JAVA PROGRAMMING
UNIT - 1

GENERAL STRUCTURE OF THE JAVA PROGRAM

WRITE SHORT NOTES ON THE GENERAL STRUCTURE OF THE JAVA PROGRAM.

Documentation section	(suggested)
Package section	(optional)
Import section	(optional)
Interface section	(optional)
Class Definition	(optional)

Main method
{
Main method definition
}
	(Essential)

<u>Documentation section:</u>

It contains set of comment lines giving the name of the program, the author and other details. Java also uses a third style of comment /* *…. */

<u>Package statement:</u>

The first statement in java is package statement, declares a package name and informs the compiler that the classes defined here belong to this package.

Ex: Package student;

It is optional.

<u>Import statement:</u>

This statement is similar to the #include statement in C.

Ex:

import student.Test;

The Test class contained in the package students.

<u>Interface statement:</u>

It is like a class but includes a group of method declarations. It is used to implementing multiple inheritances.

<u>Class definitions:</u>

A java program may contain multiple class definitions. The number of classes used depends on the complexity of the program.

<u>Main method class:</u>

It is essential part of a java program. The main method creates objects of various classes and establishes communication between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

FEATURES OF JAVA

EXPLAIN THE FEATURES OF JAVA IN BRIEF.

The Sun Microsystems describes Java with the following attributes.

- Compiled and interpreted
- Platform – independent and portable
- Object oriented
- Robust and secure
- Distributed
- Familiar, simple and small
- Multithreaded and Interactive
- High performance
- Dynamic and Extensible

COMPILED AND INTERPRETED:

- Usually a computer language is either compiled or interpreted.
- JAVA is both a compiled and an interpreted language. It is a two-stage system.
- First, java compiler translates source code into byte code instructions.
- Java interpreter generates machine code that can be directly executed by the machine that is running the java program.

PLATFORM-INDEPENDENT AND PORTABLE:

- The most significant contribution of java other language is its **portability**.
- Java programs can be easily moved from one computer system to another, anywhere and anytime.
- We can download a java applet from a remote computer on to our local system via internet & execute it locally.

Java ensures portability in two ways:

- First java compiler generates byte code instructions that can be implemented on any machine.
- Secondly, the sizes of the primitive data types are machine independent.

OBJECT-ORIENTED:

- In java all programs code & data reside within objects & classes.
- Java comes with an extensive set of classes, arranged in packages that we can use in our programs by inheritance.

ROBUST AND SECURE:

- Java is a robust language.
- It provides many safeguards to ensure reliable code.
- It is designed as garbage collected language receiving the programmers virtually all memory management problems.
- It incorporates the concept of exception handling which capture series errors & eliminates any risk of crashing the system.
- The absence of pointers in java ensures that programs cannot gain access to memory locations without proper authorization.
- **Secure:** Java systems not only verify all memory but also ensure that no viruses are communicated with an applet.

DISTRIBUTED:

- Java is designed for the distributed language for creating applications on networks.
- It has the ability to share both data & program.
- It enables multiple programmers at multiple remote locations to collaborate & work together on a single project.
- JAVA program is easily moved from one system to another.

SIMPLE, SMALL & FAMILIAR:

- Java is a small and simple language.
- It does not use pointers, preprocessor header files, goto statement and many others.
- It also eliminates operator overloading and multiple inheritance.
- Familiarity is another striking feature of java.
- Java uses many constructors of C &C++.
- It is simplified version of C++.

MULTITHREADED AND INTERACTIVE:

- Multithreaded means handling multiple tasks simultaneously.
- It supports multithreaded programs. This means we need not wait for the application to finish one task before beginning another. It supports runtime multiprocessing for interactive systems.
- It improves the interactive performance of graphical application.
- The java runtime comes with tools that support multi process synchronization.

HIGH PERFORMANCE:

JAVA performance is impressive for an interpreted language, mainly use of intermediate byte code.

DYNAMIC AND EXTENSIBLE:

JAVA is a dynamic language. It is capable of dynamically linking class, methods and objects.
It supports functions written in other languages such as C and C++. These functions are known as native methods.

DATA TYPES, VARIABLES AND ARRAYS

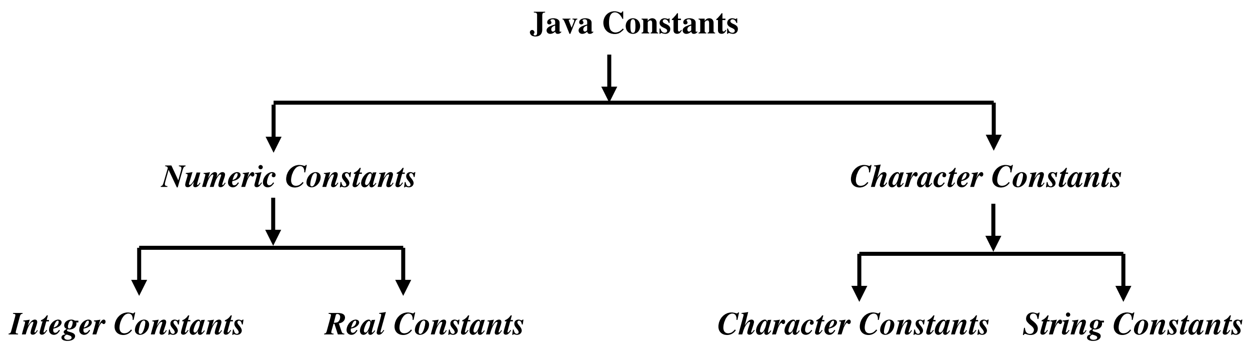
INTRODUCTION:

The task of processing data is accomplished by executing a sequence of instructions constituting a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax rules.

DEFINE CONSTANT.

Constants in java refer to fixed values that do not change during the execution of a program.

EXPLAIN THE TYPES OF CONSTANTS WITH EXAMPLE.



Integer Constants:

An integer constant refers to a sequence of digits. 3 types:

- ***Decimal integer*** ➔ It consists of set of digits 0 to 9, preceded by an optional minus sign. (Ex: 12450)
- ***Octal Integer*** ➔ It consists of any combination of digits from the set 0 to 7 with leading 0. (Ex***:*** 037)
- ***Hexadecimal integer*** ➔ It consists of sequence of digits proceeded by 0x or 0X. It may also include alphabets A through F or 'a' to 'f'. It represents the number 10 to 15. (Ex***:*** 0X2, 0X9F)

Real Constants:

Numbers containing fractional part are called real constants.

Ex: 215.03, 0.75, -0.52, 0.0083

It have a whole number followed by a decimal point and the fractional part. It is possible that the number may not have digits before the decimal point or digits after the decimal point.

Ex: 215.00, 0.95, -0.71

A real number may also be expressed in exponential or scientific notation.

Ex: 215.65 ➔ 2.1565e2

Where e2 means multiply by 10^2

Syntax: ***Mantissa e exponent***

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer with an optional plus or minus sign.

A floating point constant has 4 parts:

➔ a whole number
➔ a decimal point
➔ a fractional part
➔ an exponent.

Single Character Constants: A single character constant contains a single character enclosed within a pair of single double quote marks.

Ex: '6', 'y', ';' , ' '

String Constants: A string contains is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Ex: "hello java" "1979" "welcome", "r"

<u>Backslash Character Constants:</u>

It is used in output methods. They consist of two characters. These characters combinations are known escape sequence.

Constants	**Meaning**
'\b'	back space
'\f'	form feed
'\n'	new line
'\\'	back slash
'\r'	carriage return
'\t'	horizontal tab
'\"'	double quote
'\''	single quote

DEFINE VARIABLE. WHAT ARE THE RULES FOLLOWED BY DECLARE A VARIABLE.

A Variable is an identifier that denotes a storage location used to store a data value. The name can be chosen by the programmer in a meaning way.

Variable names consist of alphabets, digits, underscore (-) and dollar characters, subject to following conditions:

<u>Rules:</u>

- ✓ They must not begin with a digit.
- ✓ Upper case and Lower case letters are distinct.
- ✓ It should not be a keyword.
- ✓ White space is not allowed.
- ✓ Variables names can be of any length.

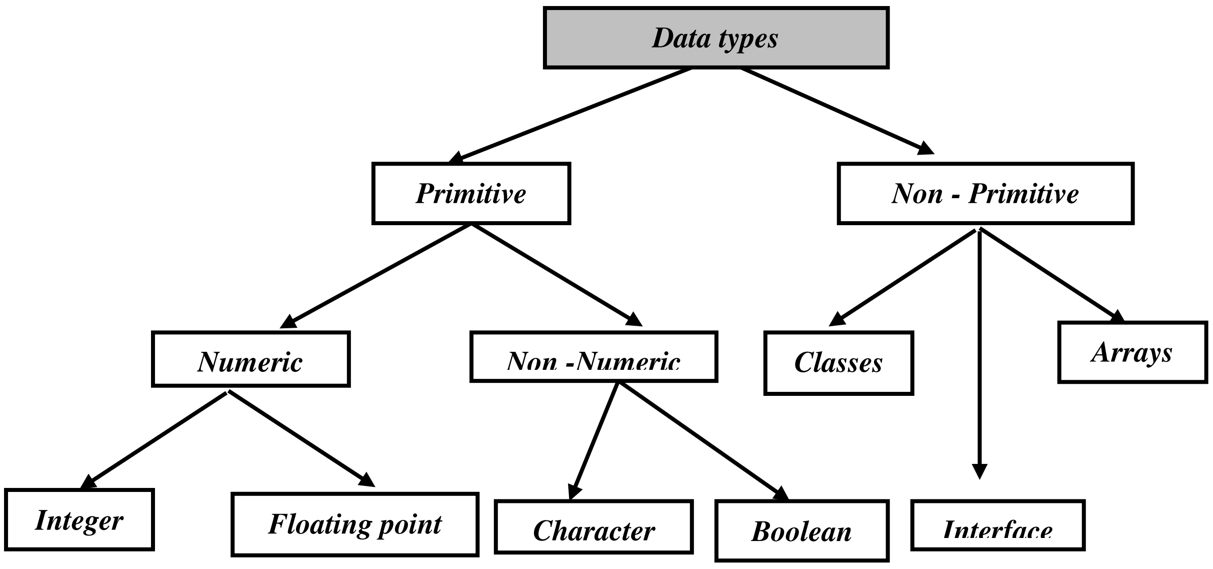
EXPLAIN THE VARIOUS DATA TYPE IN JAVA WITH EXAMPLE.

Data types specify the size and types of values that can be stored. The variety of data types available allows the programmer to select the type appropriate to the needs of the application.

- ➢ Primitive types (intrinsic or built in types)
- ➢ Derived types (reference type)

Integer types:

- ❖ A whole number without decimal point. It can hold whole numbers.
- ❖ The size of the values depends on the integer data type.
- ❖ The size of the values that can be stored depending on the integer type.
- ❖ Java does not support the concept of unsigned types. So it may be signed ie positive or negative.
- ❖ Java supports four types of integers.



Type	*Size*	*Minimum value*	*Maximum value*
byte	1 byte	-128	127
short	2 byte	-32,768	32,767
int	4 byte	-2,147,483,648	2,147,483,647
long	8 byte	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Floating-point types:

A whole number with decimal point. There are two kinds of floating point storage in Java.

- The **float** type values are single-precision numbers while the double types represent double-precision numbers.
- **Double** precision types are used when we need grater precision in storage of floating point numbers.

Append 'f 'or 'F' to single precision mode numbers. ***Ex:*** 1.23f, 7.5694e5f

Mathematical functions such as sin, cos and sqrt return double type values.

Character type:

Java provides a character data type called char. It assumes a size of 2 bytes but it can hold only a single character.

Boolean type:

It is used to test a particular condition during the execution of the program. There are only two values: true or false. This data type is denoted by the keyword Boolean and use only one bit of storage. Boolean values are often used in selection and iteration statement.

Literals in Java

Integral, Floating-Point, Char, String, Boolean

Introduction

- *Literals are number, text, or anything that represent a value.*
- *Literals in Java are the constant values assigned to the variable. It is also called a constant.*

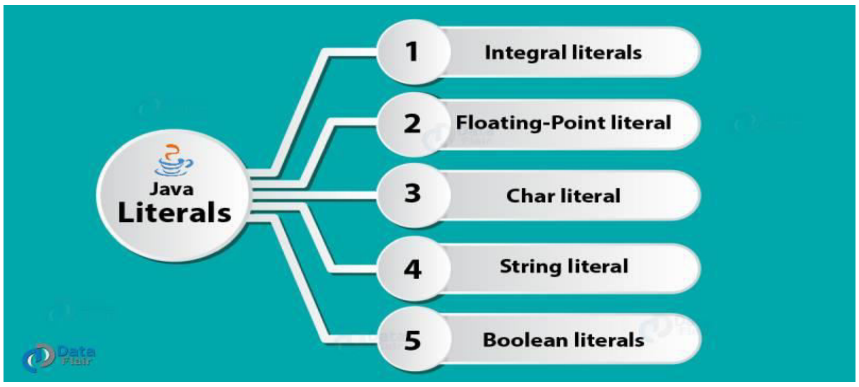
For example,

int x = 100;

So, 100 is literal.

There are 5 types of Literals can be seen in Java.

Types of Literals in Java



1. Integral Literals in Java

We can specify the integer literals in 4 different ways –

- **Decimal (Base 10)**

Digits from 0-9 are allowed in this form.

```
Int x = 101;
```

- **Octal (Base 8)**

Digits from 0 – 7 are allowed. It should always have a prefix 0.

int x = 0146;

- **Hexa-Decimal (Base 16)**

Digits 0-9 are allowed and also characters from a-f are allowed in this form. Furthermore, both uppercase and lowercase characters can be used, *Java provides an exception* here.
int x = 0X123Face;

- **Binary**

A literal in this type should have a prefix 0b and 0B, from 1.7 one can also specify in binary literals, i.e. 0 and 1.

```
int x = 0b1111;
```

2. Floating-Point Literals in Java

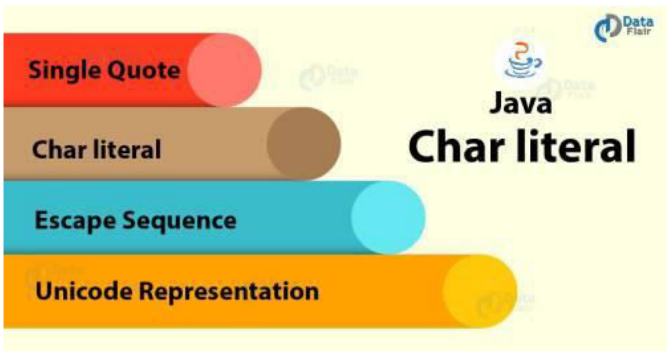
Here, datatypes can only be specified in decimal forms and not in octal or hexadecimal form.

- **Decimal (Base 10)**

Every floating type is a double type and this the reason why we cannot assign it directly to float variable, to escape this situation we use f or F as suffix, and for double we use d or D.

3. Char Literals in Java

These are the four types of char-



- **Single Quote**

Java Literal can be specified to a char data type as a single character within a single quote.

```
char ch = 'a';
```

- Char as Integral
- A char literal in Java can specify as integral literal which also represents the Unicode value of a character.
  Furthermore, an integer can specify in decimal, octal and even hexadecimal type, but the range is 0-6**Unicode Representation**

Char literals can specify in Unicode representation '\uxxxx'. Here XXXX represents 4 hexadecimal numbers.

```
char ch = '\u0061';// Here /u0061 represent a.
```

Escape Sequence

Escape sequences can also specify as char literal.

```
char ch = '\n';
```

String literals

String literals are any sequence of characters with a double quote.

```
String s = "Hello";
```

They may not contain unescaped newline or linefeed characters.

However, the Java compiler will evaluate compile-time expressions.

Boolean Literals

They allow only two values i.e. true and false.

```
boolean b = true;
```

EXPLAIN THE DECLARATION OF VARIABLES WITH EXAMPLE.

- It tells the compiler what the variable name is
- It specifies what type of data the variable will hold.
- The place of declaration decides the scope of the variable.

The variable can be used to store a value of any data type.

Syntax:

type variable1, variable2,….variable n;

Ex:

```
int x;
float x,y;
byte b;
char c1,c2,c3,c4;
```

GIVING VALUES TO THE VARIABLE:

The value is given to the variable in two ways:

1. By using assignment statement
2. By using a read statement

Assignment statement:

The value is given to the variable is through the assignment statement as follows:

Syntax	**Example**
variable name=value	x=0;
type variablename=value	int x=0;

Read statement:

The values is given to the variables through the keyboard using readLine( ) method.

Reading data from keyboard:

```
import java.io.DataInputStream;
class Reading
{
public static void main(String a[])
{
```

```
DataInputStream dis=new DataInputStream(System.in);
int intnumber = 0;
float floatnumber = 0.0f;
            try
            {
                  System.out.println("enter an integer :");
                  intnumber= Integer.parseInt(dis.readLine());
                  System.out.println("enter a float number: ");
                  floatnumber= Float.valueOf(dis.readLine()).floatValue();
            }
            catch(Exception e) { }
            System.out.println("intnumber = "+intnumber);
            System.out.println("floatnumber = "+floatnumber);
      }
}
```

EXPLAIN ABOUT SCOPE OF VARIBALES.

Java variables are actually classified into three kinds:

- Instance variables
- Class variables
- Local variables

Instance variables:

Instance and class variables are declared inside a class. Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different values for each object.

Class variables:

Class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable.

Local variables:

Variables declared and used inside methods are called local variables. They are called so because they are not available for use outside the method definition. Local variables can also be declared inside program blocks that are defined between an opening brace {and a closing brace}.

These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible is called its **scope.**

EXPLAIN ABOUT SYMBOLIC CONSTANT.

- Symbolic names take the same form as variables names. But they are written in CAPITALS to visually distinguish them from normal variable names. This is only a conversion, not a rule.
- After declaration of symbolic constant, they should not be assigned any other value within the program by using an assignment statement. Ex: STRENGTH=200, is legal.
- Symbolic constants are declared for types. This is not done in C and C++ where symbolic constants are defined using the # define statement.
- They can NOT be declared inside a method. They should be used only as class data members in the beginning of the class.

WRITE NOTES ON TYPE CASTING.

The value to be stored by preceding it with the type name in parentheses.

Syntax:

Type variable = (type) variable2;

The process of converting one data type to another is called casting.

Ex:

```
Int m= 50;
Byte n= (byte) m;
```

Automatic conversion:

It is possible to assign a value of one type to variables of a different type without a cast. Java does the conversion of the assigned value automatically. This is known as automatic type conversion.

Ex: Byte b=75; Int a=b;

HOW DO GETTING VALUES OF VARIBLES.

Java support two output methods that can be used to send the results to the screen.

- Print( ) method // print and wait
- Println( ) method // print a line and move to next line

The print ( ) method sends information into a buffer. This buffer is not flushed until a new line character is sent. As a result, the print( ) method prints output on one line until a newline character is encountered.

Ex:

```
System.out.print (" Hello");
System.out.print (" \n");
System.out.print (" Welcome");
```

Or

```
System.out.println (" Hello");
System.out.println (" Welcome");
```

TYPE CONVERSION IN EXPRESSIONS:

1. Automatic type conversion
2. Casting a value

Automatic Type Conversion:

Java permits of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the higher type before the operations proceeds. The result is of the higher type. The following changes are introduced during the final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long to int causes dropping of the excess higher order bits.

Casting a Value:

There are instances when we want to force a type conversion in a way that is different from the automatic conversion.

Ratio = (float) female_number/male_number

The general form of a cast is

(type-name) expression

Examples	**Action**
b = (**double**) sum/n	Division is done in floating point mode.
p = cost ((**double**) x)	Converts x to double before using it as parameter.

ARRAYS

EXPLAIN ABOUT ARRAYS.

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

A particular value is indicated by writing a number called ***index number*** or ***subscript*** in brackets after the array name.

Ex: **Sum [10];**

It represents the sum of 10 numbers. The complete set of values is referred to as an array; the individual values are called ***elements***.

Arrays can be of any variable type:

The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs.

There are 3 types:

- One-dimensional array

- Two dimensional array
- Multi-dimensional array

EXPLAIN ONE DIMENSIONAL ARRAY.

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted or one-dimensional array.

An array must be declared and created in the computer memory before they are used. Creating an array involves 3 steps.

➔ Declare the array.
➔ Create memory locations.
➔ Put values into the memory locations.

Declaration of array:

There are two forms to declare

1. ***type arrayname[ ];***
2. ***type [ ] arrayname;***

<u>Creation of arrays:</u>

Java allows to create arrays using new operator. **new** is a special operator that allocates memory.

General form : var-name = new type[size];
Ex : a = new int[10];
General form : type var-name[];
Ex : int a[10];

i.e. a[0], a[1],…..a[9]. The subscript can begin with number 0.That is a[0] is allowed.The subscript of an array can be integer constants, integer variable like 'i' or expressions that yield integers.

EXPLAIN DETAIL ABOUT TWO DIMENSIONAL ARRAYS.

Two dimensional array values are referred by using subscripts for both the column and row of the corresponding element. Two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Example:

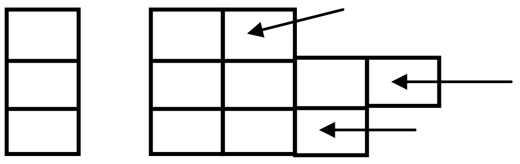
int x[][]=new int[4][4];

<u>Multidimensional Arrays:</u>

Multidimensional array is referred as "arrays of arrays". To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Example:

```
int a[ ][ ] = new int[3] [ ];
int a[0] = new int[2];
int a[1] = new int[4];
int a[2] = new int[3];
```



(Rows → 3, Colomns → undefined)

Ex:

```
class Table
{
      final static int rows =10;
      final static int cols = 10;
      public static void main(String a[])
      {
            int p[] [] = new int[rows][cols];
            int i,j;
            for(i=10; i<rows; i++)
            {
                  for(j=10; j<cols; j++)
                  {
                        p[i][j] = i * j;
                        System.out.println(p[i][j]);
```

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

TYPE CONVERSION IN EXPRESSIONS:

1. Automatic type conversion
2. Casting a value

Automatic Type Conversion:

Java permits of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the higher type before the operations proceeds. The result is of the higher type. The following changes are introduced during the final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long to int causes dropping of the excess higher order bits.

Casting a Value:

There are instances when we want to force a type conversion in a way that is different from the automatic conversion.

Ratio = (float) female_number/male_number

The general form of a cast is

(type-name) expression

Examples	**Action**
b = (**double**) sum/n	Division is done in floating point mode.
p = cost ((**double**) x)	Converts x to double before using it as parameter.

INTRODUCTION TO CLASSES

DEFINE A CLASS AND HOW TO DEFINE A CLASS.

Defining a class:

- A class is a user-defined data-type that serves to define its properties.
- Java programs must be encapsulated in a class that defines the state and behavior of the basic program components known as **objects.**

a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Classes create objects and objects use methods to communicate between them.

General form of a class:

A class is declared by use of the **class** keyword.

```
class class-name [extends super class name]
{
        [Instance variable declaration;]
        [Methods declaration;]
}
```

- The data, or variables declared within a class are called as **instance variables**. Instance variables are also known as **member variables**.
- Collectively, the methods and variables defined within a class are called members of the class.
- Everything inside the square brackets is optional.
- Class name and super class name are any valid java identifiers.
- The keyword extends indicates that the properties of the super class are extended to the class name class. This concept is known as **inheritance.**

Simple class:

Declare the instance variables exactly the same way as we define local variables.

```
class Box
{
        int width, height, depth;
}
```

Here the Box class defines three instance variables: width, height and depth. The class does not contain any methods. A class defines a new type of data. In this case, the new data type is called **Box.** Use this name to declare objects of type **Box.**

Example:

```
 class Box{
                int height, width, depth;
          }
class Demo {
                public static void main(String a[])
                {
                        Box mybox = new Box();
                        int volume;
                        mybox.height = 20;
                        mybox.depth = 15;
                        volume = mybox.width * mybox.height * mybox.depth;
                        System.out.println("Volume is  = " + volume);
        }
}
```

OBJECTS

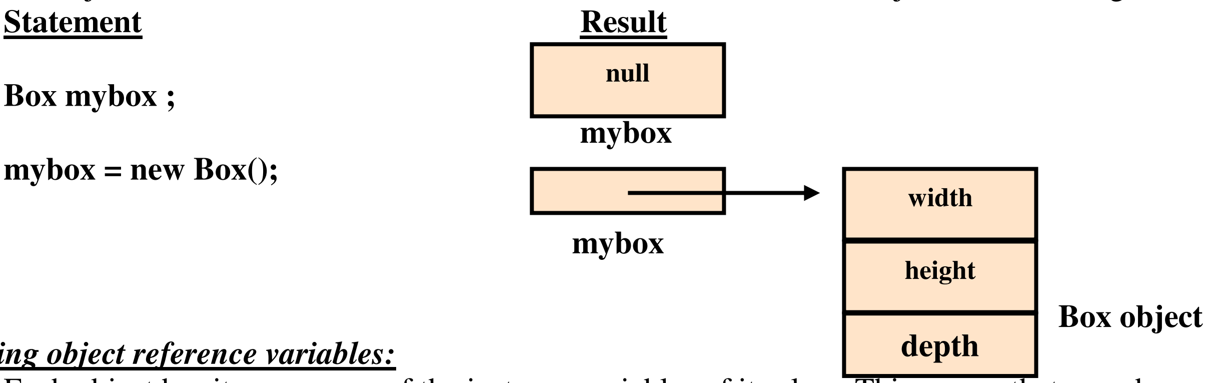
HOW WILL YOU CREATE AN OBJECT IN THE CLASS?

Declaring objects:

- An object is also referred to as instantiating an object. It is a block of memory that contains space to store the entire instance variable.
- An object in java is created using **new** operator. This operator is used to create an object of the specified class and returns a reference to that object.

```
Ex:  Box  mybox;              // declare reference to object
     mybox =  new Box( );     // allocate a Box object
```

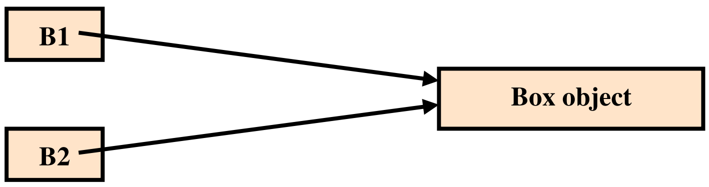
The first statement declares a variable to hold the object reference and the second one actually assigns the objects reference to the variables. The variable rect1 is now an object of the rectangle class.



Assigning object reference variables:

Each object has its own copy of the instance variables of its class. This means that any changes to the variables of another also create two or more reference to the same object.

```
Box b1 =  new Box( );
Box b2 = b1
```



When we assign one object reference variable to another object reference variable, then we are not creating a copy of the object, instead we are only making a copy of the reference.

Introducing methods:

Methods are declared inside the body of the class but immediately after the declaration of instance variables. The general form of a method declaration is

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

There are four basic parts:

- The name of the method (method name).
- The type of the value the method returns (type).
- A list of parameters (parameter list).
- The body of the method.

- The type specifies the type of values the method would return.
- This could be a simple data type such as int as well as any class type. It could even be void type, if the method does not return any values. The method name is valid identifiers.
- The parameter list is always enclosed in parenthesis.
- The variables in the list are separated by commas.
- Methods that have a return type other than **void** return a value to the calling routine using the **return** statement.

```
return value;
```

Adding a method to the Box class:

```
class Box
{
        int height, width;
        void getData(int x , int y)
        {
                height=x;
                width=y;
        }
}
```

The getData method is basically added to provide values to the instance variables. Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables declared in other methods.

There are two things to understand about returning values:

1. The type of data returned by a method must be compatible with the return type specified by the method.
2. The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Adding a method that takes parameters

A parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

Here is a method that returns the square of the number 10:

```
int square()
{
        return 10 * 10;
}
```

The above method return the value of 10 squared, its use is very limited

```
int square(int i)
{
        return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

DEFINE PARAMETER / ARGUMENT.

- A *parameter* is a variable defined by a method that receives a value when the method is called. Ex: In **square( )**, **i** is a parameter.
- An *argument* is a value that is passed to a method when it is invoked. Ex: **square(100)** passes 100 as an argument.
- In a well-designed Java programs, instance variables should be accessed only through methods defined by their class.

DEFINE CONSTRUCTOR.

Constructors:

- ★ Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- ★ A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- ★ Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- ★ They have no return type, not even **void**. This is because the implicit return type of a class constructor is the class type itself.
- ★ It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.
- ★ When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- ★ The default constructor automatically initializes all instance variables to zero. It is often sufficient for simple classes.
- ★ Once user defines his own constructor, the default constructor is no longer used.

Example:

1

```
class Box
{   double width, height,depth;
    Box ( )    // This is the constructor
    {
      width = 10;  height = 10;
      depth = 10;
    }
    double volume ( )
    {
       return width * height * depth;
    }
}
```

2

```
class BoxMain
{
  public static void main(String args[])
  {
        Box mybox1 = new Box ( );
        Box mybox2 = new Box ( );
        double vol;
        vol = mybox1.volume ( );
        System.out.println("Volume is " + vol);
        vol = mybox2.volume ( );
        System.out.println("Volume is " + vol);
  }
}
```

Output

```
Volume is 1000.0
Volume is 1000.0
```

Parameterized Constructors:

- ★ To construct **Box** objects of various dimensions, add parameters to the constructor.
- ★ For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters.

1

```
class Box
{
        double width,height, depth;
  Box (double w, double h, double d)
  {
        width = w; height = h; depth = d;
  }
  double volume()
  {
        return width * height * depth;
  }
```

2

```
class BoxMain
{
    public static void main(String args[])
     {
          Box mybox1 = new Box(10, 20, 15);
          Box mybox2 = new Box(3, 6, 9);
          double vol;
          vol = mybox1.volume();
          System.out.println("Volume is " +
vol);
          vol = mybox2.volume();
          System.out.println("Volume is " +
vol);
     }
}
```

The this Keyword:

Java defines the **this** keyword. It can be used inside any method to refer to the current object. That is, **this** is always a reference to the object on which the method was invoked.

```
Box (int w, int h)   // Constructor
{
        this.height = h;
        this.width = w;
}
```

Instance variable hiding:

When a local variable has the same name as an instance variable, the local variable hides the instance variable.

WRITE ABOUT GARBAGE COLLECTION.

Garbage collection:

- ★ Objects are dynamically allocated by using the **new** operator. Such objects can be destroyed and their memory released is used for later reallocation.
- ★ Java handles deallocation automatically. The technique that accomplishes this is called *garbage collection.*
- ★ When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- ★ Garbage collection only occurs during the execution of the program. It will not occur simply because one or more objects exist that are no longer used.

The finalize ( ) Method:

- ★ Sometimes an object will need to perform some action when it is destroyed. To handle such situations, Java provides a mechanism called finalization.
- ★ By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- ★ To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class.

- ★ Inside the **finalize( )** method we will specify those actions that must be performed before an object is destroyed.
- ★ The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
        // finalization code here
}
```

- **protected** → is a specifier that prevents access to **finalize( )** by code defined outside its class.

SIMPLE JAVA PROGRAM

Ex: 1

```
Class Sample
{
    Public static void main (String a[])
     {
       System.out.println ("COLLEGE");
     }
}
```

Ex: 2

```
class Classname
{
    public static void main (String args[ ])
     {
         int num;
         num = 100;
         System.out.println (num);
         num = num * 3;
         System.out.print(Result is = "+num);
     }
}
```

UNIT I COMPLETED

UNIT- II

METHOD OVERLOADING

EXPLAIN METHOD OVERLOADING WITH EXAMPLE.

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- Then the methods are said to be *overloaded,* and the process is referred to as *method overloading.*
- Method overloading is one of the ways that Java implements polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Example:

```
class Overload
 {
   void test ( )
   {
     System.out.println("No parameters");
   }
  void test(int a)
  {
     System.out.println("a: " + a);
   }
   void test(int a, int b)
  {
     System.out.println("a: " + a + "b:" + b);
   }
   double test(double a)
   {
       System.out.println("double a: " + a);
       return a*a;
    }
}
```

```
class OverloadMain
{
  public static void main(String args[])
  {
     Overload ob = new Overload ();
     double result;
     ob.test( );
     ob.test(10);
     result = ob.test(123.25);
     System.out.println("Result :" + result);
   }
}
```

Output:

```
No parameters
a: 10
a :10 b: 20
double a: 123.25
Result: 15190.5625
```

EXPLAIN OVERLOADING CONSTRUCTORS.

Overloading Constructors

In addition to overloading normal methods, we can also overload constructor methods.

```
class Box {
            double width;
            double height;
            double depth;
            Box (double w, double h, double d)   // constructor used when all dimensions specified
            {
                    width = w;
                    height = h;
                    depth = d;
            }
            Box ( )   // constructor used when no dimensions specified
            {
```

```
                        width = -1; // use -1 to indicate
                        height = -1; // an uninitialized
                        depth = -1; // box
                }
                Box(double len)  // constructor used when cube is created
                {
                        width = height = depth = len;
                }
        }

class OverloadCons
 {
        public static void main(String args[])
        {
                // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                // get volume of cube
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```
Output
```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters:

So far we have only been using simple types as parameters to methods. It is both correct and common to pass objects to methods.

```
class Test
{
        int a, b;
        Test(int i, int j)
        {
                a = i;
                b = j;
        }
        // return true if o is equal to the invoking object
        boolean equals(Test o)
        {
                if(o.a == a && o.b == b) return true;
                else return false;
        }
}
class ParameterOb
{
        public static void main(String args[])
        {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equals(ob2));
                System.out.println("ob1 == ob3: " + ob1.equals(ob3));
        }
}
```
Output:
```
ob1 == ob2: true
ob1 == ob3: false
```

Argument Passing:

There are two ways to pass an argument to a subroutine.

1. **Call-by-value →** This method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
2. **Call-by-reference→** In this method, a reference to an argument (not the value of the argument) is passed to the parameter.

Note: When a simple type is passed to a method, it is done by the use of call-by-value. Objects are passed by the use of call-by-reference.

DEFINE RECURSION.

Recursion:

- Recursion is the process of defining something in terms of itself.
- It is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive.*
- The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N.*

```
class Factorial
{
        // this is a recursive function
        int fact(int n)
        {
                int result;
                if(n==1) return 1;
                result = fact(n-1) * n;
                return result;
        }
}
class Recursion
{
        public static void main(String args[])
        {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```
Output:
```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

- When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start.
- A recursive call does not make a new copy of the method. Only the arguments are new.

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

DISCUSS ABOUT STATIC METHODS.

Static methods

- ♣ The members that are declared as static are called static members.
- ♣ The static variables and static methods are often referred to as class variables and class methods.
- ♣ Static variables are used when we want to have a variable common to all the objects and accessed without using a particular object.
- ♣ Java creates only one copy for a static variable. It was called without using any objects.

Syntax:

```
static data type variable1, variable2…
```

Example:

float x = Math.sqrt(25.0);

The method sqrt is a class method defined in Math class.

Example:

```
class Usestatic
{
    static int a =3;
    static int b;
    static void meth (int x)
     {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
     }
            static
    {
          System.out.println("Static initialized");
           b= a * 4;
      }
     public static void main(String args[])
     {
            Meth(42);
      }
  }
```

Output

```
Static initialized
x = 42
a = 3
b = 12
```

Restrictions for static methods:

- ⏩ Static methods are called using class names.
- ⏩ They can only call other static methods.
- ⏩ They can only access static data.
- ⏩ They cannot refer to **this** or **super** in any way.

DEFINE FINAL VARIABLE AND METHODS.

Introducing final

All methods and variables can be overridden by default in subclasses. To prevent the subclasses from overriding the members of the super class, we can declare them as final using the keyword final as a modifier.

Ex:

```
final int SIZE = 100;           // final variable
final void show ( )             // final method
{
    ------
```

}

Final variables, behave like class variable and they do not take any space on individual objects of the class.

OVERRIDING METHODS

DISCUSS ABOUT OVERRIDING METHODS.

By defining a method in the subclass that has the same name, same arguments and the same return type as a method in the super class. Then when that method is called, the method define in the sub class is invoked and executed instead of one in the super class. This is known as *overriding*.

Example:

1
```
class super
{
        int x;
        super(int x)
        {
                this .x=x1;
         }
        void display( )     // method defined
        {
            System.out.println("x = "+x);
        }
}
```

2
```
class sub extends super
{
        int y;
        sub(int x, int y)
        {
                super(x);
            this.y = y;
        }
        void display()
        {
            System.out.println("x = " +x);
           System.out.println("y = " +y);
         }
 }
```

3
```
class test
{
  public static void main(String args[])
  {
        sub s1=new sub(30,20);
        s1.display();
  }
}
```

Strings

- String is a sequence of characters enclosed within double quotes.
- Java provides **three classes** to represent a sequence of characters:

Creating a String

There are two ways

1. String literal
2. Using new keyword

String literal

Assigning a String literal to a Stringinstance String str1 = "Welcome";

String str2 = "Welcome";

Using New Keyword

String str1 = newString("Welcome");

String str2 = new String("Welcome");

Methods:

public char charAt(int index)

This method requires an integer argument that indicates the position of the character that the method returns.This method returns the character located at the String's specified index. Remember, String indexes are zero-based—for example,

```
String x = "airplane";
System.out.println( x.charAt(2) ); // output is 'r'
```

public String concat(String s)

This method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke the method—for example,

```
String x = "book";
System.out.println( x.concat(" author") ); // output is "book author"
```

The overloaded + and += operators perform functions similar to the concat()method—for example,

```
String x = "library";
System.out.println( x + " card"); // output is "library card"
String x = "United";
x += " States"
System.out.println( x ); // output is "United States"
```

public boolean equalsIgnoreCase(String s)

This method returns a boolean value (true or false) depending on whether the value of the String in the argument is the same as the value of the String used to invoke the method. This method will return true even when characters in the String objects being compared have differing cases—for example,

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

public int length()

This method returns the length of the String used to invoke the method—for example,

```
String x = "01234567";
System.out.println( x.length() ); // returns "8"
```

public String replace(char old, char new)

This method returns a String whose value is that of the String used to invoke the method, updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—for example,

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') ); // output is  "oXoXoXoX"
```

public String substring(int begin)/ public String substring(int begin, int end)

The substring() method is used to return a part (or substring) of the String used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters to the end of the original String. If the call has two arguments, the substring returned will end with the character located in the nth position of the original String where n is the second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned String will be in the original String's 7 position, which is index 6. Let's look at some examples:

```
String x = "0123456789"; // the value of each char is the same as its index!
System.out.println( x.substring(5) ); // output is "56789"
System.out.println( x.substring(5, 8)); // output is "567"
```

public String toLowerCase()

This method returns a String whose value is the String used to invoke the method, but with any uppercase characters converted to lowercase—for example,

```
String x = "A New Java Book";
System.out.println( x.toLowerCase() ); // output is "a new java book"
```

public String toUpperCase()

This method returns a String whose value is the String used to invoke the method, but with any lowercase characters converted touppercase—for example,

```
String x = "A New Java Book";
```

```
System.out.println( x.toUpperCase() ); // output is"A NEW JAVA BOOK"
```

public String trim()

This method returns a String whose value is the String used to invoke the method, but with any leading or trailing blank spaces removed—for example,

```
String x = " hi ";
System.out.println( x + "x" ); // result is" hi x"
System.out.println(x.trim() + "x"); // result is "hix"
```

public char[ ] toCharArray( )

This method will produce an array of characters from characters of String object. For example

```
String s = “Java”;
Char [] arrayChar = s.toCharArray();  //this will produce array of size 4
```

public boolean contains(“searchString”)

This method returns true of target String is containing search String provided in the argument. For example-

```
String x = “Java is programming language”;
System.out.println(x.contains(“Amit”)); // This will print false
System.out.println(x.contains(“Java”)); // This will print true
```

Command Line Arguments in Java with Example

What is Command Line Argument in Java?

- **Command Line Argument in Java** is the information that is passed to the program when it is executed.
- The information passed is stored in the string array passed to the main() method and it is stored as a string.
- It is the information that directly follows the program’s name on the command line when it is running.

Example

While running a class **Demo**, you can specify command line arguments as

java Command Line Arguments in Java: Important Points

- Command Line Arguments can be used to specify configuration information while launching your application.
- There is no restriction on the number of java command line arguments.You can specify any number of arguments
- Information is passed as Strings.
- They are captured into the String args of your main method

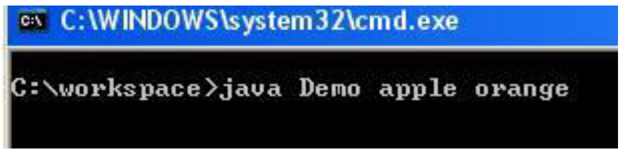
Example: To Learn java Command Line Arguments

Step 1) Copy the following code into an editor.

```
class Demo{
    public static void main(String b[]){
       System.out.println("Argument one = "+b[0]);
       System.out.println("Argument two = "+b[1]);
    }
}
```

Step 2) Save & Compile the code

Step 3) Run the code as java Demo apple orange.



```
C:\workspace>java Demo apple orange
```

Step 4) You must get an output as below

INHERITANCE

DEFINE INHERITANCE. EXPLAIN THE TYPES OF INHERITANCE.
WHAT IS INHERITANCE AND HOW DOES IT HELP US TO CREATE NEW CLASSES QUICKLY?

- ★ Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- ★ In the terminology of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***.
- ★ Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.
- ★ The mechanism of deriving a new class from an old one is called inheritance. The old class is known as base class (or) super class (or) parent class and the new one is called the sub class (or) derived class (or) child class.

Types of inheritance:

- ❖ Single inheritance (only one super class)
- ❖ Multiple inheritances (several base classes)
- ❖ Hierarchical inheritance (one super class, many subclasses)
- ❖ Multiple inheritances (derived from a derived class)

INHERITANCE: EXTENDING CLASSES

Inheritance Basics:

To inherit a class, we simply incorporate the definition of one class into another by using the **extends** keyword. The keywords extends signifies that the properties of the superclass name are extended to the subclass name.

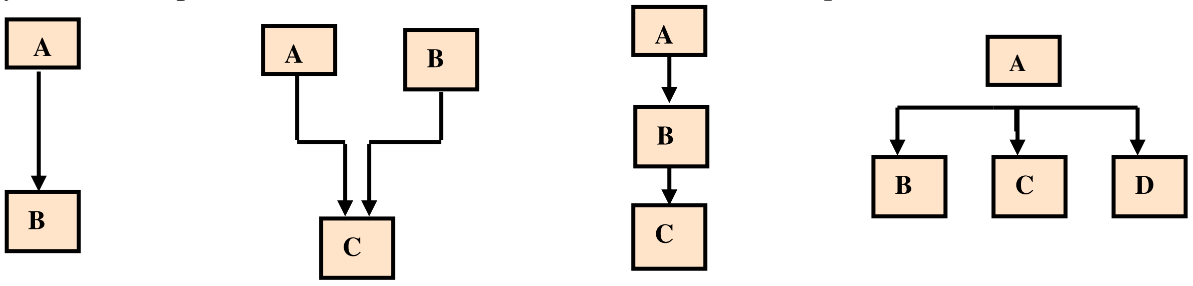
Defining a subclass:

```
class subclass extends superclass
{
  Variables declaration;
   Methods declaration;
}
```

Example:

```
class B extends A       // A → Superclass   B→ Subclass
{
   -------
   -------
}
```

Java does not directly implement multiple inheritances. This concept is implemented using a secondary inheritance path in the form of interfaces. No class can be a superclass of itself.



Single inheritance **Multiple inheritance** **Multilevel inheritance** **Hierarchical inheritance**

Member Access and Inheritance:

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
class A
{
	int i; // public by default
	private int j; // private to A
	void setij(int x, int y)
	{
		i = x;
		j = y;
	}
}
class B extends A			// A's j is not accessible here.

{
	int total;
	void sum()
	{
		total = i + j; // ERROR, j is not accessible here
	}
}
```

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. For example, consider the following:

```
class RefDemo
{
	public static void main(String args[])
	{
		BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
		Box plainbox = new Box();
		double vol;
		vol = weightbox.volume();
		System.out.println("Volume of weightbox is " + vol);
		System.out.println("Weight of weightbox is " + weightbox.weight);
		System.out.println();
			// assign BoxWeight reference to Box reference
		plainbox = weightbox;
		vol = plainbox.volume(); // OK, volume() defined in Box
		System.out.println("Volume of plainbox is " + vol);
	}
}
```

Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.

When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

***Subclass constructor**: (Using super)*

A subclass constructor is used to construct the instance variables of both the subclass and the super class .The subclass constructor uses the keyword super to invoke the constructor method of the super class.

The keyword super is used in the following conditions.

- Super may only be used within a subclass constructor method.
- The call to super class constructor must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of the instance variable in the super class.
- **super** has two general forms.
  1. The first calls the superclass' constructor.
  2. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors:

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(parameter-list);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass constructor. **super( )** is called using the appropriate arguments.

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A
{
        int i;
}
// Create a subclass by extending class A.
class B extends A
{
        int i; // this i hides the i in A
        B(int a, int b)
        {
                super.i = a; // i in A
                i = b; // i in B
        }
        void show()
        {
                System.out.println("i in superclass: " + super.i);
                System.out.println("i in subclass: " + i);
        }
}
class UseSuper
 {
        public static void main(String args[]
```

```
This program displays the following:
i in superclass: 1
i in subclass: 2
```

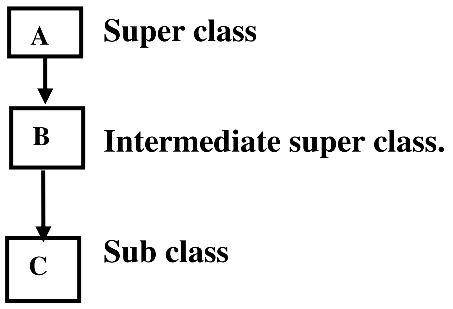
```
        {
                B subOb = new B(1, 2);
                subOb.show();
        }
    }
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass.

Creating a Multilevel Hierarchy

EXPLAIN MULTILEVEL INHERITANCE IN JAVA.

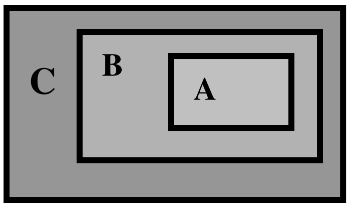
Deriving a class from a super class allows us to build a chain of classes. We can build hierarchies that contain as many layers of inheritance as we like. For example, given three classes called **A**, **B** and **C. C** can be a subclass of **B**, which is a subclass of **A**. In this case, **C** inherits all aspects of **B** and **A**.



The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as ***inheritance path*.**

```
class A
{
    ---------
}
class B extends A     // first level
{
    ---------
}
class C extends B     // second level
{
    ---------
}
```

This process may be extended to any number of levels. The class C can inherit the members of both A and B as:



C contains B which contains A

WHEN CONSTRUCTORS ARE CALLED?

★ In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
★ Further, since **super( )** must be the first statement executed in a subclass constructor, this order is the same whether or not **super( )** is used.
★ If **super( )** is not used, then the default or parameter less constructor of each superclass will be executed.

Method Overriding:

★ In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
★ When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

<u>Dynamic method dispatch:</u>

★ It is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
★ It is important because this is how Java implements run-time polymorphism.
★ When different types of objects are referred to, different versions of an overridden method will be called.
★ In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
★ Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different version of the method are executed.

Why Overridden Methods?

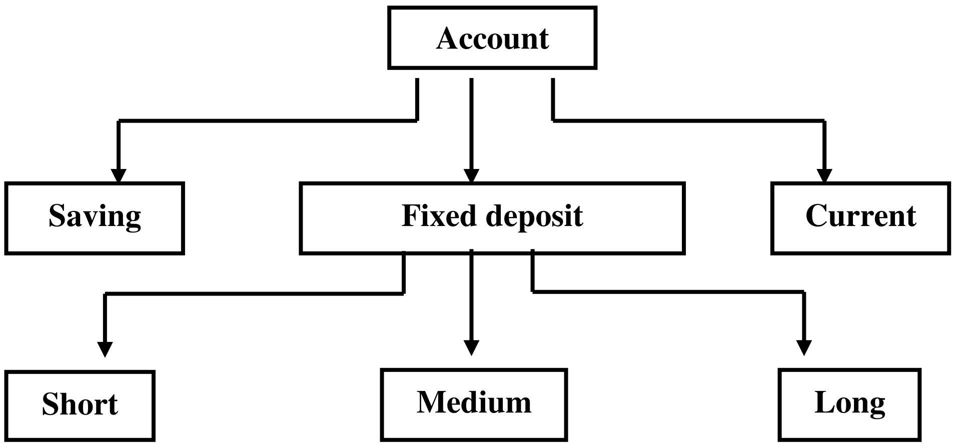
Overridden methods allow Java to support run-time polymorphism.

Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

EXPLAIN ABOUT HIERARCHICAL INHERITANCE.

In hierarchical inheritance, the base class consists of many derived class.



UNIT-III

PACKAGES AND INTERFACES

EXPLAIN THE CONCEPT OF PACKAGE IN JAVA. DEFINE PACKAGE.

INTRODUCTION:

- ◆ Packages are containers for classes that are used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- ◆ Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts:
    - A single package statement (optional)
    - Any number of import statements (optional)
    - A single public class declaration (required)
    - Any number of classes private to the package (optional)
- ◆ Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- ◆ The package is both a naming and a visibility control mechanism.
- ◆ We can define classes inside a package that are not accessible by code outside that package.
- ◆ We can also define class members that are only exposed to other members of the same package.

WRITE THE BENEFITS OF PACKAGE IN JAVA.

Benefits:

- ★ The classes contained in the packages of other programs can be easily reused.
- ★ Classes can be unique compared with classes in other packages. Two classes in two different packages can have the same name.
- ★ Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
- ★ Packages also provide a way for separating "design" from "coding".

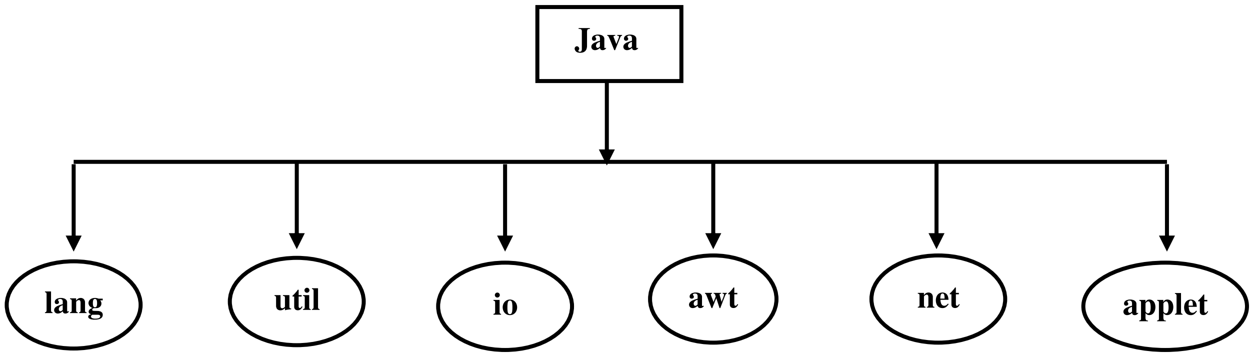
Java packages are classified into two types

1) Java API packages
2) User defined package

WRITE NOTES ON JAVA API PACKAGES?

JAVA API PACKAGES:

Java API provides a large number of classes grouped into different packages according to functionality.



Package name	***Contents***
java.lang	These classes that java compiler itself uses and automatically imported. It includes classes for primitive types, string, math functions, threads and exceptions.

java.util	Language utility classes such as vectors, hash tables, random numbers, date. etc.
java.io	Input/ output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
Java.applet	Classes for creating and implementing applets.

<u>CREATING A PACKAGE</u>

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.
- While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

Syntax: package pkg;   where ***pkg*** → name of the package.
Ex: package MyPackage;

- More than one file can include the same **package** statement.
- The **package** statement simply specifies to which package the classes defined in a file belong.
- It does not exclude other classes in other files from being part of that same package.
- Java uses file system directories to store packages.
- To create a hierarchy of packages, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:

Syntax: package *pkg1*[.*pkg2*[.*pkg3*]];
Ex: package java.awt.image;

<u>Finding Packages and CLASSPATH</u>

1. By default, the Java run-time system uses the current working directory.
2. We can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

If we are working on source code in a directory called c:\myjava, then set the CLASSPATH to

.;c:\myjava;c:\java\classes

<u>A Short Package Example</u>

```
package MyPack;
class Balance {
        String name;
        double bal;
        Balance(String n, double b) {
                name = n;
                bal = b;
        }
        void show() {
                if(bal<0)
                System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}
class AccountBalance {
        public static void main(String args[]) {
                Balance current[] = new Balance[3];
                current[0] = new Balance("Ball", 123.23);
                current[1] = new Balance("Pen", 157.02);
                current[2] = new Balance("Pencil", -12.33);
                for(int i=0; i<3; i++) current[i].show();
}
    }
```

Note:

1. Save this file as **AccountBalance.java**, and put it in a directory called **MyPack.** Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then execute the **AccountBalance** class, using the command line: **java MyPack.AccountBalance**
2. We cannot use this command line: **java AccountBalance** because **AccountBalance** is qualified with its package name.

EXPLAIN THE TYPES OF VISIBILITY CONTROLS IN JAVA.

Access protection

Apply visibility modifiers to the instance variables and methods. It is also known as *access modifiers.*

Three types of visibility modifiers:

- ♦ Public
- ♦ Private
- ♦ Protected

Four categories of visibility for class members:

- ★ Subclasses in the same package.
- ★ Non-subclasses in the same package.
- ★ Subclasses in different packages.
- ★ Classes that are neither in the same package nor subclasses.

Public Access:

Any variable or method is visible to the entire class by simply declaring it as public. To make it visible to all the classes outside this class declare the variable or method as public.

Example:

```
public int x;
public void get( )
{
   -----
   -----
}
```

A variable or method declared as public has the widest possible visibility and accessible everywhere.

Friendly Access:

- When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.
- Public modifier makes fields visible in all classes, regardless of their packages.
- Friendly access makes fields visible only in the same package, but not in other packages. A package is a group of related classes stored separately.

Protected Access:

It makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages.

Private Access:

They are accessible only within their own class. A method declared as private behaves like a method declared as final. It prevents the method from being sub-classed.

Private Protected Access:

A field can be declared with 2 keywords private and protected.

private protected int cno;

It makes the fields visible in all subclasses regardless of what package they are in.

Visibility of fields in a class

DISCUSS THE DIFFERENT LEVELS OF ACCESS PROTECTION AVAILABLE IN JAVA.

Access / Access modifiers locations	***Public***	***Protected***	***Friendly (default)***	***Private protected***	***Private***
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same packages	Yes	Yes	Yes	No	No
Sub classes in other packages	Yes	Yes	No	No	No
Non sub classes in other packages	Yes	No	No	No	No

Rules for applying appropriate access modifiers:

- Use ***public*** if the fields is to be visible everywhere.
- Use ***protected*** if the field is visible everywhere in the current package and also subclasses in other packages.
- Use ***default*** if the field is to be visible everywhere in the current package only.
- Use ***private*** protected if the field is to be visible only in subclasses, regardless of packages.
- Use ***private*** if the field is not to be visible anywhere except in its own class.

ACCESSING PACKAGES:

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

General form:

import package1 [.package2] [.package3] [...].classname;

package1 → top level package, ***package2*** → package that is inside the package1 and so on. The statement must end with a semicolon;

All the members of the class can be directly accessed using class name or its objects directly without using the package name.

Ex: import package name.*;

The star (*) indicates that the compiler should search the entire package hierarchy when it encounters a class name.

USING A PACKAGE: Sample program importing classes from other packages

```
package pack1;
public class classA
{
        public void displayA( )
        {
                System.out.println (“First package”);
        }
}
```

```
import pack1.classA;
import pack2.*;
 class packTest
{  public static void main String(a[]))
   {
        classA objA = new classA( );
        objA.displayA( );
   } }
```

- This source file should be named as classA.java and stored in the subdirectory pack1.
- Now compile this java file. The resultant classA.class will be stored in the same directory.

Importing package1 to the main method

```
import package1.classA;
import packTest
{       public static void main(String[] args)
```

```
        {
                classA objectA=new classA( );
                objectA.displayA( );
        }
}
```

HIDING CLASSES:

A package are imported using asterisk (*), all public classes are imported. We may prefer to "not import" certain classes. Those classes are hidden from accessing from outside of the package. Such classes should be declared as "not public ".

Ex:

```
package p1;
public class X          // public class available outside
{
        //body of X
}
class Y
{
        //body of Y     //not public, hidden
}
```

Here, the class Y which is not declared as public is hidden from outside of the package p1.

Ex:

```
import p1.*;
X objectx = new X( );        //OK → class X is available here
Y objectY = new Y( );        //Not OK →Y is not available
```

The java compiler generates an error message for this code because the class Y, which has not been declared as public.

INTERFACES

DEFINE INTERFACE. HOW TO EXTEND INTERFACES IN JAVA PROGRAM? INTRODUCTION:

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

Java cannot have more than one superclass. It can implement more than one interface.

DEFINING AN INTERFACE

An interface is basically a kind of class. The interface defines only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.

General form:

```
access interface iname
{
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        return-type method-name(parameter-list);
}
```

```
interface iname
{
   Variable declaration;
   Methods declaration;
}
```

Here, **access** is either **public** or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. **iname** is the name of the interface, and can be any valid identifier.

- Variables can be declared inside the interface declarations.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- They must also be initialized with a constant value.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Variables are declared as follows:

Static final type varname = value;

All variables are declared as constants. Method declaration will contain a list of methods without the body statements.

Return-type method name (parameter-list)

Example:

```
interface Item
{
            static final int code = 700;
            static  final  String  name  =  ”java”;
            void display();
            float  compute(float  x,  float  y);  //  The  method  declaration  simply  ends  with  a
 semicolon.
}
```

IMPLEMENTING INTERFACES

Interfaces are used as “super classes” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface.

Syntax: 1

```
        class classname implements Interfacename
        {
                Body of class name
        }
```

Here, the class classname ‘implements’ the interface interfacename.

Syntax: 2

```
 class classname extends superclass implements interface1, interface2
 {
 Body of class name
 }
```

When a class implements more than one interface, they are separated by a comma.

Example: Implementing interfaces

```
interface area                                      // interface defined
{
final   static   float   pi   =   3.14f;   final
compute(float x, float y);
}
        class rectangle implements area             // interface implemented
{
                public float compute(float x, float y)
{
return (x*y);
                }
}
```

Accessing interface variables:

Interfaces can be used to declare a set of constants that can be used in different classes.

Example:

```
class student
{
		int rno;
		void getdata(int n)
		{
			rno = n;
		}

		void putdata()
		{
			System.out.println("\tRoll No.    : " + rno);
		}
}
class mark extends student
{
		float m1, m2;
		void getmark(int mm1, int mm2)
		{
			m1 = mm1;
			m2 = mm2;
		}
		void putmark()
		{
			System.out.println("\t   English   :   "+   m1);   //   60
			System.out.println("\t Tamil : "+ m2); // 80
		}
}
interface extra
{
		float sports = 6.0f;
		void put();
}
class Result extends mark implements extra
{
		float		total;
		public void put()
		{
			System.out.println("\t Sprots marks : "+ sports);      // 6.0
		}
		void display()
		{
			total = m1 + m2 + sports;
			putdata();
			putmark();
			put();
```

```
                        System.out.println("total score : "+total);     // 146.0
                }
}
class inher
{               public static void main (String arg[])
                {
                        Result  obj  =  new
                        Result();
                        obj.getdata(1234);
                        obj.getmark(60,
                        80); obj.display();
                }
}
```

EXTENDING INTERFACES

An interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the super interface in the manner similar to subclasses. This is achieved using the keyword extends as follows below:

Syntax:

```
interface name2 extends name1
{
        body of name 2;
}
```

Example:

```
interface strdisplay
{
    int no=700;
    String name="java";
}
interface str extends strdisplay
{
   void display( );
}
```

Note: *Interface Item inherit both the constant code and name into it. It is treated as final and static.*

Several interfaces are combined into a single interface

Example:

```
interface A
{
}
interface B extends A                    //  B  now  includes  meth1()  and
meth2() -- it adds meth3().
{
        void meth3();
}
class MyClass implements B                      // This class must implement
all of A and B
{
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }
        public void meth2()
        {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
                System.out.println("Implement meth3().");
        }
}
class IntExtend
{
        public static void main(String arg[])
        {
                MyClass ob = new
                MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

Trying to remove the implementation for meth1( ) in **MyClass** will cause a compile- time error. Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interface

EXCEPTION HANDLING

Introduction:

A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even cause the system to crash.

DEFINE EXCEPTION.

1) An exception is an indication of a problem that occurs during a program's execution. A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
2) Exception is a condition that is caused by a run-time error in the program. It can be generated by the Java run-time system, or they can be manually generated by our code.

DEFINE EXCEPTION HANDLING.

To continue the program with the execution of the remaining code, try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This is known as exception handling.

ERROR HANDLING AND EXCEPTION HANDLING

Exception-Handling Fundamentals:

- ❖ Exception handling enables programmers to create applications that can resolve (or handle) exceptions.
- ❖ Handling an exception allows a program to continue executing as if no problem had been encountered.
- ❖ A more severe problem could prevent a program from continuing normal execution, instead requiring it to notify the user of the problem before terminating in a controlled manner.
- ❖ Exception handling enables programmers to remove error-handling code from the "main line" of the program's execution, improving program clarity and enhancing modifiability.
- ❖ When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. The exception is caught and processed.
- ❖ Exceptions can be generated by the Java run-time system, or they can be manually generated by the user code.
- ❖ Exceptions thrown by Java violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.
- ❖ Java exception handling is managed via five keywords:
  - ➢ **try**
  - ➢ **catch**
  - ➢ **throw**
  - ➢ **throws**
  - ➢ **finally**

Program statements that we want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. The program code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system.

To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that must be absolutely executed before a method returns is put in a **finally** block.

General form of an exception-handling block:

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

Here, *ExceptionType* is the type of exception that has occurred.

EXCEPTION TYPES AND HIERARCHY

LIST SOME OF THE MOST COMMON TYPES OF EXCEPTIONS THAT MIGHT OCCUR IN JAVA. GIVE EXAMPLES.

Types of exceptions:

It can be broadly classified into two categories. They are:

- Compile-time errors.
- Run-time errors

Compile-time errors:

All syntax errors will be declared and displayed by the java compiler and therefore these errors are known as compile-time errors.

Whenever the compiler displays an error, it will not create the .class file. Most of these errors are occurred by typing mistakes.

Most common problems are:

- Missing semicolons
- Missing brackets in classes and methods
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible type in assignments/ initialization
- Bad references to objects
- Use of = in place of == operator.

Example: *The following message will be displayed*

```
class Error
{
        public static void main(String a[ ])
        {
                System.out.println("Hello Java ")
        }
}
```

```
Error1.java:7: ';' expected
System.out.println("Hello Java ")
^
1 error
```

Other errors may encounter are related to directory paths. An error such as

javac: command not found ← It remains to set the path correctly.

WRITE SHORT NOTES ON RUN-TIME ERRORS.

Run-Time Errors:

A program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Most common run-time errors are:

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

Example:

```
class error
{
        public static void main(String args[])
        {
                int a = 10;
                int b =  10;
                int  c = 10;
                int x = a / (b-c);                      // division by zero
                System.out.ptinrln (“x= “ +x);
                int y = a / (b+c);
                System.out.println (“y= “+y);
        }
}
```

Error message:

```
        java.lang.ArithmeticException: / by zero at error2.main (error2.java: 10).
```

Java run-time tries to execute a division by zero, it generates an error condition.

STATE THE ACTIONS THAT ARE PEROFMED BY ERROR HANDLING CODE.

The error handling code that performs the following tasks:

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exceptions)

The error handling code basically consists of two segments; one to detect errors and to throw exception and the other is to catch exceptions and to take appropriate actions.

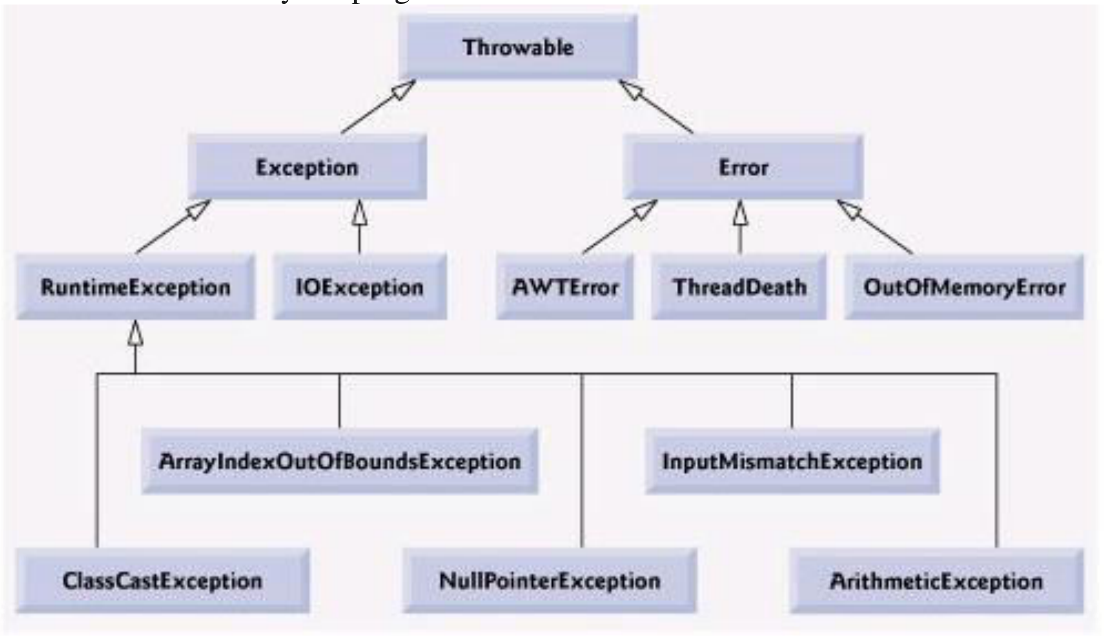
Common java exceptions

Exception Type	Cause of Exception
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexed
ArrayStoreException	Caused when a program tries to store the wrong type data in an array.
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file.
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails.
OutOfMemoryException	Caused when there’s not enough memory to allocate a new object.
SecurityException	Caused when an applet tries to perform an action not allowed by the browser’s security settings.
StackOverflowException	Caused when the system runs out of stack space.
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string.

<u>Exception Hierarchy:</u>

All exception types are subclasses of the built-in class Throwable. Throwable is at the top of the exception class hierarchy. It has two subclasses

1. Exceptions → this class is used for exceptional conditions that user programs should catch.
2. Error → it defines exceptions that are not expected to be caught under normal circumstances by the program.



Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

<u>Uncaught Exceptions:</u>

Any exception that is not caught by the program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

TRY & CATCH BLOCK

WHAT IS THE USE OF TRY AND CATCH IN JAVA?

- ❖ Handling an exception by ourselves provides two benefits.
    - First, it allows to fix the error.
    - Second, it prevents the program from automatically terminating.
- ❖ To guard against and handle a run-time error, simply enclose the code inside a **try** block.
- ❖ Immediately following the **try** block, includes a **catch** clause that specifies the exception type to catch.

Example: The following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

1

```
class Sample
{
public static void main(String args[])
{
  int d, a;
  try
  {      // monitor a block of code.
    d = 0;
    a = 42 / d;
    System.out.println("This will not be printed.");
  }
```

2

```
catch (ArithmeticException e)
{
          System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following **output**:

```
Division by zero.
After catch statement.
```

- Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.
- A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try**. The statements that are protected by **try** must be surrounded by curly braces. We cannot use **try** on a single statement.

Handle an exception and move on

1

```
import java.util.Random;
class HandleError
{
        public static void main(String args[])
        {
                int a=0, b=0, c=0;
                Random r = new Random();
                for(int i=0; i<32000; i++)
                {
                    try {
                            b = r.nextInt();
                            c = r.nextInt();
                            a = 12345 / (b/c);
                    }
```

2

```
                    catch (ArithmeticException e)
                    {
                        System.out.println("Division by zero.");
                        a = 0;    // set a to zero and continue
                    }
                    System.out.println("a: " + a);
                }  // for loop
        }  // main method
}  // class
```

EXAPLIN MULTIPLE CATCH IN JAVA.

Multiple catch Clauses

- If more than one exception is raised by a single piece of code then specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.
- When we use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Multiple catch statements

1

```
class MultiCatch
{
  public static void main(String args[])
  {
    try {
            int a = args.length;
            System.out.println("a = " + a);
             int b = 42 / a;
             int c[] = { 1 };
             c[42] = 99;
         }
   catch(ArithmeticException e)
   {
     System.out.println("Divide by 0: " + e);
   }
```

2

```
catch(ArrayIndexOutOfBoundsException e)
{
  System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Output

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob:
java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks
```

Nested try Statements:

- The **try** statement can be nested; it can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, then the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until the entire nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

```
class NestTry                                        (1)
{
   public static void main(String args[])
   {
      try {
               int a = args.length;
               int b = 42 / a;
               System.out.println("a = " + a);
               try {
                     if(a==1)
                        a = a / (a-a);
                     if(a==2)
                     {
                        int c[] = {1 };
                        c[42] = 99;
                     }
                    }
catch(ArrayIndexOutOfBoundsException e)
{
  System.out.println("Array index oobs: " + e);
}
}
catch(ArithmeticException e)                          (2)
{
  System.out.println("Divide by 0: " + e);
}
}
}
```

Output

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by
zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by
zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException
```

USE OF throw, throws AND finally

Using throw:

- ❖ It is possible for the program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

  throw *ThrowableInstance*;

  Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- ❖ Simple types, such as int or char, as well as non-**Throwable** classes, such as String and Object, cannot be used as exceptions.
- ❖ There are two ways to obtain a **Throwable** object:
  1. Using a parameter into a **catch** clause.
  2. Creating one with the **new** operator.
- ❖ The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- ❖ If try statement find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on.
- ❖ If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.
- ❖ All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.

  throw new NullPointerException("demo");

Here, **new** is used to construct an instance of **NullPointerException**. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

1

```
class ThrowDemo
{
  static void demoproc()
  {
     try
     {
        throw new NullPointerException("demo");
     }
     catch(NullPointerException e)
     {
        System.out.println("inside demoproc.");
        throw e;
       }
    }
```

2

```
public static void main(String args[])
{
    try {
           demoproc();
        }
    catch(NullPointerException e)
     {
        System.out.println("Recaught: " + e);
      }
    }
  }
```

Output

```
Caught inside demoproc.
Recaught:java.lang.NullPointerException: demo
```

Using throws:

A **throws** clause lists the types of exceptions that a method might throw. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

*General form of a method declaration that includes a **throws** clause:*

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Tries to throw an exception that it does not catch because of no throws clause	**CORRECT PROGRAM**

class ThrowsDemo
{
 static void throwOne()
 {
  System.out.println("Inside throwOne.");
  throw new IllegalAccessException("demo");
 }
public static void main(String args[])
{
  throwOne();
}
}
 	
INCORRECT
	class ThrowsDemo
	{
	static void throwOne() throws IllegalAccessException
	{
	    System.out.println("Inside throwOne.");
	    throw new IllegalAccessException("demo");
	}
	public static void main(String args[])
	{
	  try { throwOne(); }
	  catch (IllegalAccessException e)
	  { System.out.println("Caught " + e); }
	}
	}
	
	Output
	inside throwOne
	caught java.lang.IllegalAccessException: demo

Using finally statement:

WHAT IS THE USE OF FINALLY BLOCK? WHEN AND HOW IS IT USED?

- ❖ The finally statement used to handle an exception that is not caught by any of the previous statement. The finally block can be used to handle any exception generated within a try block.
- ❖ The finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown.
- ❖ When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown.

1

```
class FinallyDemo {
static void procA()
{
   try {
   System.out.println("inside procA");
   throw new RuntimeException("demo");
   }
  finally {
    System.out.println("procA's finally");
  }
}
// Return from within a try block.
static void procB()
{
 try {
    System.out.println("inside procB");
    return;
   }
  finally {
      System.out.println("procB's finally");
   }
}
```

2

```
static void procC()
{
  try {
  System.out.println("inside procC");
  }
 finally {
    System.out.println("procC's finally");
  }
}
public static void main(String args[])
{
   try {
         procA();
    }
   catch (Exception e) {
        System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

<u>Output</u>

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

PROGRAMMER DEFINED EXCEPTIONS

Throwing our own Exception:

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throws as follows:

throws new Throwable_subclass;

Example:

```
throw new ArithmeticException ( );
throw new numberFormatException( );
```

***Example program**: Throwing our own exception*

```
import java.lang.Exception;
class MyException extends Exception
{
         MyException (String message)
        {
                 super (message);
        }
}



class TestMyException
{
        public static void main(String args[])
        {
            int x=5 , y=1000;
            try
            {
                float z = (float) x / (float) / y;
                if(z < 0.01)
                {
                        throw new MyException ("Number is too small");
                }
             }
             catch (MyException e)
             {
                        System.out.println ("caught my exception ");
                        System.out.println (e.getMessage ( ) );
             }
             finally
             {
                     System.out.println("I am always here");
              }
    }
}
```

Output:

```
Caught my exception
Number is too small
I am always here
```

Chained exception:

The chained exception feature allows to associate another exception with an exception. This second exception describes the cause of the first exception.

MULTITHREADED PROGRAMMING

INTRODUCTION

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

There are two distinct types of multitasking:

- ❖ process-based
- ❖ Thread-based.

Process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a ***thread-based*** multitasking environment, the thread is the smallest unit of dispatch able code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

THREAD MODEL

WRITE A SHORT NOTE ON JAVA THREAD MODEL.

Threads exist in several states. A thread can be *running.* It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended,* which temporarily suspends its activity. A suspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

}

LIFE CYCLE OF A THREAD

EXPLAIN IN DETAIL ABOUT THE LIFE CYCLE OF A THREAD.

Life cycle of a thread:

The life cycle of a thread consists of the following states:

- New born state
- Runnable state
- Running state
- Blocked state
- Dead state

A Thread is always in one of these five states. It can move from one state to another via a variety of ways.

Newborn state:

Creation of thread object is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

➤ ***Runnable state:***

The runnable state, the thread is ready for execution and is waiting for the availability of the processor. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner.

The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as ***time-slicing.***

Yield () method is used to relinquish control from one thread to another of equal priority.

Running state:

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it preempted by a higher priority thread.

A running thread may relinquish its control in one of the following situations:

- ❖ It has been suspended using suspend ( ) method. A suspended thread can be revived by using the resume ( ) method.
- ❖ It has been made to sleep. To sleep a thread sleep (time) method is used, when time is in millisecond. The thread re-enters the runnable state as soon as this time period is elapsed.
- ❖ It has been told to wait until some event occurs. This is done using the wait ( ) method. The thread can be scheduled to run using the notify ( ) method.

Blocked state:

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. A blocked thread is considered “not runnable” but not dead and therefore fully qualified to run again.

Dead state:

Every thread has a life cycle. A running thread ends its life when it has completed executing its run( ) method. It is a natural death. A thread can be killed as soon it is born, or while it s running, or even when it is in “not runnable” condition.

CREATING A THREAD

HOW TO CREATE A THREAD WITH AN EXAMPLE?

We can create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- ✓ Implement the **Runnable** interface.
- ✓ Extend the **Thread** class, itself.
- ✓

IMPLEMENTING RUNNABLE:

The runnable interface declares the run () method that is required for implementing threads in our programs.To do this, we must perform the steps listed below.

- ❖ Declare the class as implementing the runnable interface.
- ❖ Implement the run() method.
- ❖ Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
- ❖ Call the thread start() method to run the thread.

EXTENDING THE THREAD

We can make our class runnable as a thread by extending the class java.lang.Thread.
This gives us access to all the thread methods directly.
It includes the following steps:

- Declare the class as extending the thread class.
- Implement the run () method that is responsible for executing the sequence of code that the thread will execute.
- Creating a thread object and call the start () method.

Declaring the class:

The thread class can be extended as follows:

```
class Mythread extends Threads
{
        ------------
}
```

Implementing the run () method:

The run ( ) method has been inherited by the class Mythread. The basic implementation of run( ) will look like this:

```
Public void run()
{
        ------------
}
```

Starting New Thread:

To create and run our thread class,

```
Mythread athread=new Mythread( );
athread.start( ); // invokes run( ) method
```

From the first line the thread is in a newborn state. The second line calls the start() method causing the thread to move into the runnable state. The java runtime will schedule the thread to run by invoking its run () method. Now the thread is said to be in running state.

Ex:

Creating threads using the thread class Class A extends Thread

```
                {
        Public void run ( )
                        {
            for (int i=1;i<5;i++)
                        System.out.println(i);
                                }
        }
Class B extends Thread
        {
        Public void run ( )
        {
            for (int j=1;j<5;j++)
                {
                System.out.println(j);
                }
            }
        }
    class Threadtest
        {
        public static void main(String a[])
        {
```

```
                    new A( ).start( ) ;
                    new B( ).start( );        }         }
```

Output: java Threadtest

```
                    from thread A =1
                    from thread A =2
                    from thread B =1
                    from thread B =2
                    from thread A =3
                    from thread A =4
                    from thread B =3
                    from thread B =4
```

MULTIPLE THREADS

We are using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
String  name;  Thread
t;
NewThread(String threadname)
                    {
                    name = threadname;
                    t     =     new     Thread(this,     name);
                    System.out.println("New thread: " + t);
                    t.start();
                    }
// This is the entry point for thread. public
void run( )
{
                 try
                    {
                    for(int i = 5; i > 0; i--)
                              {
                              System.out.println(name    +    ":    "    +    i);
                              Thread.sleep(1000);
                              }
                    }
catch (InterruptedException e)
                    {
                    System.out.println(name + "Interrupted");
                    }
System.out.println(name + " exiting.");
}
 }
 class MultiThreadDemo
{
public static void main(String args[])
                    {
                    new NewThread("One"); // start threads
                    new NewThread("Two");
```

```
                new    NewThread("Three");
                try
                           {
                           System.out.println("Main thread Interrupted");
                           }
                System.out.println("Main thread exiting.");





          }
          t
          h
          r
          e
          a
          d
          s

          t
          o

          e
          n
          d

          T
          h
          r
          e
          a
          d
          .
          s
          l
          e
          e
          p
          (
          1
          0
          0
          0
          0
          )
          ;
          }
catch
(Interrup
tedExce
ption e)
```

```
New thread: Thread[One,5,main] New thread:
Thread[Two,5,main] New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
```

UNIT-IV

APPLETS

EXPLAIN APPLETS IN BRIEF. (OR) DISCUSS ABOUT APPLETS IN JAVA.
DEFINE APPLET.

- Ap*plets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document
- They can be transported over the internet from one computer to another and run using the applet viewer or any web browser that support the java program.
- It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation & play interactive games.
- A web page contain simple text or a static image but also a java applet which, when run, can produce graphics, sounds and moving images.

Steps involved in developing and testing an applets are:

First ensure that java is installed properly and also ensures that either the java applet viewer or a java enabled browser is available.

Steps:

- Building an applet code (.java file)
- Creating an executable applet (.class file)
- Designing a WebPages using HTML tags.
- Preparing <APPLET> tag.
- Incorporating <APPLET> tag into the web page.
- Creating HTML files.
- Testing the applet code

Ex:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet
{
        public void paint(Graphics g)
        {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

Explanation to the example:

This applet begins with two **import** statements.

- The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes.
- The second import statement imports the applet package, which contains the class Applet.

Every applet created must be a subclass of Applet. The next line in the program declares the class SimpleApplet. This class must be declared as public, because it will be accessed by code that is outside the program.

Four of these methods — init( ), start( ), stop( ), and destroy( )—are defined by Applet. Another method paint( ) is defined by the AWT Component class.

paint ():

- It is a method is defined by the AWT and must be overridden by the applet.
- It is called each time that the applet must redisplay its output and the applet window can be minimized and then restored.
- It is also called when the applet begins execution.
- It has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running.

drawstring():

- ❖ It is a member of the Graphics class.
- ❖ It outputs a string beginning at the specified X,Y location.
- ❖ It has the following general form:

void drawString(String message, int *x*, int *y*)

Here, message is the string to be output beginning at *x*,*y*. In a Java window, the upper-left corner is location 0, 0. The call to drawString( ) in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.

Running SimpleApplet involves:

There are two ways to run an applet:

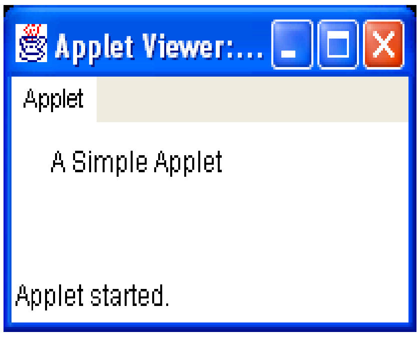
- Executing the applet within a Java-compatible Web browser.
- Using an applet viewer, such as the standard SDK tool, appletviewer. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test the applet.
- To execute an applet in a Web browser, write a short HTML text file that contains the appropriate APPLET tag.

  <applet code="SimpleApplet" width=200 height=60>
  </applet>
- The width and height statements specify the dimensions of the display area used by the applet. We can execute your browser and then load this file, which causes SimpleApplet to be executed.
- To execute SimpleApplet with an applet viewer, we may also execute the HTML file shown earlier. For example, if the preceding HTML file is called RunApp.html, then the following command line will run SimpleApplet:

  C:\>appletviewer RunApp.html

Example:

```
import java.awt.*;
import java.applet.*;
/*      <applet   code="SimpleApplet"   width=200   height=60>
</applet>    */
public class SimpleApplet extends Applet
{
  public void paint(Graphics g)
  {
        g.drawString("A Simple Applet", 20, 20);
  }
}
```



A Simple Applet
Applet started.

In general, we can quickly iterate through applet development by using these three steps:

- ➢ Edit a Java source file.
- ➢ Compile the program.
- ➢ Execute the applet viewer, specifying the name of the applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute the applet.

The window produced by SimpleApplet, as displayed by the applet viewer, is

Applet Initialization and Termination:

When an applet begins, the AWT calls the following methods, in this sequence:

1) init( ) → Used to initialize variables and is called only once during the run time of the applet.
2) start( ) → It is called to restart an applet after it has been stopped. It is called each time an applet's HTML document is displayed onscreen.
3) paint( ) → It is called each time the applet's output must be redrawn.

When an applet is terminated, the following sequence of method calls takes place:

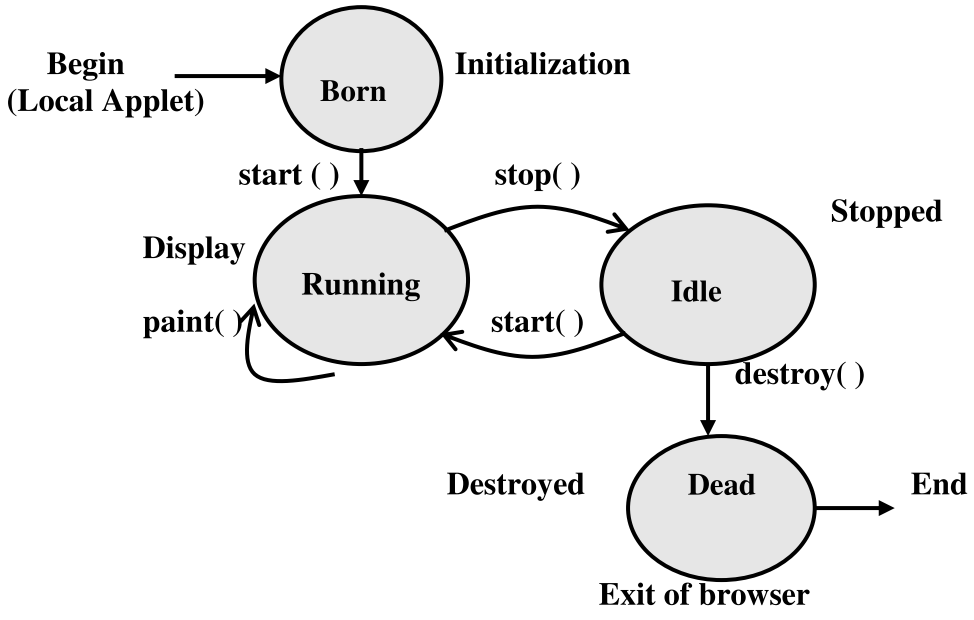
1) stop( ) → It is called when a web browser leaves the HTML document containing the applet. Used to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.
2) destroy( ) → It is called when the environment determines that your applet needs to be removed completely from memory.

APPLET LIFE CYCLE

EXPLAIN ABOUT APPLET LIFE CYCLE?

Every Java applet inherits a set of default behaviors from the Applet class. The applet state includes:

- Born or initialization state
- Running state
- Idle state
- Dead or destroyed state



⭘ *Born or initialization state:*

Applet enters the initialization state when it is first loaded .This is achieved by calling init( ) method of the applet class. This applet is born. It occurs only once in the applet life cycle.

Stages:

- Create objects needed by the applet.
- Set up initial values.
- Load images or fonts.
- Setup colors.

```
G.F : public void init()
      {
              …….
              ……..   // action
      }
```

⭘ *Running state:*

Applet enters the running state when the system calls the start ( ) method of the applet class. This occurs automatically after the applet is initialized.

The Starting can also occur if the applet is already in the idle stopped state. The start( ) method may be called by more than one time.

```
G.F: public void start( )
     {
          ------
     }
```

⭘ *Idle or stopped state:*

The applet enters the idle state when it is stopped from running. Stopping occurs automatically by invoking the destroy( ) method.

Also call the stop ( ) method explicitly. If we use the thread to run the applet then we must use the stop ( ) method to terminate the thread. We can achieve by overriding the stop ( ) method.

G.F:
```
public void stop( )
{
        --------
}
```

- ***Dead state:***

An applet is said to be in dead state when it is removed from the memory. This occurs automatically by invoking the destroy ( ) method. It occurs only once in the applets life cycle. This method is used to clean up the resources.

G.F:
```
public void destroy( )
{
        ------
}
```

- ***Display state:***

Applet moves to the display state whenever it has to perform some output operation on the screen. This happens immediately after the applet enters into the running state. The paint ( ) method is called to accomplish this task. Almost every applet will have a paint ( ) method like other method in the life cycle.

G.F:
```
public void paint(Graphics s)
{
        //ACTIONS
}
```

It is inherited from the component class, a super class of applet.

THE APPLET & HTML TAG

EXPLAIN ABOUT THE HTML APPLET TAG IN BRIEF.

- The APPLET tag is used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.
- The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
    [CODEBASE = codebaseURL]
    CODE = appletFile
    [ALT = alternateText]
    [NAME = appletInstanceName]
    WIDTH = pixels HEIGHT = pixels
    [ALIGN = alignment]
    [VSPACE = pixels] [HSPACE = pixels]
    >
    [< PARAM NAME = AttributeName VALUE = AttributeValue>]
    [< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
    . . .
    [HTML Displayed in the absence of Java]
</APPLET>
```

- **CODEBASE →** It is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file.
- **CODE →** It is a required attribute that gives the name of the file containing the applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
  ALT → It is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets.
- **NAME →** It is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with

them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

- **WIDTH AND HEIGHT** → It is a required attributes that give the size (in pixels) of the applet display area.
- **ALIGN** → It is an optional attribute that specifies the alignment of the applet. This attribute has the possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.
- **VSPACE AND HSPACE** → These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet.
- **PARAM NAME AND VALUE** → It allows to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

<u>Passing Parameters to Applets</u>

The APPLET tag in HTML allows to pass parameters to the applet. To retrieve a parameter, use the getParameter( ) method. It returns the value of the specified parameter in the form of a String object. Thus, for numeric and Boolean values, we will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

<u>Use Parameters</u>

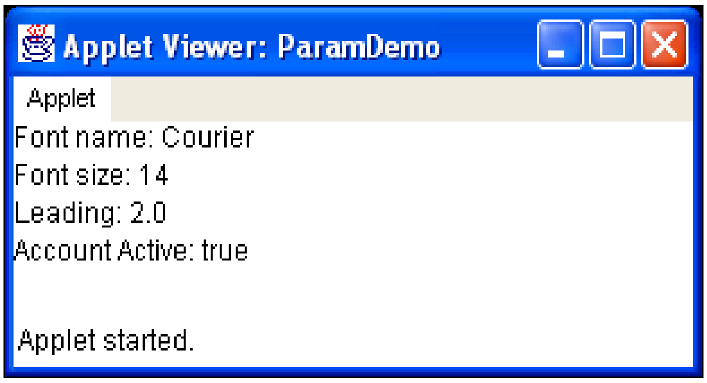
1

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300
height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet
{
 String fontName;
 int fontSize;
 float leading;
 boolean active;
 // Initialize the string to be displayed.
public void start() {
   String param;
   fontName = getParameter("fontName");
   if(fontName == null)
        fontName = "Not Found";
   param = getParameter("fontSize");
try{
     if(param != null) // if not found
        fontSize = Integer.parseInt(param);
     else
        fontSize = 0;
}
catch(NumberFormatException e)
{
   fontSize = -1;
}
param = getParameter("leading");
```

2

```
try
{
   if(param != null) // if not found
     leading = Float.valueOf(param).floatValue();
   else
      leading = 0;
}
catch(NumberFormatException e)
{
     leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
   active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g)
{
  g.drawString("Font name: " + fontName, 0, 10);
  g.drawString("Font size: " + fontSize, 0, 26);
  g.drawString("Leading: " + leading, 0, 42);
  g.drawString("Account Active: " + active, 0, 58);
}
}
```

<u>OUTPUT</u>



```
Font name: Courier
Font size: 14
Leading: 2.0
Account Active: true


Applet started.
```

Key points:

- Applets do not need a main( ) method.
- It cannot be run independently. It must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT.
- It cannot communicate with other servers on the network.
- Applets are restricted from using libraries from other languages such as C or C++.

GRAPHICS

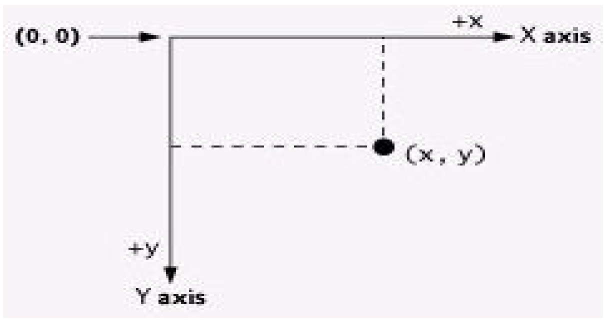
Introduction

The most important features of java is its ability to draw graphics. Using java applets we can draw lines, figures of different shapes, images and text in different fonts and styles.

To begin drawing in Java, we must first understand Java's coordinate system, which is a scheme for identifying every point on the screen. By default, the upper-left corner of a GUI component (e.g., a window) has the coordinates (0, 0). A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate).

The x-coordinate is the horizontal distance moving right from the left of the screen. The y-coordinate is the vertical distance moving down from the top of the screen. The x-axis describes every horizontal coordinate, and the y-axis describes every vertical coordinate.

Java coordinate system. Units are measured in pixels.



Text and shapes are displayed on the screen by specifying coordinates. The coordinates are used to indicate where graphics should be displayed on a screen. Coordinate units are measured in pixels. A pixel is a display monitor's smallest unit of resolution.

GRAPHICS CLASS AND METHODS

EXPLAIN GRAPHICS CLASS AND METHODS IN BRIEF.

Class Graphics is an abstract class i.e., Graphics objects cannot be instantiated. A Graphics object manages a graphics context and draws pixels on the screen that represents text and other graphical object (e.g., lines, ellipses, rectangles and other polygons).

Graphics objects contain methods for drawing, font manipulation, color manipulation and so on.

Drawing PolyLines: A polyline is a series of connected line segments, which are drawn by

- drawPolyline(int[] xPoints, int[] yPoints, int numPoints)

Drawing Lines: Lines are drawn by invoking

- drawLine(int x, int y, int x2, int y2).

Drawing Rectangles: In contrast to lines, the Graphics class provides a wealth of support for rectangles; three types of rectangles are supported:

- solid
- rounded
- 3D

The Graphics methods for painting and filling rectangles are listed below:

- void clearRect(int x, int y, int w, int h)
- void drawRect(int x, int y, int w, int h)
- void drawRoundRect(int x, int y, int w, int h, int arcWidth, in arcHeight)

- void draw3DRect(int x, int y, int w, int h, boolean raise)
- void fillRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight)
- void fillRect(int x, int y, int w, int h)
- void fill3DRect(int x, int y, int w, int h, boolean raise)

Drawing Arcs: java.awt.Graphics provides the following methods for drawing and filling arcs.

- void drawArc(int x, int y, int w, int h, int startAngle, int endAngle)
- void fillArc(int x, int y, int w, int h, int startAngle, int endAngle)

Drawing Ovals: The Graphics class provides methods for drawing and filling elliptical shapes.

- void drawOval(int x, int y, int w, int h)
- void fillOval(int x, int y, int w, int h)

Drawing Polygons: Polygons may be drawn and filled by the following Graphics methods.

- void drawPolygon(int[] xPoints, int[] yPoints, int numPoints)
- void drawPolygon(Polygon polygon)

***Drawing Text*:** The Graphics class provides three methods for rendering text.

- void drawString(String s, int x, int y)
- void drawChars(char[], int offset, int length, int x, int y)
- void drawBytes(byte[], int offset, int length, int x, int y)

In addition to performing graphical operations within a component, each `Graphics` also keeps track of the following graphical properties:

- the color used for drawing and filling shapes
- the font used for rendering text
- a clipping rectangle
- a graphical mode (Paint)
- a translation origin for rendering and clipping coordinates

Graphics Parameters:

The Graphics class fulfills two major responsibilities:

- set and get graphical parameters
- perform graphical operations in an output device

Graphics References:

There are two ways to obtain a reference to a component's Graphics:

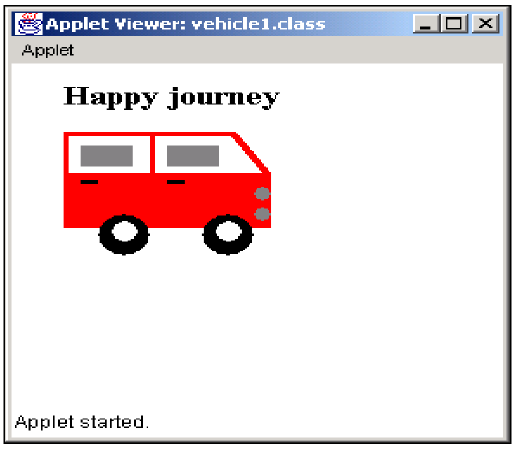
1. Override one of the methods that are passed as a Graphics Reference.
2. Invoke one of the methods that return a Graphics Reference.

Example for graphics method

```
import java.awt.*;
import java.applet.*;
public class vehicle extends Applet
{
 public void paint(Graphics g)
 {
       Font myFont=new
       Font("serif",Font.BOLD,20);
       g.setFont(myFont);
       g.drawString("Happy journey",30,30);
       g.setColor(Color.red);
       g.fillRect(30,50,50,30);
       g.setColor(Color.white);
       g.fillRect(33,53,47,27);
       g.setColor(Color.gray);
       g.fillRect(40,60,30,15);
       int x[] = {80,130,150,80,80};
       int y[] = {50,50,80,80,50};
       g.setColor(Color.red);
       g.fillPolygon(x,y,5);
       int x1[] = {83,127,147,83,83};
       int y1[] = {53,53,83,83,53};
       g.setColor(Color.white);
       g.fillPolygon(x1,y1,5);
       g.setColor(Color.red);
       g.fillRect(30,80,120,40);
       g.setColor(Color.black);
       g.fillOval(50,110,30,30);
       g.setColor(Color.white);
       g.fillOval(58,115,15,15);
       g.setColor(Color.black);
       g.fillOval(110,110,30,30);
       g.setColor(Color.white);
       g.fillOval(118,115,15,15);
       g.setColor(Color.gray);
       g.fillOval(140,90,10,10);
```

```
            g.fillOval(140,105,10,10);
            g.setColor(Color.gray);
            g.fillRect(90,60,30,15);
            g.setColor(Color.black);
            g.fillRect(40,85,10,3);
            g.fillRect(90,85,10,3);
    }
  }
/*<applet code="vehicle.class"  width=200
height=200>
 </applet>*/
```

3



EVENT HANDLING TYPES AND EXAMPLES

EXPLAIN EVENT HANDLING IN BRIEF.

- ➢ An event is an object that describes a state change in a source. An event handler requires a single piece of information: a reference to the instance of the Event class containing information about the event that just occurred.
- ➢ Events represent all activity that goes on between the user and the application. Java's Abstract Windowing Toolkit (AWT) communicates these actions to the programs using events. All user actions belong to an abstract set of things called *events.*
- ➢ There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the java.awt.event package.
- ➢ Each type of event has its own registration method.

Syn: public void addTypeListener (TypeListener el)

Event classes:

- ❖ The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. Its one constructor is shown here:
  - EventObject(Object *src*) // Here, *src* is the object that generates this event.
- ❖ EventObject contains two methods: getSource( ) → method returns the source of the event, toString( ) returns the string equivalent of the event.
- ❖ The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass of all AWT-based events used by the delegation event model. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here:
  - int getID( )

Table Event types and corresponding EventSource & EventListener

Event Type	Event Source	Event Listener interface	Meaning
ActionEvent	Button, List, TextField MenuItem,	ActionListener KeyListener	User clicked on button
AdjustmentEvent	Scrollbar	AdjustmentListener	User moved the scrollbar
ItemEvent	Choice, Checkbox, CheckboxMenuItem, List	ItemListener	User double-clicked on list item User selected or deselected item

TextEvent	TextArea, TextField	TextListener	User changed text User finished editing text
ComponentEvent FocusEvent KeyEvent MouseEvent	Component	ComponentListener FocusListener KeyListener MouseListener, MouseMotionListener	Component moved, resized, hidden, shown Component gained or lost focus User pressed or released a key many events with mouse
ContainerEvent	Container	ContainerListener	Component added to or removed from container
WindowEvent	Window	WindowListener	Window opened, closed, iconified, deiconofied,or close requested
MenuItem	ActionEvent	KeyListener	User selected a menu item

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements.

- First, it must have been registered with one or more sources to receive notifications about specific types of events.
- Second, it must implement methods to receive and process these notifications.

Table Event Listener Interfaces and corresponding methods which it defines

Event Listener interface	**Event Listener Methods**
ActionListener	actionPerformed(ActionEvent evt)
AdjustmentListener	adjustmentValueChanged(AjustmentEvent evt)
ItemListener	itemStateChanged(ItemEvent evt)
TextListener	textValueChanged(TextEvent evt)
ComponentListener	componentHidden(ComponentEvent evt), componentMoved(ComponentEvent evt), componentResized(ComponentEvent evt), componentShown(ComponentEvent evt)
ContainerListener	componentAdded(ContainerEvent evt), componentRemoved(ContainerEvent evt)
FocusListener	focusGained(FocusEvent evt), focusLost(FocusEvent evt)
KeyListener	keyPressed(KeyEvent evt), keyReleased(KeyEvent evt), keyTyped(KeyEvent evt)
MouseListener	mouseClicked(MouseEvent evt), mouseEntered(MouseEvent evt), mouseExited(MouseEvent evt), mousePressed(MouseEvent evt), mouseReleased(MouseEvent evt)
MouseMotionListener	mouseDragged(MouseEvent evt), mouseMoved(MouseEvent evt)
WindowListener	windowActivated(WindowEvent evt), windowClosed(WindowEvent evt), windowClosing(WindowEvent evt), windowDeactivated(WindowEvent evt), windowDeiconified(WindowEvent evt), windowIconified(WindowEvent evt), windowOpened(WindowEvent evt)

Demonstrate the mouse event handlers

1

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="MouseEvents"
       width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
  implements MouseListener,
                  MouseMotionListener
{
String msg = "";
int mouseX = 0, mouseY = 0; e
public void init()
{
addMouseListener(this);
addMouseMotionListener(this);
}
```

2

```
// Handle mouse clicked.
public void mouseClicked(MouseEvent me)
{
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me)
{
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
```

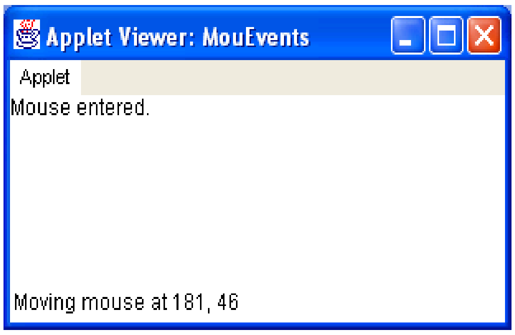
3

```
public void mouseEntered(MouseEvent me)
 {
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me)
{
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me)
{
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
mouseX = me.getX();
mouseY = me.getY();
```

4

```
msg = "*";
showStatus("Dragging mouse at "
        + mouseX + ", " + mouseY);
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me)
{
// show status
showStatus("Moving mouse at "
       + me.getX() + ", " + me.getY());
}
// Display msg in applet window
public void paint(Graphics g)
{
 g.drawString(msg, mouseX, mouseY);
}
}
```

OUTPUT



```
Mouse entered.

Moving mouse at 181, 46
```

There are five main tasks in creating our own event type:

- Create an event listener
- Create a listener adapter
- Create an event class
- Modify the component
- Managing multiple listeners

UNIT-V

AWT

- **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.

DISCUSS ABOUT AWT COMPONENTS IN BRIEF.

AWT classes fall in four categories

- components
- containers
- layout managers
- event handling

AWT COMPONENTS:

1. Labels: This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in your application and label never perform any type of action. Syntax for defining the label only and with justification.

   Label label_name = new Label ("This is the label text.");

   Above code simply represents the text for the label.

   Label label_name = new Label ("This is the label text.", Label.CENTER);

   Justification of label can be left, right or centered. Above declaration used the center justification of the label using the Label.CENTER.

2. Buttons: This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for your application. The syntax of defining the button is as follows :

   Button button_name = new Button ("This is the label of the button.");

   We can change the Button's label or get the label's text by using the Button.setLabel(String) and Button.getLabel() method. Buttons are added to the container using the add(button_name) method.

3. Check Boxes: This component of Java AWT allows you to create check boxes in your applications. The syntax of the definition of Checkbox is as follows :

   CheckBox checkbox_name = new Checkbox ("Optional check box 1", false);

   Above code constructs the unchecked Checkbox by passing the boolean valued argument *false* with the Checkbox label through the Checkbox() constructor. Defined Checkbox is added to it's container using add (checkbox_name) method. We can change and get the checkbox's label using the setLabel (String) and getLabel() method. We can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the Checkbox class.

4. Radio Button: This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows :

   ```
   CheckboxGroup chkgp = new CheckboxGroup();
   add (new Checkbox ("One", chkgp, false);
   add (new Checkbox ("Two", chkgp, false);
   add (new Checkbox ("Three",chkgp, false);
   ```

   In the above code we are making three check boxes with the label "One", "Two" and "Three". If we mention more than one true valued for checkboxes then our program takes the last true and shows the last check box as checked.

5. Text Area: This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

   TextArea txtArea_name = new TextArea();

   We can make the Text Area editable or not using the setEditable (boolean) method. If we pass the boolean valued argument *false* then the text area will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in the text area using the setText(string) method of the TextArea class.

6. Text Field: This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows :

   TextField txtfield = new TextField(20);

   We can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

AWT CONTROLS

EXPLAIN AWT CONTROLS IN BRIEF.

Controls are components that allow a user to interact with our application in various ways. For example- a commonly used control is the push button.

An awt controls is a component that allows the end users to interact with applications.

Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of **Component**.

Adding Controls:

To include a control in a window, first create an instance of the desired control and then add it to a window by calling **add(**), which is defined by **Container**. The **add( )** method has several forms.

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that we want to add. A reference to *compObj* is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Removing Controls:

To remove a control from a window when the control is no longer needed call **remove(**). This method is also defined by **Container**. It has this general form:

void remove(Component *obj*)

Here, *obj* is a reference to the control we want to remove. We can remove all controls by calling **removeAll(**).

Responding to Controls

Except labels, all controls generate events when they are accessed by the user.

1. Labels

A *label* is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:

❖ Label( ) → creates a blank label

- Label(String *str*) → creates a label that contains the string specified by *str* (i.e.: left-justified)
- Label(String *str*, int *how*) → creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.
- void setText(String *str*) → set or change the text in a label
- String getText( ) → obtain the current label
- void setAlignment(int how) → set the alignment of the string within the label.
- int getAlignment( ) → obtain the current alignment.

2. Using Buttons

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

- Button( ) → creates an empty button
- Button(String *str*) → creates a button that contains *str* as a label.
- void setLabel (Strint str) → set its label of button.
- String getLabel() → retrieve its label .

Handling Buttons

- Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.
- Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs.
- An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the string that is the label of the button.

3. Check Boxes:

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. We change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class. Checkbox supports these constructors:

- Checkbox( ) → creates a check box whose label is initially blank.
- Checkbox(String *str*) → creates a check box whose label is specified by *str*.
- Checkbox(String *str*, boolean *on*) → allows to set the initial state of the check box.
- Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) → Create a check box whose label is specified by *str* and whose group is specified by *cbGroup*
- Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) → The value of *on* determines the initial state of the check box.
- boolean getState( ) → To retrieve the current state of a check box.
- void setState(boolean *on*) → To set the state.
- String getLabel( ) → Obtain the current label associated with a check box.
- void setLabel(String *str*) → To set the label.
- Checkbox getSelectedCheckbox( ) → determine which check box in a group is currently selected.
- void setSelectedCheckbox(Checkbox *which*) → set a check box.

Handling Check Boxes

Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method.

4. Choice Controls

- The Choice class is used to create a *pop-up list* of items from which the user may choose. Thus, a Choice control is a form of menu.
- When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made.

- Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.
- Choice only defines the default constructor, which creates an empty list.
- To add a selection to the list, call add( ). It has this general form:

  void add(String *name*) → *name* is the name of the item being added.
- Items are added to the list in the order in which calls to add( ) occur. To determine which item is currently selected, the following methods are called:
  - String getSelectedItem( ) → returns a string containing the name of the item.
  - int getSelectedIndex( ) → returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.
  - getItemCount( ) → obtain the number of items in the list,
  - select( ) → set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:
    - int getItemCount( )
    - void select(int *index*)
    - void select(String *name*)

<u>Handling Choice Lists</u>

Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.

<u>5. Using Lists</u>

The List class provides a compact, multiple-choice, scrolling selection list. A List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors:

- ❖ List( ) → creates a **List** control that allows only one item to be selected at any one time.
- ❖ List(int *numRows*) → to specifiy the number of entries in the list that will always be visible.
- ❖ List(int *numRows*, boolean *multipleSelect*) → the user may select two or more items at a time.

To add a selection to the list, call **add( )**. It has the following two forms:

- void add(String *name*) → the name of the item added to the list.
- void add(String *name*, int *index*) → adds the item at the index specified by *index.* Indexing begins at zero. You can specify –1 to add the item to the end of the list.

- ❖ To determine which item is currently selected by calling either String getSelectedItem( ) or int getSelectedIndex( ).
- ❖ The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.
- ❖ For lists that allow multiple selection, use either String[ ] getSelectedItems( ) or int[ ] getSelectedIndexes( ) to determine the current selections.
  - getSelectedItems( ) returns an array containing the names of the currently selected items.
  - getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.
- ❖ To obtain the number of items in the list, call int getItemCount( ). We can set the currently selected item by using the void select(int index) method with a zero-based integer index.
- ❖ To obtain the name associated with the item at that index by calling String getItem(int index). Here, *index* specifies the index of the desired item.

Handling Lists

- To process list events implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated.
- Its getActionCommand( ) method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated.
- Its getStateChange( ) method can be used to determine whether a selection or deselection triggered this event. getItemSelectable( ) returns a reference to the object that triggered this event

6. Managing Scroll Bars

- Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts. Each end has an arrow to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar.
- The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the Scrollbar class. Scrollbar defines the following constructors:
    - Scrollbar( ) → creates a vertical scroll bar.
    - Scrollbar(int style) → allows to specify the orientation of the scroll bar.
    - Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
        - → If style is Scrollbar.VERTICAL, a vertical scroll bar is created.
        - → If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal.
        - → The initial value of the scroll bar is passed in initialValue.
        - → The number of units represented by the height of the thumb is passed in thumbSize
    - The minimum and maximum values for the scroll bar are specified by min and max.
    - void setValues(int initialValue, int thumbSize, int min, int max)
    - To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows: int getValue( ) & void setValue(int newValue) → newValue specifies the new value for the scroll bar.
    - To retrieve the minimum and maximum values call int getMinimum( ) and int getMaximum( ).

Handling Scroll Bars

To process scroll bar events implement the AdjustmentListener interface. Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated. Its getAdjustmentType( ) method can be used to determine the type of the adjustment.
The types of adjustment events are as follows:

- BLOCK_DECREMENT-- A page-down event has been generated.
- BLOCK_INCREMENT-- A page-up event has been generated.
- TRACK --An absolute tracking event has been generated.
- UNIT_DECREMENT --The line-down button in a scroll bar has been pressed.
- UNIT_INCREMENT -- The line-up button in a scroll bar has been pressed.

7. Using a TextField:

- The TextField class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- TextField is a subclass of TextComponent. TextField defines the following constructors:
    - TextField( ) → Creates a default text field.
    - TextField(int *numChars*) → creates a text field that is *numChars* characters wide.
    - TextField(String *str*) → initializes the text field with the string contained in *str*.

- TextField(String *str*, int *numChars*) → initializes a text field and sets its width.

- TextField provides several methods that allow to utilize a text field. To obtain the string currently contained in the text field, the following methods are used:
  - String getText( )
  - void setText(String str) → Here, str is the new string.

- The user can select a portion of the text in a text field. Also, select a portion of text under program control by using select( ) and obtain the currently selected text by calling getSelectedText( ). These methods are shown here:
  - String getSelectedText( ) → returns the selected text.
  - void select(int startIndex, int endIndex) → selects the characters beginning at startIndex and ending at endIndex–1.

- We can control whether the contents of a text field may be modified by the user by calling setEditable( ). We can determine editability by calling isEditable( ). These methods are shown here:
  - boolean isEditable( ) → returns true if the text may be changed and false if not.
  - void setEditable(boolean canEdit) → if canEdit is true, the text may be changed. If it is false, the text cannot be altered.

- We can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered.
- We can check a text field to see if it is in this mode with the echoCharIsSet( ) method. We can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:
  - void setEchoChar(char ch)
  - boolean echoCharIsSet( )
  - char getEchoChar( ) → Here, ch specifies the character to be echoed.

8. Using a TextArea:

- Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:
  - TextArea( )
  - TextArea(int numLines, int numChars) → numLines specifies the height, in lines.
  - TextArea(String str) → Initial text.
  - TextArea(String str, int numLines, int numChars) → numChars specifies its width, in characters.
  - TextArea(String str, int numLines, int numChars, int sBars) → specify the scroll bars. Where *sBars* must be one of these values:
    - SCROLLBARS_BOTH SCROLLBARS_NONE
    - SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY

- TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods.
- Text areas only generate got-focus and lost-focus events.
- **TextArea** adds the following methods:
  - void append(String str) → appends the string specified by str to the end of the current text.
  - void insert(String str, int index) → inserts the string passed in str at the specified index.
  - void replaceRange(String str, int startIndex, int endIndex) → replaces the characters from startIndex to endIndex–1, with the replacement text passed in str.

AWT Layouts

Introduction

Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container. The task of layouting the controls is done automatically by the Layout Manager.

Layout Manager

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.

Following are the interfaces defining functionalities of Layout Managers.

Introduction

The class **FlowLayout** components in a left-to-right flow.

Class declaration

Following is the declaration for **java.awt.FlowLayout** class:

```
public class FlowLayout
  extends Object
    implements LayoutManager, Serializable
```

Field

Following are the fields for **java.awt.BorderLayout** class:

- **static int CENTER** -- This value indicates that each row of components should be centered.
- **static int LEADING** -- This value indicates that each row of components should be justified to the leading edge of the container's orientation, for example, to the left in left-to-right orientations.
- **static int LEFT** -- This value indicates that each row of components should be left-justified.
- **static int RIGHT** -- This value indicates that each row of components should be right-justified.
- **static int TRAILING** -- This value indicates that each row of components should be justified to the trailing edge of the container's orientation, for example, to the right in left-to-right orientations.

Class constructors

S.N.	Constructor & Description
1	**FlowLayout()** Constructs a new FlowLayout with a centered alignment and a default 5-unit horizontal and vertical gap.
2	**FlowLayout(int align)** Constructs a new FlowLayout with the specified alignment and a default 5-unit horizontal and vertical gap.
3	**FlowLayout(int align, int hgap, int vgap)** Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

Class methods

S.N.	Method & Description
1	**void addLayoutComponent(String name, Component comp)** Adds the specified component to the layout.
2	**int getAlignment()** Gets the alignment for this layout.
3	**int getHgap()** Gets the horizontal gap between components.
4	**int getVgap()** Gets the vertical gap between components.
5	**void layoutContainer(Container target)** Lays out the container.
6	**Dimension minimumLayoutSize(Container target)** Returns the minimum dimensions needed to layout the visible components contained in the specified target container.

7	**Dimension preferredLayoutSize(Container target)** Returns the preferred dimensions for this layout given the visible components in the specified target container.
8	**void removeLayoutComponent(Component comp)** Removes the specified component from the layout.
9	**void setAlignment(int align)** Sets the alignment for this layout.
10	**void setHgap(int hgap)** Sets the horizontal gap between components.
11	**void setVgap(int vgap)** Sets the vertical gap between components.
12	**String toString()** Returns a string representation of this FlowLayout object and its values.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

FlowLayout Example

Create the following java program using any editor of your choice in say **D:/ > AWT > com > tutorialspoint > gui >**

AwtLayoutDemo.java

```
package com.tutorialspoint.gui;

import java.awt.*;
import java.awt.event.*;

public class AwtLayoutDemo {
  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;
  private Label msglabel;

  public AwtLayoutDemo(){
    prepareGUI();
  }
  public static void main(String[] args){
    AwtLayoutDemo  awtLayoutDemo = new AwtLayoutDemo();
    awtLayoutDemo.showFlowLayoutDemo();
  }

  private void prepareGUI(){
    mainFrame = new Frame("Java AWT Examples");
    mainFrame.setSize(400,400);
    mainFrame.setLayout(new GridLayout(3, 1));
    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
      }
    });
    headerLabel = new Label();
    headerLabel.setAlignment(Label.CENTER);
    statusLabel = new Label();
    statusLabel.setAlignment(Label.CENTER);
    statusLabel.setSize(350,100);

    msglabel = new Label();
    msglabel.setAlignment(Label.CENTER);
    msglabel.setText("Welcome to TutorialsPoint AWT Tutorial.");

    controlPanel = new Panel();
    controlPanel.setLayout(new FlowLayout());

    mainFrame.add(headerLabel);
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }

  private void showFlowLayoutDemo(){
    headerLabel.setText("Layout in action: FlowLayout");

    Panel panel = new Panel();
    panel.setBackground(Color.darkGray);
    panel.setSize(200,200);
    FlowLayout layout = new FlowLayout();
    layout.setHgap(10);
    layout.setVgap(10);
    panel.setLayout(layout);
    panel.add(new Button("OK"));
    panel.add(new Button("Cancel"));

    controlPanel.add(panel);

    mainFrame.setVisible(true);
  }
}
```

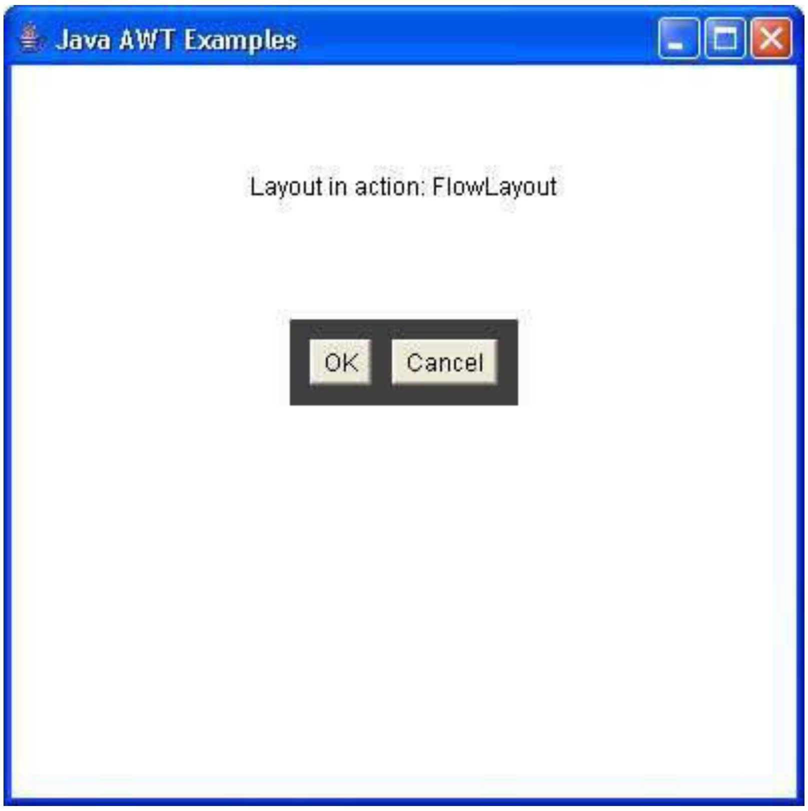
Compile the program using command prompt. Go to **D:/ > AWT** and type the following command.

```
D:\AWT>javac com\tutorialspoint\gui\AwtlayoutDemo.java
```

If no error comes that means compilation is successful. Run the program using following command.

```
D:\AWT>java com.tutorialspoint.gui.AwtlayoutDemo
```

Verify the following output



BAR CHARTS

DRAWING BAR CHARTS

Applets can be designed to display bar charts, which are commonly used in comparative analysis of data. The method getParameter() is used to fetch the data values from the HTML file. It returns only string values and it use the wrapper class method parseInt() to convert strings to integer values.

Ex:

```
import java.awt.*;
import java.applet.*;
/*<Applet code = "chart.class" height=100 width=200> </Applet>*/
public class chart extends Applet
{
int n = 0;
String label [ ];
int value[ ];
public void init( )
{
        try
        {
        n = Integer.parseInt(getParameter("columns"));
        label = new String[n];
        value = new int[n];
        label[0] = getParameter("l1");
```

```
                label[1] = getParameter("l2");
                label[2] = getParameter("l3");
                value[0] = Integer.parseInt(getParameter("c1"));
                value[1] = Integer.parseInt(getParameter("c2"));
                value[2] = Integer.parseInt(getParameter("c3"));
                }catch(NumberFormatException e) {  }
        }
    public void paint(Graphics g)
        {
                for(int i = 0; i<n; i++)
                {
                g.setColor(Color.red);
                g.drawString(label[i], 20, i*50+30);
                g.fillRect(50, i*50+10, value[i], 40);
        }
    }
}
```

Java AWT MenuItem and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

```
public class MenuItem extends MenuComponent implements Accessible
```

AWT Menu class declaration

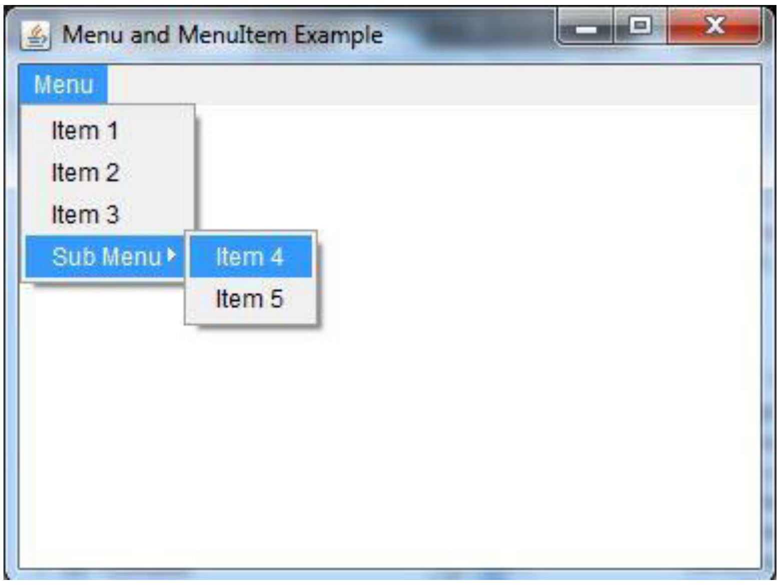
```
public class Menu extends MenuItem implements MenuContainer, Accessible
```

Java AWT MenuItem and Menu Example

```
import java.awt.*;
class MenuExample
{
     MenuExample(){
         Frame f= new Frame("Menu and MenuItem Example");
         MenuBar mb=new MenuBar();
         Menu menu=new Menu("Menu");
         Menu submenu=new Menu("Sub Menu");
         MenuItem i1=new MenuItem("Item 1");
         MenuItem i2=new MenuItem("Item 2");
         MenuItem i3=new MenuItem("Item 3");
         MenuItem i4=new MenuItem("Item 4");
         MenuItem i5=new MenuItem("Item 5");
         menu.add(i1);
         menu.add(i2);
         menu.add(i3);
```

```
                submenu.add(i4);
                submenu.add(i5);
                menu.add(submenu);
                mb.add(menu);
                f.setMenuBar(mb);
                f.setSize(400,400);
                f.setLayout(null);
                f.setVisible(true);
        }
        public static void main(String args[])
        {
        new MenuExample();
        }
        }
```

Output:


Menu and MenuItem Example
Menu
Item 1
Item 2
Item 3
Sub Menu
Item 4
Item 5