catch(Exception e){System.out.println("common task completed");
11. }
12.    System.out.println("rest of the code...");
13. } }
```

```
Output:task1 completed
      rest of the code...
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
```

```
   int a[]=new int[5];
   a[5]=4;
   }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
   System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}
  System.out.println("normal flow..");
 }
1. }
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

Usage of Java finally

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:5
      finally block is always executed
      rest of the code...
```

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1{
     static void validate(int age){
     if(age<18)
     throw new ArithmeticException("not valid");
     else
     System.out.println("welcome to vote");
     }
     public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
     }  }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1.  return_type method_name() throws exception_class_name{
2.  //method code
3. }
4.
```

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
```

```
    }
    void n()throws IOException{
      m();
    }
    void p(){
     try{
      n();
     }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
     Testthrows1 obj=new Testthrows1();
     obj.p();
     System.out.println("normal flow..."); } }
```

Output:

```
exception handled
normal flow...
```

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 } }
class TestCustomException1{
  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     try{
     validate(13);
     }catch(Exception m){System.out.println("Exception occured: "+m);}

     System.out.println("rest of the code...");
  } }
```

Output:Exception occured: InvalidAgeException:not valid rest of the code...

Multithreading

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

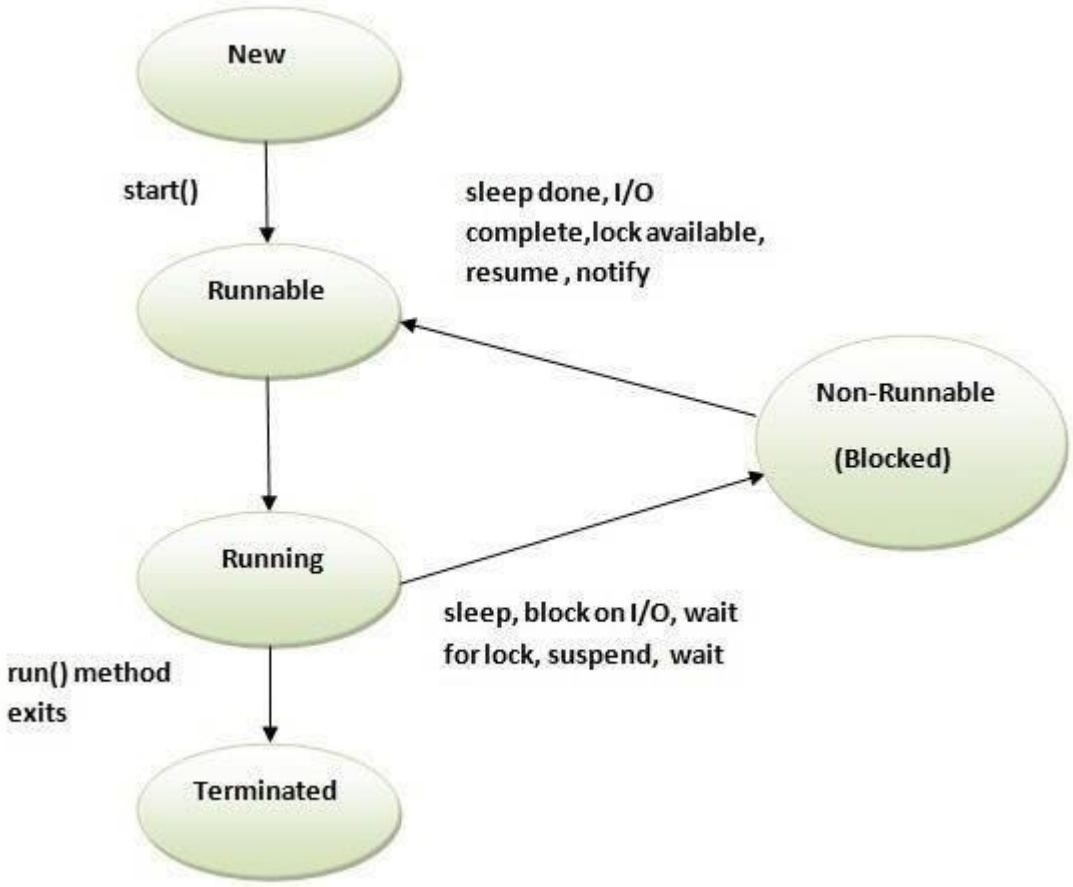
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 } }
```

Output:thread is running...

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 } }
```

Output:thread is running...

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
  }
 public static void main(String args[]){
```

```
TestMultiPriority1 m1=new TestMultiPriority1();
TestMultiPriority1 m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
m2.start();
} }
```

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
```

```
public void run(){c.deposit(10000);}
}
start();
}}
```

```
Output: going to withdraw...
        Less balance; waiting for deposit...
        going to deposit...
        deposit completed...
        withdraw completed
```

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

> ***Note: Now suspend(), resume() and stop() methods are deprecated.***

Java thread group is implemented by *java.lang.ThreadGroup* class.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)

Let's see a code to group multiple threads.

```
1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
1. Thread.currentThread().getThreadGroup().interrupt();
```

java.net

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols −

- **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects −

- **Socket Programming** − This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** − This would be covered separately.

java.text

The java.text package is necessary for every java developer to master because it has a lot of classes that is helpful in formatting such as dates, numbers, and messages.

java.text Classes

The following are the classes available

for java.text package [table]

Class|Description

SimpleDateFormat|is a concrete class that helps in formatting and

parsing of dates. [/table]

SERIALIZATION:

- *Serialization* is the process of writing the state of an object to a byte stream.
- This is useful when you want to save the state of your program to a persistent storage area, such as a file.
- At a later time, you may restore these objects by using the process of deserialization.
- Serialization is also needed to implement *Remote Method Invocation (RMI)*

'Serializable' Interface:

- Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities
- The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.
- Variables that are declared as **transient** are not saved by the serialization facilities.
- Also, **static** variables are not saved.
- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.
- The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out –

  public final void writeObject(Object x) throws IOException

  The above method serializes an Object and sends it to the output stream.
- Similarly, the ObjectInputStream class contains the following method for deserializing an object –

  public final Object readObject() throws IOException, ClassNotFoundException

  This method retrieves the next Object out of the stream and deserializes it
- The return value is Object, so you will need to cast it to its appropriate data type.

Example Program for Serialization/Deserialization:

```
public class Employee implements java.io.Serializable {
public String name;
public String address;
public transient int SSN;
public int number;

public void mailCheck() {
  System.out.println("Mailing a check to " + name + " " + address);
 }
}
```

Serializing an Object

- The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

```
import java.io.*;
public class SerializeDemo {

  public static void main(String [] args) {
    Employee e = new Employee();
    e.name = "Reyan Ali";
    e.address = "Phokka Kuan, Ambehta Peer";
    e.SSN = 11122333;
    e.number = 101;

    try {
      FileOutputStream fileOut =
      new FileOutputStream("/tmp/employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in /tmp/employee.ser");
    } catch (IOException i) {
      i.printStackTrace();
    }
  }
}
```

Deserializing an Object:

```
import java.io.*;
public class DeserializeDemo {
```

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

   System.out.println("Deserialized Employee...");
   System.out.println("Name: " + e.name);
   System.out.println("Address: " + e.address);
   System.out.println("SSN: " + e.SSN);
```

```
    System.out.println("Number: " + e.number);
   }
  }
```

Output:
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIT-4

Collection Framework in Java

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

What is framework in java

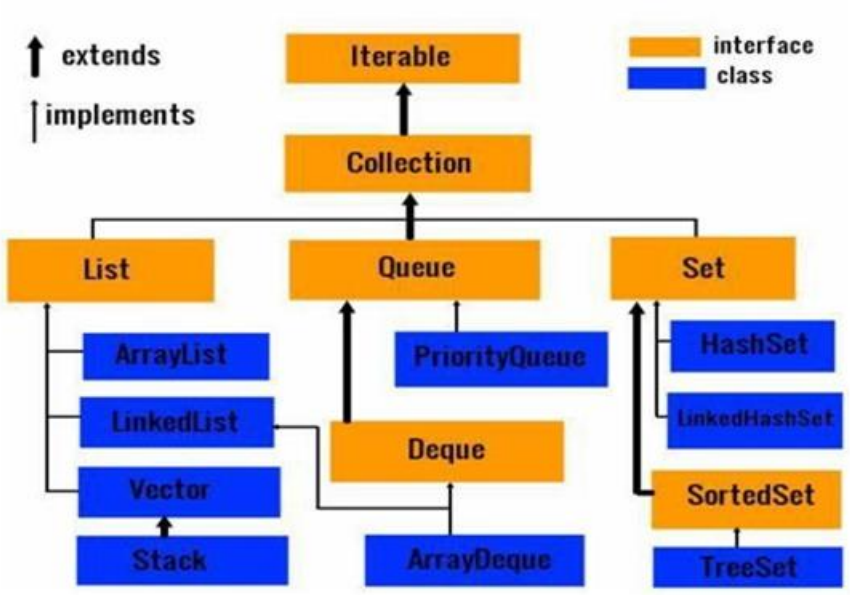
- provides readymade architecture.
- represents set of classes and interface.
- is optional.

What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

Hierarchy of Collection Framework:



Java ArrayList class

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

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList class declaration:

Let's see the declaration for java.util.ArrayList class.

Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.

ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Java ArrayList Example

```
import java.util.*;
class TestCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); }
}}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector:

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

ArrayList	Vector
1) ArrayList is not synchronized.	Vector is **synchronized**.
2)ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.

3)ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayLis tuses Iterator interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

Example of Java Vector:

Let's see a simple example of java Vector class that uses Enumeration interface.

```
1. import java.util.*;
2. class TestVector1{
3.  public static void main(String args[]){
4.   Vector<String> v=new Vector<String>();//creating vector
5.   v.add("umesh");//method of Collection
6.   v.addElement("irfan");//method of Vector
7.   v.addElement("kumar");
8.   //traversing elements using Enumeration
9.   Enumeration e=v.elements();
10.  while(e.hasMoreElements()){
11.   System.out.println(e.nextElement());
12. } } }
```

Output:

```
umesh
irfan
kumar
```

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.

- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Hashtable class declaration :

Let's see the declaration for java.util.Hashtable class

public class Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

Hashtable class Parameters

Let's see the Parameters for java.util.Hashtable class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It is the default constructor of hash table it instantiates the Hashtable class.
Hashtable(int size)	It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size.
Hashtable(int size, float fillRatio)	It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

Java Hashtable Example:

```
import java.util.*;
class TestCollection16{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,String>();
  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  } } }
```

OUTPUT:

103 Rahul

102 Ravi

101 Vijay

100 Amit

Stack

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Stack( )

Example

The following program illustrates several of the methods supported by this collection

```
import java.util.*;
public 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) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st); }
  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
```

```
showpush(st, 99);
showpop(st);
showpop(st);
showpop(st);
try {
  showpop(st);
} catch (EmptyStackException e) {
  System.out.println("empty stack");
 }}}
```

This will produce the following result −

Output

```
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
```

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.
The methods declared by Enumeration are summarized in the following table −

Sr.No.	Method & Description
1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

Example

Following is an example showing usage of Enumeration.

```
import java.util.Vector;
import java.util.Enumeration;
public class EnumerationTester {
  public static void main(String args[]) {
    Enumeration days;
    Vector dayNames = new Vector();
    dayNames.add("Sunday");
    dayNames.add("Monday");
    dayNames.add("Tuesday");
    dayNames.add("Wednesday");
    dayNames.add("Thursday");
    dayNames.add("Friday");
    dayNames.add("Saturday");
    days = dayNames.elements();
    while (days.hasMoreElements()) {
      System.out.println(days.nextElement());
    } }}
```

This will produce the following result −

Output

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Iterator

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.
Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.
Iterator object can be created by calling *iterator()* method present in Collection interface.

```
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

Iterator interface defines **three** methods:

```
// Returns true if the iteration has more elements
public boolean hasNext();

// Returns the next element in the iteration
// It throws NoSuchElementException if no more
// element present
public Object next();

// Remove the next element in the iteration
// This method can be called only once per call
// to next()

public void remove();
```

***remove()* method can throw two exceptions**

- *UnsupportedOperationException :* If the remove operation is not supported by this iterator
- *IllegalStateException :* If the next method has not yet been called, or the remove method has already been called after the last call to the next method

Limitations of Iterator:

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

Simple example of StringTokenizer class

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
```

```
StringTokenizer st = new StringTokenizer("my name is khan"," ");
 while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
} } }
```

Output:my
name
is
khan

Example of nextToken(String delim) method of StringTokenizer class

```
import java.util.*;
public class Test {
  public static void main(String[] args) {
    StringTokenizer st = new StringTokenizer("my,name,is,khan");
    // printing next token
    System.out.println("Next token is : " + st.nextToken(","));
  }    }
```

Output: Next token is : my

java.util.Random

- For using this class to generate random numbers, we have to first create an instance of this class and then invoke methods such as nextInt(), nextDouble(), nextLong() etc using that instance.
- We can generate random numbers of types integers, float, double, long, booleans using this class.
- We can pass arguments to the methods for placing an upper bound on the range of the numbers to be generated. For example, nextInt(6) will generate numbers in the range 0 to 5 both inclusive.

```
// A Java program to demonstrate random number generation
    // using java.util.Random;
    import java.util.Random;

    public class generateRandom{

       public static void main(String args[])
       {
          // create instance of Random class
          Random rand = new Random();

          // Generate random integers in range 0 to 999
          int rand_int1 = rand.nextInt(1000);
          int rand_int2 = rand.nextInt(1000);
```

```
        // Print random integers
        System.out.println("Random Integers: "+rand_int1);
        System.out.println("Random Integers: "+rand_int2);

        // Generate Random doubles
        double rand_dub1 = rand.nextDouble();
        double rand_dub2 = rand.nextDouble();

        // Print random doubles
        System.out.println("Random Doubles: "+rand_dub1);
        System.out.println("Random Doubles: "+rand_dub2);
    }}
```

Output:

```
Random Integers: 547
Random Integers: 126
Random Doubles: 0.8369779739988428
Random Doubles: 0.5497554388209912
```

Java Scanner class

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them. The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

Commonly used methods of Scanner class

There is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.

public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.

Java Scanner Example to get input from console

Let's see the simple example of the Java Scanner class which reads the int, string and double value as an input:

```
import java.util.Scanner;
class ScannerTest{
public static void main(String args[]){
 Scanner sc=new Scanner(System.in);
 System.out.println("Enter your rollno");
 int rollno=sc.nextInt();
 System.out.println("Enter your name");
 String name=sc.next();
 System.out.println("Enter your fee");
 double fee=sc.nextDouble();
 System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
 sc.close();
} } Output:
```

```
Enter your rollno
111
Enter your name
Ratan
Enter
450000
Rollno:111 name:Ratan fee:450000
```

Java Calendar Class

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

Java Calendar class declaration

Let's see the declaration of java.util.Calendar class.

1. **public abstract class** Calendar **extends** Object
2. **implements** Serializable, Cloneable, Comparable<Calendar>

Java Calendar Class Example

```
import java.util.Calendar;
public class CalendarExample1 {
  public static void main(String[] args) {
  Calendar calendar = Calendar.getInstance();
  System.out.println("The current date is : " + calendar.getTime());
  calendar.add(Calendar.DATE, -15);
  System.out.println("15 days ago: " + calendar.getTime());
  calendar.add(Calendar.MONTH, 4);
  System.out.println("4 months later: " + calendar.getTime());
  calendar.add(Calendar.YEAR, 2);
  System.out.println("2 years later: " + calendar.getTime());
  } }
```

Output:

```
The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019
```

The LinkedList Class:

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Following are the constructors supported by the LinkedList class.

Sr.No.	Constructor & Description
1	**LinkedList( )** This constructor builds an empty linked list.
2	**LinkedList(Collection c)** This constructor builds a linked list that is initialized with the elements of the collection **c**.

To add elements to the start of a list, you can use **addFirst( )** or **offerFirst( )**. To add elements to the end of the list, use **addLast( )** or **offerLast( )**. To obtain the first element, you can use **getFirst( )** or **peekFirst( )**. To obtain the last element, use **getLast( )** or **peekLast( )**. To remove the first element, use **removeFirst( )** or **pollFirst( )**. To remove the last element, use **removeLast( )** or **pollLast( )**.

```
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String args[]) {
    // Create a linked list.
    LinkedList<String> ll = new LinkedList<String>();

    // Add elements to the linked list.
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // Remove elements from the linked list.
    ll.remove("F");
    ll.remove(2);

    System.out.println("Contents of ll after deletion: "
                       + ll);
    // Remove first and last elements.
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "
                       + ll);

    // Get and set a value.

    String val = ll.get(2);
    ll.set(2, val + " Changed");

    System.out.println("ll after change: " + ll);
  }
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
 ll after deleting first and last: [A2, D, E, C]
 ll after change: [A2, D, E Changed, C]
```

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast( )**. To insert items at a specific location, use the **add(int, E)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.

Notice how the third element in **ll** is changed by employing calls to **get( )** and **set( )**. To obtain the current value of an element, pass **get( )** the index at which the element is stored. To assign a new value to that index, pass **set( )** the index and its new value.

The HashSet Class:

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

A hash table 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 transformation of the key into its hash code is performed automatically.

Following is the list of constructors provided by the HashSet class.

Sr.No.	Constructor & Description
1	**HashSet( )** This constructor constructs a default HashSet.
2	**HashSet(Collection c)** This constructor initializes the hash set by using the elements of the collection **c**.
3	**HashSet(int capacity)** This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	**HashSet(int capacity, float fillRatio)** This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

	Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.

The advantage of hashing is that it allows the execution time of **add( )**, **contains( )**, **remove( )**, and **size( )** to remain constant even for large sets.

```
import java.util.*;
public class HashSetDemo {

   public static void main(String args[]) {
      // create a hash set
      HashSet hs = new HashSet();

      // add elements to the hash set
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      System.out.println(hs);
   }
}
```

Output

```
[A, B, C, D, E, F]
```

The TreeSet Class:

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and 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.

Following is the list of the constructors supported by the TreeSet class.

Sr.No.	Constructor & Description
1	**TreeSet( )** This constructor constructs an empty tree set that will be sorted in an ascending order according to the natural order of its elements.
2	**TreeSet(Collection c)**

	This constructor builds a tree set that contains the elements of the collection **c**.
3	**TreeSet(Comparator comp)** This constructor constructs an empty tree set that will be sorted according to the given comparator.
4	**TreeSet(SortedSet ss)** This constructor builds a TreeSet that contains the elements of the given SortedSet.

```
import java.util.*;
public class TreeSetDemo {

   public static void main(String args[]) {
      // Create a tree set
      TreeSet ts = new TreeSet();

      // Add elements to the tree set
      ts.add("C");
      ts.add("A");
      ts.add("B");
      ts.add("E");
      ts.add("F");
      ts.add("D");
      System.out.println(ts);
   }
}
```

Output

```
[A, B, C, D, E, F]
```

The PriorityQueue Class:

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It creates a queue that is prioritized based on the queue's comparator.

PriorityQueue is a generic class that has this declaration: class

PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary.

PriorityQueue defines the seven constructors shown here:

PriorityQueue( ) PriorityQueue(int *capacity*)
PriorityQueue(Comparator<? super E> *comp*) PriorityQueue(int *capacity*, Comparator<? super E> *comp*)
PriorityQueue(Collection<? extends E> *c*)

PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor specifies a comparator, and the fourth builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c*. In all cases, the capacity grows automatically as elements are added.

You can obtain a reference to the comparator used by a **PriorityQueue** by calling its **comparator( )** method, shown here:

Comparator<? super E> comparator( )

The ArrayDeque Class:

The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions. (The **Deque** interface supports implementations that restrict capacity, but does not require such restrictions.) **ArrayDeque** is a generic class that has this declaration:

class ArrayDeque<E>

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

1. ArrayDeque( )
2. ArrayDeque(int *size*)
3. ArrayDeque(Collection<? extends E> *c*)

The first constructor builds an empty deque. Its starting capacity is 16.
The second constructor builds a deque that has the specified initial capacity.
The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

The following program demonstrates **ArrayDeque** by using it to create a stack:

```
// Demonstrate ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
  public static void main(String args[]) {
    // Create an array deque.
    ArrayDeque<String> adq = new ArrayDeque<String>();

    // Use an ArrayDeque like a stack.
    adq.push("A");
    adq.push("B");
    adq.push("D");
    adq.push("E");
    adq.push("F");
    System.out.print("Popping the stack: ");

    while(adq.peek() != null)
      System.out.print(adq.pop() + " ");

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

Output:

```
Popping the stack: F E D B A
```

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps −

- Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
- Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
- Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling ListIterator.

The Methods Declared by Iterator:

Sr.No.	Method & Description
1	**boolean hasNext( )** Returns true if there are more elements. Otherwise, returns false.
2	**Object next( )** Returns the next element. Throws NoSuchElementException if there is not a next element.
3	**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 Provided by **ListIterator** are**:**

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
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**.
E 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.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
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(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

The For-Each Alternative to Iterators:

Iterator is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

For eachloop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
  for (Element e: c)
      System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

When to use which traversal?

- If we have to modify collection, we can use Iterator.
- While using nested for loops it is better to use for-each loop, consider the below code for better understanding.

`

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;

class ForEachDemo {
  public static void main(String args[]) {
    // Create an array list for integers.
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // Add values to the array list.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Contents of vals: ");
    for(int v : vals)
      System.out.print(v + " ");

    System.out.println();

    // Now, sum the values by using a for loop.
    int sum = 0;
    for(int v : vals)
      sum += v;

    System.out.println("Sum of values: " + sum);
  }
}
```

The output from the program is shown here:

Contents of vals: 1 2 3 4 5 Sum of
values: 15

As you can see, the **for** loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can't modify the contents of the collection.

Comparator

Both TreeSet and TreeMap store elements in a sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

The Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order −

The compare Method

int compare(Object obj1, Object obj2)

obj1 and obj2 are the 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( ), you can alter the way that objects are ordered. For example, to sort in a reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals Method

The equals( ) method, shown here, tests whether an object equals the invoking comparator −

boolean equals(Object obj)

obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Overriding equals( ) is unnecessary, and most simple comparators will not do so.

Example:

```
import java.util.*;

class Dog implements Comparator<Dog>, Comparable<Dog> {
   private String name;
   private int age;
   Dog() {
   }

   Dog(String n, int a) {
      name = n;
      age = a;
   }

   public String getDogName() {
      return name;
   }

   public int getDogAge() {
      return age;
   }
```

```
   // Overriding the compareTo method
   public int compareTo(Dog d) {
      return (this.name).compareTo(d.name);
   }

   // Overriding the compare method to sort the age
   public int compare(Dog d, Dog d1) {
      return d.age - d1.age;
   }
}

public class Example {

   public static void main(String args[]) {
      // Takes a list o Dog objects
      List<Dog> list = new ArrayList<Dog>();

      list.add(new Dog("Shaggy", 3));
      list.add(new Dog("Lacy", 2));
      list.add(new Dog("Roger", 10));
      list.add(new Dog("Tommy", 4));
      list.add(new Dog("Tammy", 1));
      Collections.sort(list);   // Sorts the array list

      for(Dog a: list)   // printing the sorted list of names
         System.out.print(a.getDogName() + ", ");

      // Sorts the array list using comparator
      Collections.sort(list, new Dog());
      System.out.println(" ");

      for(Dog a: list)   // printing the sorted list of ages
         System.out.print(a.getDogName() +"  : "+ a.getDogAge() + ",
");
   }
}
```

Output:

```
Lacy, Roger, Shaggy, Tammy, Tommy,
Tammy  : 1, Lacy  : 2, Shaggy  : 3, Tommy  : 4, Roger  : 10,
```

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. Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

The methods defined in collection framework's algorithm are summarized in the following table –

Sr.No.	Method & Description
1	**static int binarySearch(List list, Object value, Comparator c)** Searches for value in the list ordered according to **c**. Returns the position of value in list, or -1 if value is not found.
2	**static int binarySearch(List list, Object value)** Searches for value in the list. The list must be sorted. Returns the position of value in list, or -1 if value is not found.
3	**static void copy(List list1, List list2)** Copies the elements of list2 to list1.
4	**static Enumeration enumeration(Collection c)** Returns an enumeration over **c**.
5	**static void fill(List list, Object obj)** Assigns obj to each element of the list.
6	**static int indexOfSubList(List list, List subList)** Searches list for the first occurrence of subList. Returns the index of the first match, or .1 if no match is found.
7	**static int lastIndexOfSubList(List list, List subList)** Searches list for the last occurrence of subList. Returns the index of the last match, or .1 if no match is found.
8	**static ArrayList list(Enumeration enum)** Returns an ArrayList that contains the elements of enum.
9	**static Object max(Collection c, Comparator comp)** Returns the maximum element in **c** as determined by comp.
10	**static Object max(Collection c)** Returns the maximum element in **c** as determined by natural ordering. The collection need not be sorted.
11	**static Object min(Collection c, Comparator comp)**

	Returns the minimum element in **c** as determined by comp. The collection need not be sorted.
12	**static Object min(Collection c)** Returns the minimum element in **c** as determined by natural ordering.
13	**static List nCopies(int num, Object obj)** Returns num copies of obj contained in an immutable list. num must be greater than or equal to zero.
14	**static boolean replaceAll(List list, Object old, Object new)** Replaces all occurrences of old with new in the list. Returns true if at least one replacement occurred. Returns false, otherwise.
15	**static void reverse(List list)** Reverses the sequence in list.
16	**static Comparator reverseOrder( )** Returns a reverse comparator.
17	**static void rotate(List list, int n)** Rotates list by **n** places to the right. To rotate left, use a negative value for **n**.
18	**static void shuffle(List list, Random r)** Shuffles (i.e., randomizes) the elements in the list by using **r** as a source of random numbers.
19	**static void shuffle(List list)** Shuffles (i.e., randomizes) the elements in list.
20	**static Set singleton(Object obj)** Returns obj as an immutable set. This is an easy way to convert a single object into a set.
21	**static List singletonList(Object obj)** Returns obj as an immutable list. This is an easy way to convert a single object into a list.
22	**static Map singletonMap(Object k, Object v)** Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.
23	**static void sort(List list, Comparator comp)**

	Sorts the elements of list as determined by comp.
24	**static void sort(List list)** Sorts the elements of the list as determined by their natural ordering.
25	**static void swap(List list, int idx1, int idx2)** Exchanges the elements in the list at the indices specified by idx1 and idx2.
26	**static Collection synchronizedCollection(Collection c)** Returns a thread-safe collection backed by **c**.
27	**static List synchronizedList(List list)** Returns a thread-safe list backed by list.
28	**static Map synchronizedMap(Map m)** Returns a thread-safe map backed by **m**.
29	**static Set synchronizedSet(Set s)** Returns a thread-safe set backed by **s**.
30	**static SortedMap synchronizedSortedMap(SortedMap sm)** Returns a thread-safe sorted set backed by **sm**.
31	**static SortedSet synchronizedSortedSet(SortedSet ss)** Returns a thread-safe set backed by **ss**.
32	**static Collection unmodifiableCollection(Collection c)** Returns an unmodifiable collection backed by **c**.
33	**static List unmodifiableList(List list)** Returns an unmodifiable list backed by the list.
34	**static Map unmodifiableMap(Map m)** Returns an unmodifiable map backed by **m**.
35	**static Set unmodifiableSet(Set s)** Returns an unmodifiable set backed by **s**.

36	**static SortedMap unmodifiableSortedMap(SortedMap sm)** Returns an unmodifiable sorted map backed by **sm**.
37	**static SortedSet unmodifiableSortedSet(SortedSet ss)** Returns an unmodifiable sorted set backed by **ss**.

Example:

```
import java.util.*;
public class AlgorithmsDemo {

  public static void main(String args[]) {

    // Create and initialize linked list
    LinkedList ll = new LinkedList();
    ll.add(new Integer(-8));
    ll.add(new Integer(20));
    ll.add(new Integer(-20));
    ll.add(new Integer(8));

    // Create a reverse order comparator
    Comparator r = Collections.reverseOrder();

    // Sort list by using the comparator
    Collections.sort(ll, r);

    // Get iterator
    Iterator li = ll.iterator();
    System.out.print("List sorted in reverse: ");

    while(li.hasNext()) {
      System.out.print(li.next() + " ");
    }
    System.out.println();
    Collections.shuffle(ll);

    // display randomized list
    li = ll.iterator();
    System.out.print("List shuffled: ");

    while(li.hasNext()) {
      System.out.print(li.next() + " ");
    }

    System.out.println();
    System.out.println("Minimum: " + Collections.min(ll));
    System.out.println("Maximum: " + Collections.max(ll));
  }
```

```
}
```

Output:

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

Arrays:

The **Arrays** class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by **Arrays** is examined in this section.

The **asList( )** method returns a **List** that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

static <T> List asList(T... *array*)

Here, *array* is the array that contains the data.

The **binarySearch( )** method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms. (Additional forms let you search a subrange):

static int binarySearch(byte *array*[ ], byte *value*) static int
binarySearch(char *array*[ ], char *value*) static int
binarySearch(double *array*[ ], double *value*) static int
binarySearch(float *array*[ ], float *value*) static int
binarySearch(int *array*[ ], int*value*)
static int binarySearch(long *array*[ ], long *value*) static int
binarySearch(short *array*[ ], short *value*) static int
binarySearch(Object *array*[ ], Object *value*)
static <T> int binarySearch(T[ ] *array*, T *value*, Comparator<? super T> *c*)

Here, *array* is the array to be searched, and *value* is the value to be located. The last two forms throw a **ClassCastException** if *array* contains elements that cannot be compared (for example, **Double** and **StringBuffer**) or if *value* is not compatible with the types in *array*. In the last form, the **Comparator** *c* is used to determine the order of the elements in *array*. In all cases, if *value* exists in *array*, the index of the element is returned. Otherwise, a negative value is returned.

The **copyOf( )** method returns a copy of an array and has the following forms:

static boolean[ ] copyOf(boolean[ ] *source*, int *len*) static byte[
] copyOf(byte[ ] *source*, int *len*)
static char[ ] copyOf(char[ ] *source*, int *len*) static
double[ ] copyOf(double[ ] *source*, int *len*) static float[ ]
copyOf(float[ ] *source*, int *len*) static int[ ] copyOf(int[
] *source*, int *len*)

static long[ ] copyOf(long[ ] *source*, int *len*) static
short[ ] copyOf(short[ ] *source*, int *len*) static <T>
T[ ] copyOf(T[ ] *source*, int *len*)
static <T,U> T[ ] copyOf(U[ ] *source*, int *len*, Class<? extends T[ ]> *resultT*)

The original array is specified by *source*, and the length of the copy is specified by *len*. If the copy is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). If the copy is shorter than *source*, then the copy is truncated. In the last form, the type of *resultT* becomes the type of the array returned. If *len* is negative, a **NegativeArraySizeException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **copyOfRange( )** method returns a copy of a range within an array and has the following forms:

static boolean[ ] copyOfRange(boolean[ ] *source*, int *start*, int *end*)
static byte[ ] copyOfRange(byte[ ] *source*, int *start*, int *end*)
static char[ ] copyOfRange(char[ ] *source*, int *start*, int *end*)
static double[ ] copyOfRange(double[ ] *source*, int *start*, int *end*)
static float[ ] copyOfRange(float[ ] *source*, int *start*, int *end*)
static int[ ] copyOfRange(int[ ] *source*, int *start*, int *end*)
static long[ ] copyOfRange(long[ ] *source*, int *start*, int *end*)
static short[ ] copyOfRange(short[ ] *source*, int *start*, int *end*)
static <T> T[ ] copyOfRange(T[ ] *source*, int *start*, int *end*)
static <T,U> T[ ] copyOfRange(U[ ] *source*, int *start*, int *end*,
Class<? extends T[ ]> *resultT*)

The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*. The range runs from *start* to *end* – 1. If the range is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). In the last form, the type of *resultT* becomes the type of the array returned. If *start* is negative or greater than the length of *source*, an **ArrayIndexOutOfBoundsException** is thrown. If *start* is greater than *end*, an **IllegalArgumentException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **equals( )** method returns **true** if two arrays are equivalent. Otherwise, it returns **false**. Here are a number of its forms. Several more versions are available that let you specify a range and/or a comparator.

static boolean equals(boolean *array1*[ ], boolean *array2* [ ]) static
boolean equals(byte *array1*[ ], byte *array2* [ ])
static boolean equals(char *array1*[ ], char *array2* [ ]) static
boolean equals(double *array1*[ ], double *array2* [ ]) static boolean
equals(float *array1*[ ], float *array2* [ ]) static boolean equals(int
array1[ ], int *array2* [ ])
static boolean equals(long *array1*[ ], long *array2* [ ]) static
boolean equals(short *array1*[ ], short *array2* [ ]) static boolean

equals(Object *array1*[ ], Object *array2* [ ])

Here, *array1* and *array2* are the two arrays that are compared for equality.

The **deepEquals( )** method can be used to determine if two arrays, which might contain nested arrays, are equal. It has this declaration: static

boolean deepEquals(Object[ ] *a*, Object[ ] *b*)

It returns **true** if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked. It returns **false** if the arrays, or any nested arrays, differ.

The **fill( )** method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The **fill( )** method has two versions. The first version, which has the following forms, fills an entire array:

static void fill(boolean *array*[ ], boolean *value*) static
void fill(byte *array*[ ], byte*value*)
static void fill(char *array*[ ], char *value*) static void
fill(double *array*[ ], double *value*) static void
fill(float *array*[ ], float *value*) static void fill(int
array[ ], int*value*)
static void fill(long *array*[ ], long *value*) static void
fill(short *array*[ ], short *value*) static void
fill(Object *array*[ ], Object *value*)

Here, *value* is assigned to all elements in *array*. The second version of the **fill( )** method assigns a value to a subset of an array.

The **sort( )** method sorts an array so that it is arranged in ascending order. The **sort( )** method has two versions. The first version, shown here, sorts the entire array:

static void sort(byte *array*[ ]) static
void sort(char *array*[ ]) static void
sort(double *array*[ ]) static void
sort(float *array*[ ]) static void
sort(int *array*[ ]) static void sort(long
array[ ]) static void sort(short *array*[
]) static void sort(Object *array*[ ])
static <T> void sort(T *array*[ ], Comparator<? super T> *c*)

Here, *array* is the array to be sorted. In the last form, *c* is a **Comparator** that is used to order the elements of *array*. The last two forms can throw a

ClassCastException if elements of the array being sorted are not comparable. The second version of **sort( )** enables you to specify a range within an array that you want to sort.

One quite powerful method in **Arrays** is **parallelSort( )** because it sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. Like **sort( )**, there are two basic types of **parallelSort( )**, each with several overloads. The first type sorts the entire array. It is shown here:

static void parallelSort(byte *array*[ ]) static
void parallelSort(char *array*[ ]) static void
parallelSort(double *array*[ ]) static void
parallelSort(float *array*[ ]) static void
parallelSort(int *array*[ ]) static void
parallelSort(long *array*[ ]) static void
parallelSort(short *array*[ ])
static <T extends Comparable<? super T>> void parallelSort(T *array*[ ]) static <T> void
parallelSort(T *array*[ ], Comparator<? super T> *c*)

Here, *array* is the array to be sorted. In the last form, *c* is a comparator that is used to order the elements in the array. The last two forms can throw a **ClassCastException** if the elements of the array being sorted are not comparable. The second version of **parallelSort( )** enables you to specify a range within the array that you want to sort.

Arrays supports spliterators by including the **spliterator( )** method. It has two basic forms. The first type returns a spliterator to an entire array. It is shown here:

static Spliterator.OfDouble spliterator(double *array*[ ]) static
Spliterator.OfInt spliterator(int *array*[ ])
static Spliterator.OfLong spliterator(long *array*[ ]) static
<T> Spliterator spliterator(T *array*[ ])

Here, *array* is the array that the spliterator will cycle through. The second version of **spliterator( )** enables you to specify a range to iterate within the array.

Arrays supports the **Stream** interface by including the **stream( )** method. It has two forms. The first is shown here:

static DoubleStream stream(double *array*[ ]) static
IntStream stream(int *array*[ ])
static LongStream stream(long *array*[ ]) static
<T> Stream stream(T *array*[ ])

Here, *array* is the array to which the stream will refer. The second version of **stream( )** enables you to specify a range within the array.

Another two methods are related: **setAll( )** and **parallelSetAll( )**. Both assign values to all of the elements, but **parallelSetAll( )** works in parallel. Here is an example of each:

static void setAll(double *array*[ ],
IntToDoubleFunction<? extends T> *genVal*)

static void parallelSetAll(double *array*[ ],
IntToDoubleFunction<? extends T> *genVal*)

Several overloads exist for each of these that handle types **int**, **long**, and generic.

One of the more intriguing methods defined by **Arrays** is called **parallelPrefix( )**, and it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values. It has several overloads. Here is one example:

static void parallelPrefix(double *array*[ ], DoubleBinaryOperator *func*)

Here, *array* is the array being acted upon, and *func* specifies the operation applied. (**DoubleBinaryOperator** is a functional interface defined in **java.util.function**.) Many other versions are provided, including those that operate on types **int**, **long**, and generic, and those that let you specify a range within the array on which to operate.

JDK 9 added three comparison methods to **Arrays**. They are **compare( )**, **compareUnsigned( )**, and **mismatch( )**. Each has several overloads and each has versions that let you define a range to compare. Here is a brief description of each. The **compare( )** method compares two arrays. It returns zero if they are the same, a positive value if the first array is greater than the second, and negative if the first array is less than the second. To perform an unsigned comparison of two arrays that hold integer values, use **compareUnsigned( )**. To find the location of

the first mismatch between two arrays, use **mismatch( )**. It returns the index of the mismatch, or −1 if the arrays are equivalent.

Arrays also provides **toString( )** and **hashCode( )** for the various types of arrays. In addition, **deepToString( )** and **deepHashCode( )** are provided, which operate effectively on arrays that contain nested arrays.

The following program illustrates how to use some of the methods of the **Arrays** class:

```
// Demonstrate Arrays
import java.util.*;

class ArraysDemo {
  public static void main(String args[]) {

    // Allocate and initialize array.
    int array[] = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // Display, sort, and display the array.
    System.out.print("Original contents: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Sorted: ");
    display(array);

    // Fill and display the array.
    Arrays.fill(array, 2, 6, -1);
    System.out.print("After fill(): ");
    display(array);

    // Sort and display the array.
    Arrays.sort(array);
    System.out.print("After sorting again: ");
    display(array);

    // Binary search for -9.
    System.out.print("The value -9 is at location ");
    int index =
      Arrays.binarySearch(array, -9);

    System.out.println(index);
  }
    static void display(int array[]) {
      for(int i: array)
        System.out.print(i + " ");

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

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1
0 The value -9 is at location 2
```

Dictionary Class

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you 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 currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is discussed here.It is declared as shown here:

class Dictionary<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in

Method	Purpose
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

```
// Java Program explaining util.Dictionary class Methods
// put(), elements(), get(), isEmpty(), keys()
// remove(), size()

import java.util.*;
public class New_Class
{
	public static void main(String[] args)
	{

		// Initializing a Dictionary
		Dictionary geek = new Hashtable();

		// put() method
		geek.put("123", "Code");
		geek.put("456", "Program");

		// elements() method :
		for (Enumeration i = geek.elements(); i.hasMoreElements();)
		{
			System.out.println("Value in Dictionary : " + i.nextElement());
		}

		// get() method :
		System.out.println("\nValue at key = 6 : " + geek.get("6"));
		System.out.println("Value at key = 456 : " + geek.get("123"));

		// isEmpty() method :
		System.out.println("\nThere is no key-value pair : " + geek.isEmpty() + "\n");

		// keys() method :
		for (Enumeration k = geek.keys(); k.hasMoreElements();)
		{
			System.out.println("Keys in Dictionary : " + k.nextElement());
		}

		// remove() method :
		System.out.println("\nRemove : " + geek.remove("123"));
		System.out.println("Check the value of removed key : " + geek.get("123"));

		System.out.println("\nSize of Dictionary : " + geek.size());

	}
}
```

Output:

```
Value in Dictionary : Code
Value in Dictionary : Program

Value at key = 6 : null
Value at key = 456 : Code

There is no key-value pair : false

Keys in Dictionary : 123
Keys in Dictionary : 456

Remove : Code
Check the value of removed key : null

Size of Dictionary : 1
```

BitSet:

A **BitSet** class creates a special type of array that holds bit values in the form of **boolean** values. This array can increase in size as needed. This makes it similar to a vector of bits. The **BitSet** constructors are shown here:

BitSet( ) BitSet(int
size)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to **false.**

BitSet defines the methods listed in

Method	Description
void and(BitSet *bitSet*)	ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object.
void andNot(BitSet *bitSet*)	For each set bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void clear( )	Zeros all bits.
void clear(int *index*)	Zeros the bit specified by *index*.
void clear(int *startIndex*, int *endIndex*)	Zeros the bits from *startIndex* to *endIndex*–1.
Object clone( )	Duplicates the invoking **BitSet** object.
boolean equals(Object *bitSet*)	Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**.
void flip(int *index*)	Reverses the bit specified by *index*.
void flip(int *startIndex*, int *endIndex*)	Reverses the bits from *startIndex* to *endIndex*–1.
boolean get(int *index*)	Returns the current state of the bit at the specified index.
BitSet get(int *startIndex*, int *endIndex*)	Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed.
int hashCode( )	Returns the hash code for the invoking object.
boolean intersects(BitSet *bitSet*)	Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are set.

boolean isEmpty( )	Returns **true** if all bits in the invoking object are cleared.
int length( )	Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last set bit.
int nextClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit), starting from the index specified by *startIndex*.
int nextSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned.
void or(BitSet *bitSet*)	ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.
int previousClearBit(int *startIndex*)	Returns the index of the next cleared bit (that is, the next **false** bit) at or prior to the index specified by *startIndex*. If no cleared bit is found, –1 is returned.
int previousSetBit(int *startIndex*)	Returns the index of the next set bit (that is, the next **true** bit) at or prior to the index specified by *startIndex*. If no set bit is found, –1 is returned.
void set(int *index*)	Sets the bit specified by *index*.
void set(int *index*, boolean *v*)	Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit; **false** clears the bit.
void set(int *startIndex*, int *endIndex*)	Sets the bits from *startIndex* to *endIndex*–1.
void set(int *startIndex*, int *endIndex*, boolean *v*)	Sets the bits from *startIndex* to *endIndex*–1 to the value passed in *v*. **true** sets the bits; **false** clears the bits.
int size( )	Returns the number of bits in the invoking **BitSet** object.
IntStream stream( )	Returns a stream that contains the bit positions, from low to high, that have set bits.
byte[ ] toByteArray( )	Returns a **byte** array that contains the invoking **BitSet** object.
long[ ] toLongArray( )	Returns a **long** array that contains the invoking **BitSet** object.
String toString( )	Returns the string equivalent of the invoking **BitSet** object.
static BitSet valueOf(byte[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(ByteBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(long[ ] *v*)	Returns a **BitSet** that contains the bits in *v*.
static BitSet valueOf(LongBuffer *v*)	Returns a **BitSet** that contains the bits in *v*.
void xor(BitSet *bitSet*)	XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object.

```
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);

    // AND bits
    bits2.and(bits1);
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

The output from this program is shown here. When **toString( )** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position.

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
```

```
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Date:

The **Date** class encapsulates the current date and time.

Date supports the following non-deprecated constructors:

Date( )
Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.. **Date** also implements the **Comparable** interface.

Method	Description
boolean after(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**.
boolean before(Date *date*)	Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**.
Object clone( )	Duplicates the invoking **Date** object.
int compareTo(Date *date*)	Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*.
boolean equals(Object *date*)	Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**.
static Date from(Instant t)	Returns a **Date** object corresponding to the **Instant** object passed in *t*.
long getTime( )	Returns the number of milliseconds that have elapsed since January 1, 1970.
int hashCode( )	Returns a hash code for the invoking object.
void setTime(long *time*)	Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
Instant toInstant( )	Returns an **Instant** object corresponding to the invoking **Date** object.
String toString( )	Converts the invoking **Date** object into a string and returns the result.

```
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println(date);

    // Display number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
  }
}
```

Sample output is shown here:

```
Mon Jan 01 10:52:44 CST 2018
Milliseconds since Jan. 1, 1970 GMT = 1514825564360
```

Formatter:

At the core of Java's support for creating formatted output is the **Formatter** class. It provides *format conversions* that let you display numbers, strings, and time and date in virtually any format you like.

The Formatter Constructors:
You can use **Formatter** to format output, you must create a **Formatter** object. In general, **Formatter** works by converting the binary form of data used by a program into formatted text.

The **Formatter** class defines many constructors, which enable you to construct a **Formatter** in a variety of ways. Here is a sampling:
Formatter( ) Formatter(Appendable *buf*)
Formatter(Appendable *buf*, Locale *loc*)
Formatter(String *filename*)throws FileNotFoundException
Formatter(String *filename*, String *charset*)throws FileNotFoundException, UnsupportedEncodingExceptio
Formatter(File *outF*) throws FileNotFoundException
Formatter(OutputStream *outStrm*)

Here, *buf* specifies a buffer for the formatted output. If *buf* is null, then **Formatter** automatically allocates a **StringBuilder** to hold the formatted output. The *loc* parameter specifies a locale. If no locale is specified, the default locale is used. The *filename* parameter specifies the name of a file that will receive the formatted output. The *charset* parameter specifies the character set. If no character set is specified, then the default character set is used. The *outF* parameter specifies a reference to an open file that will receive output. The *outStrm* parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a **StringBuilder** to hold the formatted output

The Formatter Methods:

Method	Description
void close( )	Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**.
void flush( )	Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file.
Formatter format(String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object.
Formatter format(Locale *loc*, String *fmtString*, Object ... *args*)	Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object.
IOException ioException( )	If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, **null** is returned.
Locale locale( )	Returns the invoking object's locale.
Appendable out( )	Returns a reference to the underlying object that is the destination for output.
String toString( )	Returns a **String** containing the formatted output.

Formatting Basics:

After you have created a **Formatter**, you can use it to create a formatted string. To do so, use the **format( )** method. The version we will use is shown here:

Formatter format(String *fmtString*, Object ... *args*)

The *fmtSring* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains *format specifiers* that define the way the subsequent arguments are displayed.

```
Formatter fmt = new Formatter();
fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

This sequence creates a **Formatter** that contains the following string:

Formatting with Java is easy 10 98.600000

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c %C	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point
%g %G	Uses **%e** or **%f**, based on the value being formatted and the precision
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a % sign

UNIT-5

GUI Programming with Swing

The AWT Class hierarchy

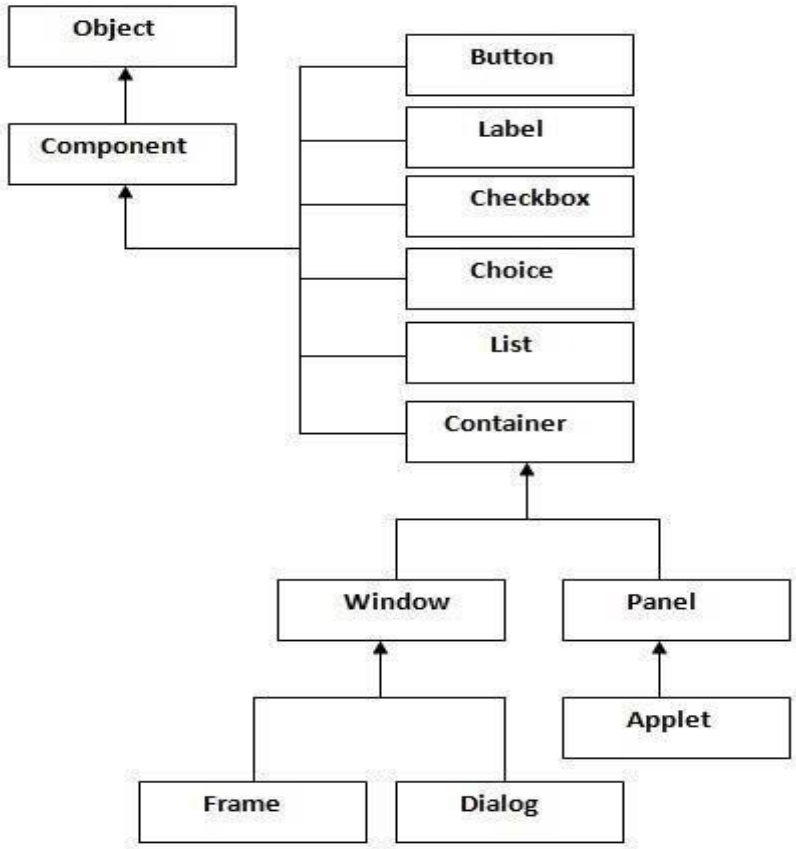
Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

Java Swing

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window- based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing.

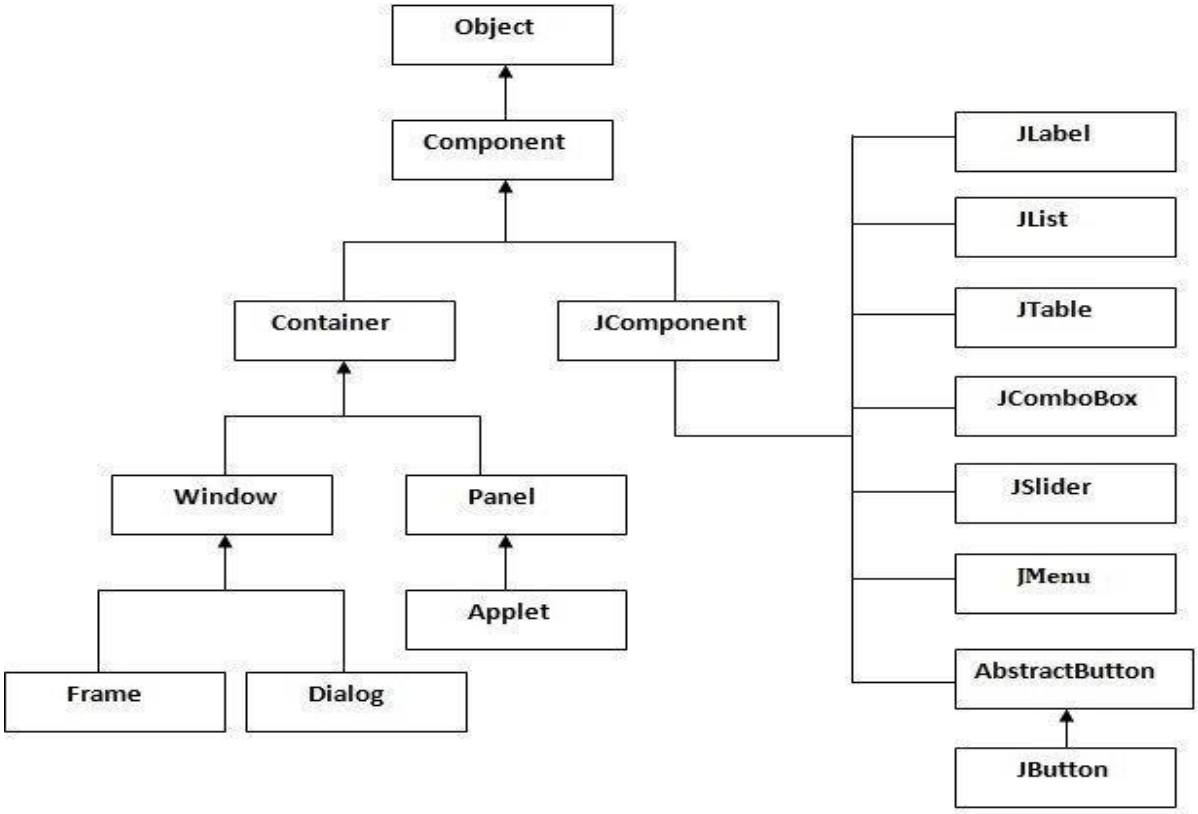
No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components**such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Java Swing Examples

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

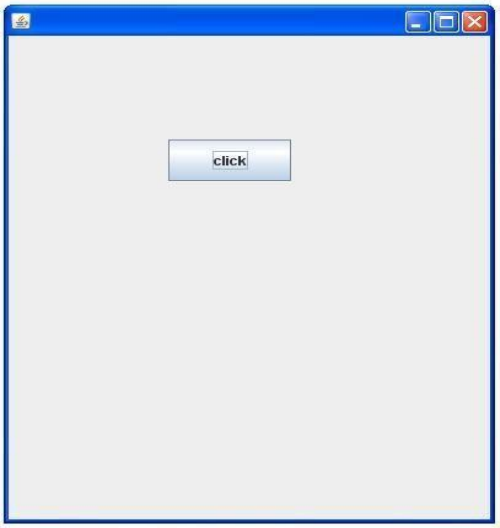
We can write the code of swing inside the main(), constructor or any other method.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
} }
```



Containers

Java JFrame

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

JFrame Example

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.Jpanel;
public class JFrameExample {
public static void main(String s[]) {
JFrame frame = new JFrame("JFrame Example");
JPanel panel = new JPanel();
panel.setLayout(new FlowLayout());
JLabel label = new JLabel("JFrame By Example");
JButton button = new JButton();
button.setText("Button");
panel.add(label);
```

```
panel.add(button);
frame.add(panel);
frame.setSize(200, 300);
frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
} }
```

JApplet

As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class.

Example of EventHandling in JApplet:

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
add(b);add(tf);
b.addActionListener(this);
setLayout(null);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
} }
```

In the above example, we have created all the controls in init() method because it is invoked only once.

myapplet.html

```
1. <html>
2. <body>
3. <applet code="EventJApplet.class" width="300" height="300">
```

</applet>
</body>
</html>

JDialog

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.

Unlike JFrame, it doesn't have maximize and minimize buttons.

JDialog class declaration

Let's see the declaration for javax.swing.JDialog class.

1. **public class** JDialog **extends** Dialog **implements** WindowConstants, Accessible, RootPaneContainer

Commonly used Constructors:

Constructor	Description
JDialog()	It is used to create a modeless dialog without a title and without a specified Frame owner.
JDialog(Frame owner)	It is used to create a modeless dialog with specified Frame as its owner and an empty title.
JDialog(Frame owner, String title, boolean modal)	It is used to create a dialog with the specified title, owner Frame and modality.

Java JDialog Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
private static JDialog d;
DialogExample() {
JFrame f= new JFrame();
d = new JDialog(f , "Dialog Example", true);
d.setLayout( new FlowLayout() );
JButton b = new JButton ("OK");
b.addActionListener ( new ActionListener()
{
public void actionPerformed( ActionEvent e )
{
        DialogExample.d.setVisible(false);
}

});

d.add( new JLabel ("Click button to continue."));
d.add(b);
d.setSize(300,300);
d.setVisible(true);
}
public static void main(String args[])
{
new DialogExample();
} }
```

Output:

JPanel

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

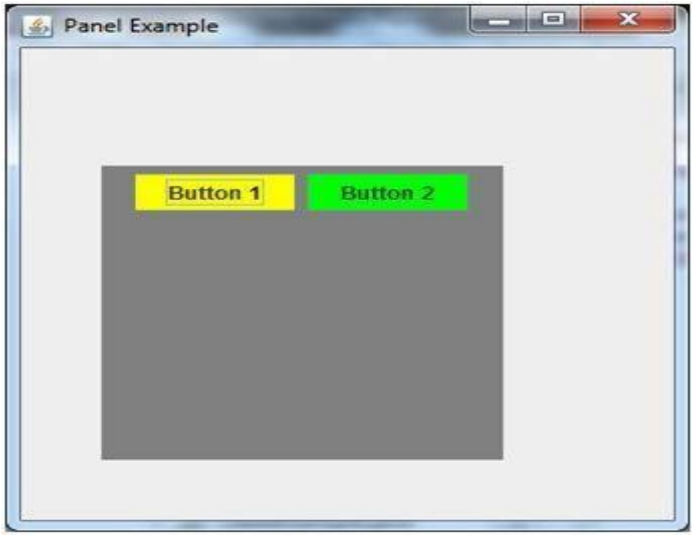
It doesn't have title bar.

JPanel class declaration

1. **public class** JPanel **extends** JComponent **implements** Accessible

Java JPanel Example

```
import java.awt.*;
import javax.swing.*;
public class PanelExample {
    PanelExample()
      {
      JFrame f= new JFrame("Panel Example");
      JPanel panel=new JPanel();
      panel.setBounds(40,80,200,200);
      panel.setBackground(Color.gray);
      JButton b1=new JButton("Button 1");
      b1.setBounds(50,100,80,30);
      b1.setBackground(Color.yellow);
      JButton b2=new JButton("Button 2");
      b2.setBounds(100,100,80,30);
      b2.setBackground(Color.green);
      panel.add(b1); panel.add(b2);
      f.add(panel);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
      }
      public static void main(String args[])
      {
      new PanelExample();
      } }
```



Overview of some Swing Components

Java JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

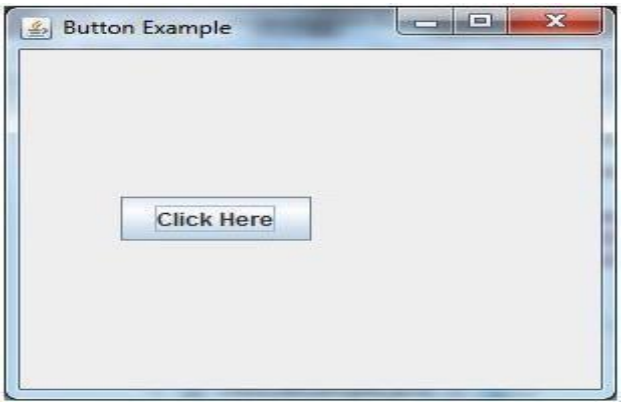
JButton class declaration

Let's see the declaration for javax.swing.JButton class.

1. **public class** JButton **extends** AbstractButton **implements** Accessible

Java JButton Example

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
    JFrame f=new JFrame("Button Example");
    JButton b=new JButton("Click Here");
    b.setBounds(50,100,95,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true); } }
```



Java JLabel

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

JLabel class declaration

Let's see the declaration for javax.swing.JLabel class.

1. **public class** JLabel **extends** JComponent **implements** SwingConstants, Accessible

Commonly used Constructors:

Constructor	Description
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.

Commonly used Methods:

Methods	Description
String getText()	t returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.
Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

Java JLabel Example

```
import javax.swing.*;
class LabelExample
{
public static void main(String args[])
   {
   JFrame f= new JFrame("Label Example");
   JLabel l1,l2;
   l1=new JLabel("First Label.");
   l1.setBounds(50,50, 100,30);
   l2=new JLabel("Second Label.");
   l2.setBounds(50,100, 100,30);
   f.add(l1); f.add(l2);
   f.setSize(300,300);
   f.setLayout(null);
   f.setVisible(true);
   }
   }
```

JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

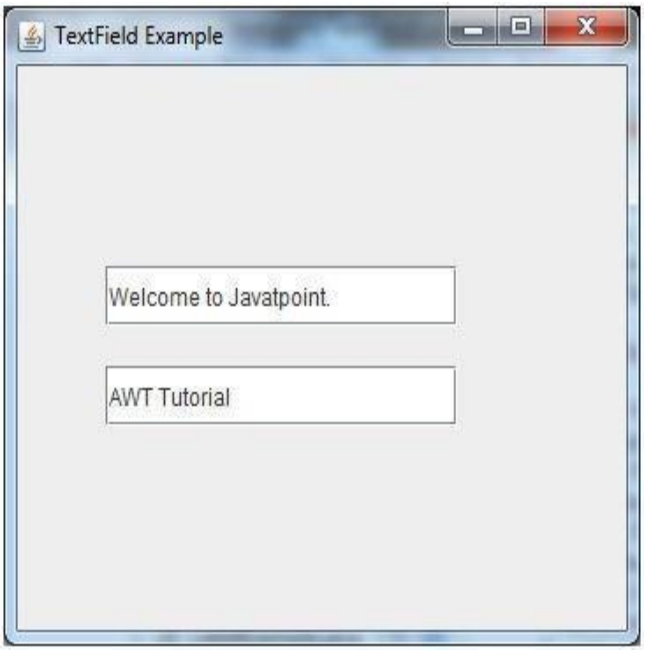
JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

1. **public class** JTextField **extends** JTextComponent **implements** SwingConstants

Java JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
    {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.s etLayout(null);
    f.setVisible(true);
    }    }
```



Java JTextArea

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

JTextArea class declaration

Let's see the declaration for javax.swing.JTextArea class.

1. **public class** JTextArea **extends** JTextComponent

Java JTextArea Example

```
import javax.swing.*;
public class TextAreaExample
{
    TextAreaExample(){
       JFrame f= new JFrame();
       JTextArea area=new JTextArea("Welcome to javatpoint");
       area.setBounds(10,30, 200,200);
       f.add(area);
       f.setSize(300,300);
       f.setLayout(null);
       f.setVisible(true);
    }
public static void main(String args[])
   {
  new TextAreaExample();
   }}
```

Simple Java Applications

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Example extends JFrame {

   public Example() {
      setTitle("Simple example");
      setSize(300, 200);
      setLocationRelativeTo(null);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   public static void main(String[] args) {
      Example ex = new Example();
      ex.setVisible(true);
   }}
```

Layout Management

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

BorderLayout

The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class:

```
import java.awt.*;
import javax.swing.*;
public class Border
 {
JFrame f;
Border()
{
   f=new JFrame();
   JButton b1=new JButton("NORTH");;
   JButton b2=new JButton("SOUTH");;
   JButton b3=new JButton("EAST");;
   JButton b4=new JButton("WEST");;
   JButton b5=new JButton("CENTER");;
   f.add(b1,BorderLayout.NORTH);
   f.add(b2,BorderLayout.SOUTH);
   f.add(b3,BorderLayout.EAST);
   f.add(b4,BorderLayout.WEST);
   f.add(b5,BorderLayout.CENTER);
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args)
{
  new Border();
} }
```

Output:

GridBag Layout

A **GridBagLayout** is a layout manager that is very flexible where the components can be aligned vertically, horizontally or on the baseline. In the **GridLayout** all the components are adjusted to occupy all the space provided to them.

The components in a **GridBagLayout** also occupy a rectangular grid of cells, wherein a component can occupy one or more than one cells. This grid of cells occupied by the component is called the display area. A component in a **GridBagLayout** has a set of constraints called the **GridBagConstraints** that need to be followed which specifies the component's minimum and preferred sizes.

Constructors

public GridBagLayout()

Methods

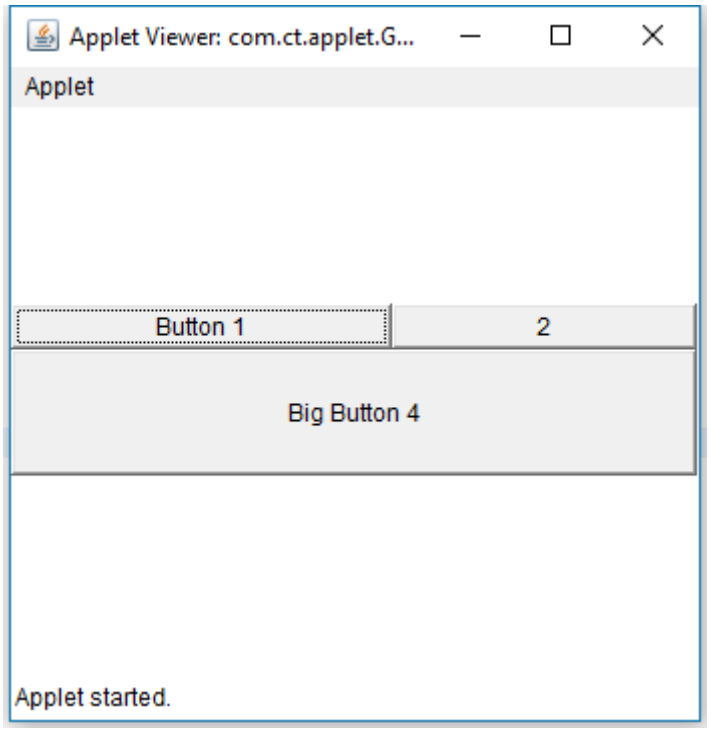
public void setConstraints(Component comp,GridBagConstraints constraints)
public GridBagConstraints getConstraints(Component comp)
public void addLayoutManager(Component comp,Object constraints)
public void removeLayoutManager(Component comp)
public Dimension preferredLayoutSize(Container parent)
public Dimension minimumLayoutSize(Container parent)
public Dimension maximumLayoutSize(Container parent)

EXAMPLE PGM:

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.GridBagConstraints;
    import java.awt.GridBagLayout;
public class GridBagLayoutExample extends Applet {
    Button button;

    public void init() {
        // Declaringand initializing gridbag layout and constraints
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();
        this.setLayout(gridbag);
        // The constraints constants are assigned values
        c.fill = GridBagConstraints.HORIZONTAL;
        // A button is created and constraints added
        button = new Button("Button 1");
        c.weightx = 0.5;
        c.gridx = 0;
        c.gridy = 0;
        gridbag.setConstraints(button, c);
        this.add(button);
        // A button is created and constraints added
        button = new Button("2");
        c.gridx = 1;
        c.gridy = 0;
        gridbag.setConstraints(button, c);
        this.add(button);
        // A Big button is created and constraints added
        button = new Button("Big Button 4");
        c.ipady = 40; // This is a long button
        c.weightx = 0.0;
        c.gridwidth = 3;
        c.gridx = 0;
        c.gridy = 1;
```

```
        gridbag.setConstraints(button, c);
        this.add(button);
    }}
```

OUTPUT:



Java GridLayout

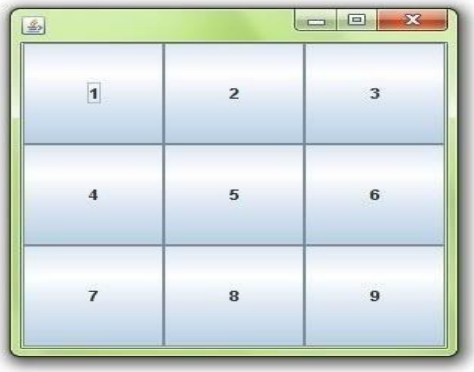
The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class

```
1. import java.awt.*;
2. import javax.swing.*;
public class MyGridLayout{
JFrame f;
MyGridLayout(){
   f=new JFrame();
   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
   JButton b6=new JButton("6");
   JButton b7=new JButton("7");
   JButton b8=new JButton("8");
   JButton b9=new JButton("9");
   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
   f.add(b6);f.add(b7);f.add(b8);f.add(b9);
   f.setLayout(new GridLayout(3,3));
   //setting grid layout of 3 rows and 3 columns
   f.setSize(300,300);
   f.s etVisible(true);
}
public static void main(String[] args) {
 new MyGridLayout(); }}
```



Java FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Fields of FlowLayout class

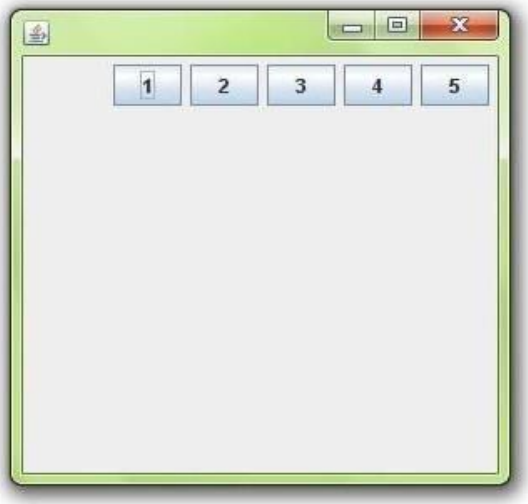
1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
    f=new JFrame();
    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment
   f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
new MyFlowLayout();
} }
```



Event Handling

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.

- **Background Events** - Those events that require the interaction of end user are known as

background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Event classes and Listener interfaces:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

EventHandling Codes:

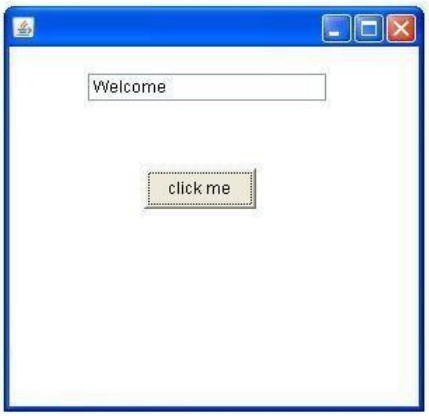
We can put the event handling code into one of the following places:

1. Same class
2. Other class
3. Annonymous class

Example of event handling within class:

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

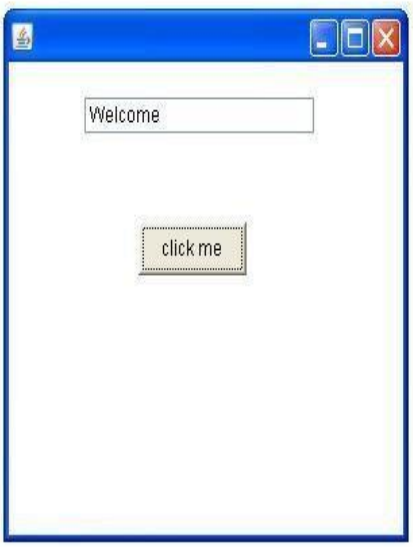
```
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
b.addActionListener(this);
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
} }
```



public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
b.addActionListener(this);//passing current instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
} }
```



Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

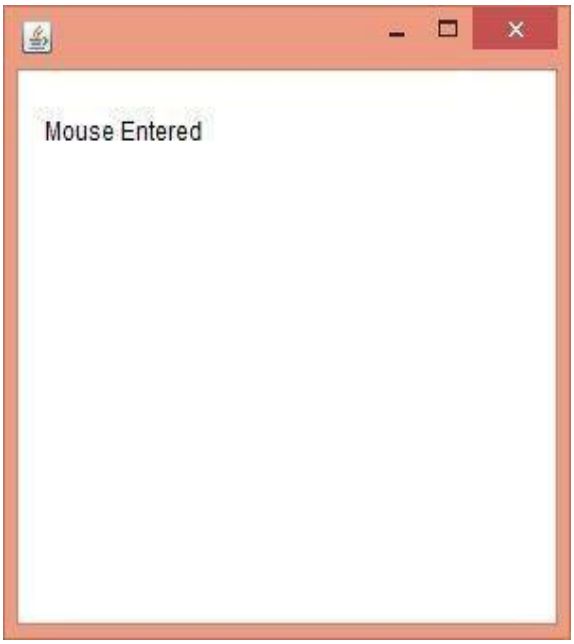
Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

1. public abstract void mouseClicked(MouseEvent e);
2. public abstract void mouseEntered(MouseEvent e);
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. public abstract void mouseReleased(MouseEvent e);

Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
   Label l;
   MouseListenerExample(){
      addMouseListener(this);
      l=new Label();
      l.setBounds(20,50,100,20);
      add(l);
      setSize(300,300);
      setLayout(null);
      setVisible(true);
   }
   public void mouseClicked(MouseEvent e) {
      l.setText("Mouse Clicked");
   }
   public void mouseEntered(MouseEvent e) {
      l.setText("Mouse Entered");
   }
   public void mouseExited(MouseEvent e) {
      l.setText("Mouse Exited");
   }
   public void mousePressed(MouseEvent e) {
      l.setText("Mouse Pressed");
   }
   public void mouseReleased(MouseEvent e) {
      l.setText("Mouse Released");
   }
public static void main(String[] args) {
   new MouseListenerExample();


} }
```



Java KeyListener Interface

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods.

Methods of KeyListener interface

The signature of 3 methods found in KeyListener interface are given below:

1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);

Java KeyListener Example

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
   Label l;
   TextArea area;
   KeyListenerExample(){
      l=new Label();
      l.setBounds(20,50,100,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
      add(l);add(area);
      setSize(400,400);
      setLayout(null);
      setVisible(true);
   }
   public void keyPressed(KeyEvent e) {
      l.setText("Key Pressed");
   }
   public void keyReleased(KeyEvent e) {
      l.setText("Key Released");
   }
   public void keyTyped(KeyEvent e) {
      l.setText("Key Typed");
   }
   public static void main(String[] args) {
      new KeyListenerExample(); }  }
```

Java Adapter Classes

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter Example


```
1. import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
   Frame f;
   AdapterExample(){
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            f.dispose();     }    });
      f.s etSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String[] args) {
   new AdapterExample();
} }
```

Inner Classes:

If we have an applet which does event handling, not only does it need to extend the **Applet** class, it must extend the **Adapter** class also.

But multiple inheritance of classes is not supported in Java. Here we take help of the inner classes. The **Applet** class can be extended by our class, and an inner class is written to extend the **Adapter** class.

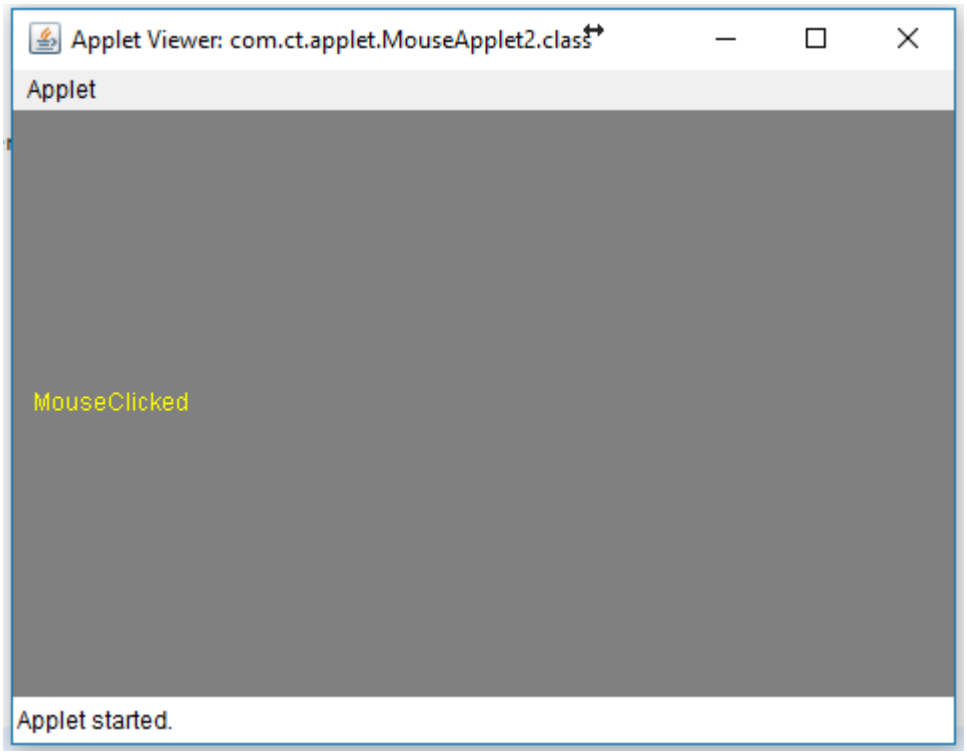
Let us now build a similar class based on **MouseApplet** example, that listens to mouse click event only, and uses the **MouseAdapter** class to do the event handling using inner class.

The changes from the previous example are:

- We add an inner class which extends the adapter class.
- This inner class has a mouseClicked implementation.
- The init method now calls an object of inner class as the MouseListener.
- The paint method just prints the Mouse clicked.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MouseApplet2 extends Applet {
    boolean mouseClicked = false;

    public void init() {
        this.setBackground(Color.gray);
        this.addMouseListener(new MouseAppletAdapter());
    }

    class MouseAppletAdapter extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            mouseClicked = true;
            repaint();
        }

    }

    public void paint(Graphics g) {
        g.setColor(Color.yellow);
        if (mouseClicked) {
            g.drawString("MouseClicked", 10, 150);
            mouseClicked = false;
        }

    }
}
```

OUTPUT:


Applet Viewer: com.ct.applet.MouseApplet2.class
Applet
MouseClicked
Applet started.

Applets

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

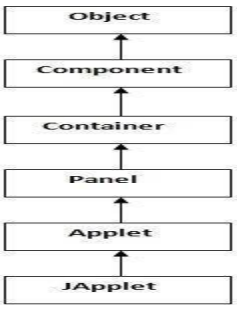
Drawback of Applet

- Plugin is required at client browser to execute applet.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Hierarchy of Applet



Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
1. //First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);
}
}
```

Simple example of Applet by appletviewer tool:

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
1. //First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}
}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

Difference between Applet and Application programming

	Java Applet	Java Application
User graphics	Inherently graphical	Optional
Memory requirements	Java application requirements plus web browser requirements	Minimal java application requirements
Distribution	Linked via HTML and transported via HTTP	Loaded from the file system or by a custom class loading process
Environmental input	Browser client location and size; parameters embedded in the host HTML document	command-line parameters
Method expected by the virtual Machine	init- initialization method start-startup method stop pause/ deactive method destroy-termination method paint-drawing method	Main - startup method
Typical applications	public-access order-entry systems for the web, online multimedia persentations, web page animation	Network server, multimedia kiosks, developer tools, appliance and consumer electronics control and navigation.

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

```
1. public String getParameter(String parameterName)
```

Example of using parameter in Applet:

```
1.  import java.applet.Applet;
2.  import java.awt.Graphics;
3.  public class UseParam
extends Applet 4. {
5. public void
paint(Graphics g)
6. {
7.  String str=getParameter("msg");
8.  g.drawString(str,50, 50);
9. } }
```

myapplet.html

```
1.  <html>
2.  <body>
3.  <applet code="UseParam.class" width="300" height="300">
4.  <param name="msg" value="Welcome to applet">
5.  </applet>
6.  </body>
7.  </html>
```

OTHER SWING COMPONENTS:

JTOGGLE BUTTON:

A JToggleButton is a two-state button. The two states are selected and unselected. When the user presses the toggle button, it toggles between being pressed or unpressed. JToggleButton is used to select a choice from a list of possible choices.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**. Thus, **JToggleButton** defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String *str*)

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection.

When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected. To handle item events, you must implement the **ItemListener** interface. Inside **itemStateChanged( )**, the **getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

Object getItem( )

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected( )** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

boolean isSelected( )

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected( )** to determine the button's state

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JToggleButtonDemo {

  public JToggleButtonDemo() {
    // Set up the JFrame.
    JFrame jfrm = new JFrame("JToggleButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 100);

    // Create a label.
    JLabel jlab = new JLabel("Button is off.");

    // Make a toggle button.
    JToggleButton jtbn =  new JToggleButton("On/Off");

    // Add an item listener for the toggle button.
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("Button is on.");
        else
          jlab.setText("Button is off.");
      }
    });

    // Add the toggle button and label to the content pane.
    jfrm.add(jtbn);
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JToggleButtonDemo();
        }
      }
```

The output from the toggle button example is shown here:

Check Boxes:

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

JCheckBox(String *str*)

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected( )** determines if the box was selected or cleared. The **getText ( )** method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JCheckBoxDemo implements ItemListener {
  JLabel jlab;

  public JCheckBoxDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JCheckBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);

    // Add check boxes to the content pane.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    jfrm.add(cb);

    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    jfrm.add(cb);
```

```
    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    jfrm.add(cb);

    // Create the label and add it to the content pane.
    jlab = new JLabel("Select languages");
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle item events for the check boxes.
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText(cb.getText() + " is selected");
    else
      jlab.setText(cb.getText() + " is cleared");
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JCheckBoxDemo();
        }
      }
    );

  }
}
```

Output from this example is shown here:

Radio Buttons:

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends **JToggleButton**. **JRadioButton** provides several constructors. The one used in the example is shown here:

JRadioButton(String *str*)

Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the **ActionListener** interface. Recall that the only method defined by **ActionListener** is **actionPerformed( )**. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand( )**. By default, the action command is the same as the button label, but you can set the action command to something else by calling **setActionCommand( )** on the radio button. Second, you can call **getSource( )** on the **ActionEvent** object and check that reference against the buttons. Third, you can check each radio button to find out which one is currently selected by calling **isSelected( )** on each button. Finally, each button could use its own action event handler implemented as either an anonymous inner class or a lambda expression. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by **actionPerformed( )**.

Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo implements ActionListener {
  JLabel jlab;

  public JRadioButtonDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JRadioButtonDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(250, 100);
```

```
    // Create radio buttons and add them to content pane.
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    jfrm.add(b1);

    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    jfrm.add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    jfrm.add(b3);

    // Define a button group.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Create a label and add it to the content pane.
    jlab = new JLabel("Select One");
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  // Handle button selection.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JRadioButtonDemo();
        }
      }
    );

  }
}
```

Output from the radio button example is shown here:

JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs..

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. **JTabbedPane** uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab( )**. Here is one of its forms:

void addTab(String *name*, Component *comp*)

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

1. Create an instance of **JTabbedPane**.
2. Add eachn tab by calling **addTab( )**.
3. Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and ontains three check boxes. Each check box displays the name of a color. The third tab is titled

```
// Demonstrate JTabbedPane.
import javax.swing.*;
import java.awt.*;

public class JTabbedPaneDemo {

  public JTabbedPaneDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JTabbedPaneDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 200);

    // Create the tabbed pane.
    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    jfrm.add(jtp);
```

```
        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String[] args) {
        // Create the frame on the event dispatching thread.

        SwingUtilities.invokeLater(
          new Runnable() {
            public void run() {
              new JTabbedPaneDemo();
            }
          }
        );

    }
}

// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {

    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}

class ColorsPanel extends JPanel {

    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}

class FlavorsPanel extends JPanel {

    public FlavorsPanel() {
        JComboBox<String> jcb = new JComboBox<String>();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");

        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```

Output from the tabbed pane example is shown in the following three illustrations:

JScrollPane:

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.
Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed. **JScrollPane** defines several constructors. The one used in this chapter is shown here:

JScrollPane(Component *comp*)

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.

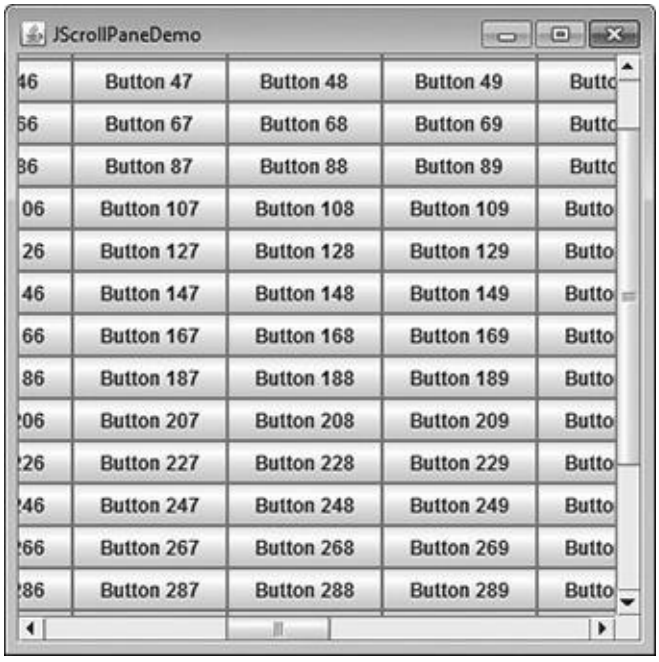
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane.
Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;

public class JScrollPaneDemo {

  public JScrollPaneDemo() {

    // Set up the JFrame.  Use the default BorderLayot.
    JFrame jfrm = new JFrame("JScrollPaneDemo");
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 400);

    // Create a panel and add 400 buttons to it.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));

    int b = 0;
    for(int i = 0; i < 20; i++) {
      for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
      }
    }

    // Create the scroll pane.
    JScrollPane jsp = new JScrollPane(jp);

    // Add the scroll pane to the content pane.
    // Because the default border layout is used,
    // the scroll pane will be added to the center.
    jfrm.add(jsp, BorderLayout.CENTER);

    // Display the frame.
    jfrm.setVisible(true);
  }
```

```
  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JScrollPaneDemo();
        }
      }
    );

  }
}
```

Output from the scroll pane example is shown here:



JList:

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list

JList provides several constructors. The one used here is

JList(E[ ] *items*)

This creates a **JList** that contains the items in the array specified by *items*.

JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged( )**, which is shown here:

void valueChanged(ListSelectionEvent *le*)

Here, *le* is a reference to the event. Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:

int getSelectedIndex( )

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is

selected, –1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue()**:

```
E getSelectedValue( )
```

It returns a reference to the first selected value. If no value has been selected, it returns **null**.

The following program demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged( )** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class JListDemo {

  // Create an array of cities.
  String Cities[] = { "New York", "Chicago", "Houston",
                      "Denver", "Los Angeles", "Seattle",
                      "London", "Paris", "New Delhi",
                      "Hong Kong", "Tokyo", "Sydney" };

  public JListDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JListDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(200, 200);

    // Create a JList.
    JList<String> jlst = new JList<String>(Cities);

    // Set the list selection mode to single-selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add the list to a scroll pane.
    JScrollPane jscrlp = new JScrollPane(jlst);

    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Make a label that displays the selection.
    JLabel jlab = new JLabel("Choose a City");

    // Add selection listener for the list.
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Get the index of the changed item.
        int idx = jlst.getSelectedIndex();

        // Display selection, if item was selected.
        if(idx != -1)
          jlab.setText("Current selection: " + Cities[idx]);
```

```
        else // Otherwise, reprompt.
          jlab.setText("Choose a City");
      }
    });

    // Add the list and label to the content pane.
    jfrm.add(jscrlp);
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JListDemo();
        }
      }
    );

  }
}
```

Output from the list example is shown here:

JComboBox:

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry.
You can also create a combo box that lets the user enter a selection into the text field.
JComboBox was made generic and is now declared like this:class JComboBox<E>

Here, **E** represents the type of the items in the combo box.
The **JComboBox** constructor used by the example is shown here: JComboBox(E[ ] *items*)

Here, *items* is an array that initializes the combo box. Other constructors are available.
In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem( )** method, shown here:

void addItem(E *obj*)

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.
JComboBox generates an action event when the user selects an item from the list.
JComboBox also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.
One way to obtain the item selected in the list is to call **getSelectedItem( )** on the combo box. It is shown here:

Object getSelectedItem( )

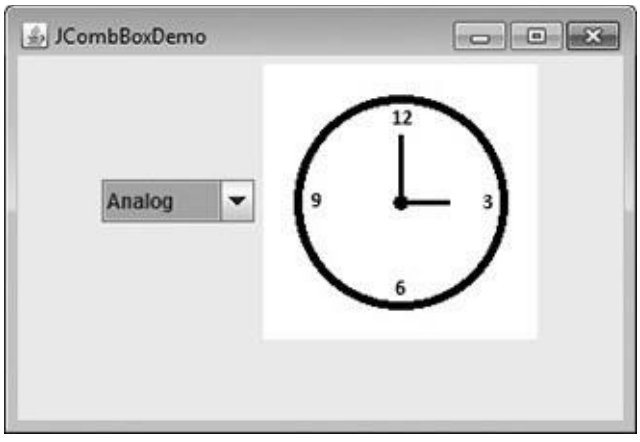
You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "Hourglass", "Analog", "Digital", and "Stopwatch". When a timepiece is selected, an icon-based label is updated to display it. You can see how little code is required to use this powerfulcomponent.

```
// Demonstrate JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JComboBoxDemo {

  String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };

  public JComboBoxDemo() {

    // Set up the JFrame.
    JFrame jfrm = new JFrame("JCombBoxDemo");
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    jfrm.setSize(400, 250);

    // Instantiate a combo box and add it to the content pane.
    JComboBox<String> jcb = new JComboBox<String>(timepieces);
    jfrm.add(jcb);
```

```
    // Create a label and add it to the content pane.
    JLabel jlab = new JLabel(new ImageIcon("hourglass.png"));
    jfrm.add(jlab);

    // Handle selections.
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        String s = (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".png"));
      }
    });

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String[] args) {
    // Create the frame on the event dispatching thread.

    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          new JComboBoxDemo();
        }
      }
    );

  }
}
```

OUTPUT:



SWING MENUS:

JMenuBar, JMenu and JMenuItems are a part of Java Swing package. JMenuBar is an implementation of menu bar . the JMenuBar contains one or more JMenu objects, when the JMenu objects are selected they display a popup showing one or more JMenuItems .

JMenu basically represents a menu . It contains several JMenuItem Object . It may also contain JMenu Objects (or submenu).

Constructors :

1. **JMenuBar() :** Creates a new MenuBar.
2. **JMenu() :** Creates a new Menu with no text.
3. **JMenu(String name) :** Creates a new Menu with a specified name.
4. **JMenu(String name, boolean b) :** Creates a new Menu with a specified name and boolean value specifies it as a tear-off menu or not. A tear-off menu can be opened and dragged away from its parent menu bar or menu.

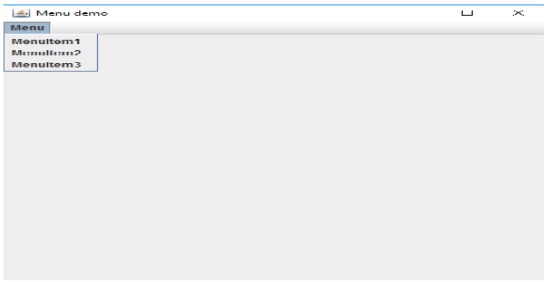
Commonly used methods:

1. **dd(JMenu c) :** Adds menu to the menu bar. Adds JMenu object to the Menu bar.
2. **add(Component c) :** Add component to the end of JMenu
3. **add(Component c, int index) :** Add component to the specified index of JMenu
4. **add(JMenuItem menuItem) :** Adds menu item to the end of the menu.
5. **add(String s) :** Creates a menu item with specified string and appends it to the end of menu.
6. **getItem(int index) :** Returns the specified menuitem at the given index

// **Java program to construct**
// Menu bar to add menu items

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class menu extends JFrame {
                                        // menubar
                                    static JMenuBar mb;

                                        // JMenu
                                    static JMenu x;

                                    // Menu items
                                static JMenuItem m1, m2, m3;

                                    // create a frame
                                    static JFrame f;

                                public static void main()
                                        {

                                            // create a frame
                                            f = new JFrame("Menu demo");

                                            // create a menubar
                                            mb = new JMenuBar();

                                            // create a menu
```

```
        x = new JMenu("Menu");

        // create menuitems
        m1 = new JMenuItem("MenuItem1");
        m2 = new JMenuItem("MenuItem2");
        m3 = new JMenuItem("MenuItem3");

        // add menu items to menu
        x.add(m1);
        x.add(m2);
        x.add(m3);

        // add menu to menu bar
        mb.add(x);

        // add menubar to frame
        f.setJMenuBar(mb);

        // set the size of the frame
        f.setSize(500, 500);
        f.setVisible(true);
    }
}
```

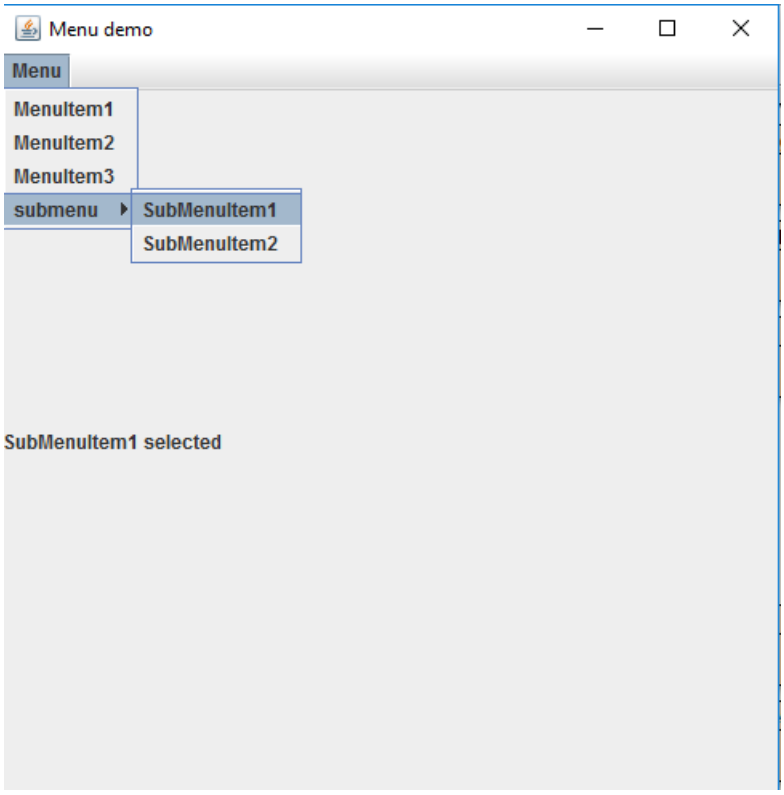


. Program to add a menubar and add menuitems, submenu items and also add ActionListener to menu items

```
// Java program Program to add a menubar
// and add menuitems, submenu items and also add
// ActionListener to menu items
```

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class menu1 extends JFrame implements ActionListener {
	// menubar
	static JMenuBar mb;

	// JMenu
	static JMenu x, x1;

	// Menu items
	static JMenuItem m1, m2, m3, s1, s2;

	// create a frame
	static JFrame f;

	// a label
	static JLabel l;

	// main class
	public static void main()
	{
		// create an object of the class
		menu1 m = new menu1();

		// create a frame
		f = new JFrame("Menu demo");

		// create a label
		l = new JLabel("no task ");

		// create a menubar
		mb = new JMenuBar();

		// create a menu
		x = new JMenu("Menu");
		x1 = new JMenu("submenu");

		// create menuitems
		m1 = new JMenuItem("MenuItem1");
		m2 = new JMenuItem("MenuItem2");
		m3 = new JMenuItem("MenuItem3");
		s1 = new JMenuItem("SubMenuItem1");
		s2 = new JMenuItem("SubMenuItem2");

		// add ActionListener to menuItems
		m1.addActionListener(m);
		m2.addActionListener(m);
		m3.addActionListener(m);
```

```
            s1.addActionListener(m);
            s2.addActionListener(m);

            // add menu items to menu
            x.add(m1);
            x.add(m2);
            x.add(m3);
            x1.add(s1);
            x1.add(s2);

            // add submenu
            x.add(x1);

            // add menu to menu bar
            mb.add(x);

            // add menubar to frame
            f.setJMenuBar(mb);

            // add label
            f.add(l);

            // set the size of the frame
            f.setSize(500, 500);
            f.setVisible(true);
        }
        public void actionPerformed(ActionEvent e)
        {
            String s = e.getActionCommand();

            // set the label to the menuItem that is selected
            l.setText(s + " selected");
        }
}
```


Menu demo
Menu
MenuItem1
MenuItem2
MenuItem3
submenu
SubMenuItem1
SubMenuItem2
SubMenuItem1 selected