



NAGARJUNA COLLEGE OF ENGINEERING AND TECHNOLOGY
(An autonomous college under VTU)
DEPARTMENT OF CSE (DATA SCIENCE)

B.E 3rd SEM

Object Oriented Programming through

JAVA

(22CDT133)

MODULE 1

<p>INTRODUCTION OBJECT ORIENTED CONCEPTS</p>	<ul style="list-style-type: none">➤ PROCEDURE-ORIENTED PROGRAMMING➤ OBJECT ORIENTED PROGRAMMING➤ COMPARISON OF OBJECT ORIENTED LANGUAGE WITH C.➤ INTRODUCTION TO JAVA:<ul style="list-style-type: none">❖ JAVA BUZZWORDS❖ THE BYTE CODE❖ JAVA DEVELOPMENT KIT (JDK)❖ DATA TYPES, VARIABLES❖ ARRAYS❖ OPERATORS❖ CONTROL STATEMENTS❖ SIMPLE JAVA PROGRAMS.
--	--

Procedure Oriented Programming (POP)

- A program in a procedural language is a list of instruction where each statement tells the computer to do something.
- It focuses on procedure (Process / function) rather than data while doing computation.
- POP follows top-down approach during the designing of a program.
- When program become larger, it is divided into function & each function has clearly defined purpose.

Examples- **C, BASIC, ALGOL, COBAL, FORTRAN**

Dis-Advantage of POPS:

- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- POP doesn't support object programming features like – abstraction, Encapsulation, Inheritance etc.,

Object Oriented Programming (OOP)

- It focuses mainly on creation of objects rather than function.
- The main idea behind object oriented approach is to combine process (function / method) and data into a unit called an **object** (member+member function). Hence, it focuses on objects rather than procedure.
- Object Oriented programming (OOP) is a programming standard that relies on the concept of **classes** and **objects**.
- Example - C++, Java, Python , c# (c sharp) etc.,

Advantages

- Secure, protects information through encapsulation.
- OOP models complex things as reproducible, simple structures Reusable.
- We can create new data types
- Abstraction
- Less development time
- Data hiding
- Multiple objects feature

Features (Characteristics) of OOC

- Class
- Object
- Data Abstraction
- Data Encapsulation

- Inheritance
- Polymorphism

Class

- A Class is a collection of objects (Members and member functions).
- Keyword is **class**.
- Class can contain fields, methods, constructors, and certain properties.
- Class acts like a blueprint.
- When defining class, it should starts with keyword class followed by name of the class; and the class body, enclosed by a pair of curly braces.

Syntax:

```
class <class_name>
{
    // class Body
}
```

Example:

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

Object

- Object is an instance of the class.
- Object is used to access members (variables) and member functions (methods).

Example :

```
public class First
{
    public void display( )                // method
    {
        System.out.println("NCET");
    }
    public static void main(String args[ ])
    {
        First obj = new First( ) ;        //create object
        obj . display( ) ;                 //access display method using object obj
    }
}
```

Output:

NCET

Data Abstraction

- Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- In simple words: Display what is necessary. Let other things rest in peace.

For example, when you login to your face book account, you enter your user_id and password and press login, what happens when you press login; how the input data sent to server, how it gets verified is all abstracted away from you.

Example:

```
public class First
{
    public static void main(String args[ ])
    {
        System.out.println("My name is Kiran");
    }
}
```

In the above example we are printing the message “My name is Kiran” using println function but we are not bothered about how println is working internally to display that message.

Data Encapsulation

- Wrapping (combining) of data and functions into a single unit (class) is known as data encapsulation.
- Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.
- We can also call Information hiding.
- Java supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.
- Data Encapsulation can be provided by Access modifiers of the class i.e
 - ❖ public (members of class can be access from anywhere)
 - ❖ private (members of class can be access only within a class)
 - ❖ protected (members of class can be accessible through its child class)

Example: //I shown here only for private; similarly we can write for public and protected also.

```
class First
{
    private int a=10;
}
class Second extends First
{
    public static void main(String args[ ])
    {
        Second obj = new Second();
        System.out.println( obj.a );
    }
}
```

Output:

```
a has private access in First
System.out.println( obj.a );
^
1 error
```

Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- **extends** keyword should be used to inherit properties.

Example: Below example shows Second class can access values of a and b from Class One

```
class One
{
    int a=10,b=20;
}
class Second extends One
{
    public static void main(String[ ] args)
    {
        Second obj=new Second( );
        System.out.println(obj.a + obj.b);
    }
}
```

Output:

30

Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- Example: Method overloading (same method name but different action) and method overriding (same method name override in child class with different action)

```
//Method Overloading
class One
{
    public void sum(int a,int b)
    { System.out.println(a+b); }

    public void sum(int a,int b,int c)
    { System.out.println(a+b+c); }

    public static void main(String s[ ] )
    {
        One obj=new One( );
        obj.sum(10,20);
        obj.sum(10,20,30);
    }
}
```

Output:

30
60

```
//Method Overriding
We will discuss in 3rd Module in details
```

Comparison of Object oriented language and Procedure Oriented language

Sl.	Object oriented	Procedure oriented
1	Program is divided into objects	Program is divided into functions
2	Bottom-up approach	Top-down approach
3	Inheritance property is supported.	Inheritance is not allowed
4	Supports Abstraction	Doesn't Supports Abstraction
5	Polymorphism property is supported.	Polymorphism property is not supported.
6	It uses access specifier.	It doesn't use access specifier.
7	Encapsulation is used to hide the data	No data hiding.
8	Concept of virtual function	No virtual function.
9	It is complex to implement	It is simple to implement
10	C++, JAVA	C, PASCAL
11	file extension is .java	file extension is .c
12	Allows you to use function overloading.	C does not allow you to use function overloading.
13	The data is secured.	The data is not secured.
14	Supports Multi-Threading.	Doesn't Supports Multi-Threading.

Introduction to JAVA

JAVA was developed by **James Gosling** at **Sun Microsystems** Inc in the year **1995**, later acquired by Oracle Corporation.

It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs.

Java is a high level, robust, object-oriented and secure programming language.

History:

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.

- ❖ James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- ❖ Firstly, it was called "**Greentalk**" by James Gosling, and the file extension was .gt.
- ❖ After that, it was called **Oak** and was developed as a part of the Green project.
- ❖ In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies.
- ❖ Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
- ❖ Initially developed by James Gosling at Sun Microsystems and released in 1995.

- ❖ JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Versions

Version	Date	Version	Date
JDK Beta	1995	Java SE 11	2018
JDK 1.0	1996	Java SE 12	2019
JDK 1.1	1997	Java SE 13	2019
J2SE 1.2	1998	Java SE 14	2020
J2SE 1.3	2000	Java SE 15	2020
J2SE 1.4	2002	Java SE 16	2021
Java SE 5	2004	Java SE 17	2021
Java SE 6	2006	Java SE 18	2022
Java SE 7	2011	Java SE 19 (Latest)	2022
Java SE 8	2014		
Java SE 9	2017		
Java SE 10	2018		

JAVA Buzzwords

- ❖ The features of Java are also known as Java buzzwords.
- ❖ Java Buzzwords are –
 - Simple
 - Secure
 - Portable
 - Object-oriented
 - Platform Independent
 - Robust
 - Multithreaded
 - Architecture-neutral
 - Interpreted
 - High performance
 - Distributed
 - Dynamic

Simple

- ❖ Java was designed to be easy for the professional programmer to learn and use effectively.
- ❖ If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Because Java inherits the C/C++ syntax.

Secure

- ❖ Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
 - No explicit pointer
 - Java Programs run inside a virtual machine sandbox
 - **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically.
 - **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Portable

- ❖ Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Object -Oriented

- ❖ Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.
- ❖ Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Platform Independent

- ❖ Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.
- ❖ A platform is the hardware or software environment in which a program runs.
- ❖ Java provides a software-based platform.
- ❖ Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into **bytecode**. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere.

Robust

The English meaning of Robust is strong. Java is robust because:

- ❖ It uses strong memory management.
- ❖ There is a lack of pointers that avoids security problems.
- ❖ Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- ❖ There are exception handling and the type checking mechanism in Java.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architectural Neutral / Machine Independent

- ❖ Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
- ❖ In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Interpreted and High Performance

- ❖ Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine.
- ❖ Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

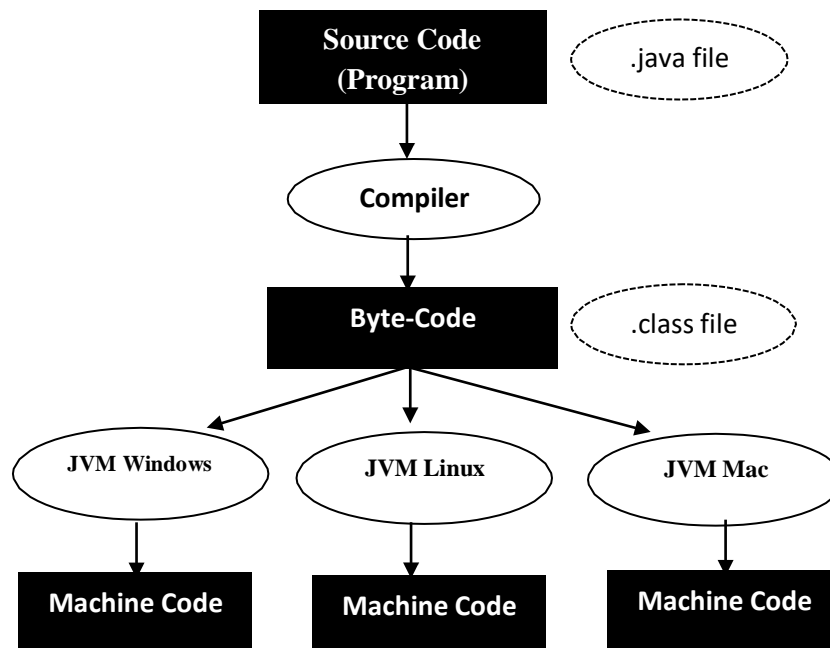
- ❖ Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- ❖ In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

- ❖ Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.
- ❖ Java supports dynamic compilation and automatic memory management (garbage collection).

JAVA Byte code

- ❖ Byte-code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- ❖ Java compiler is not executable code.
- ❖ It required Byte-code to execute java program.
- ❖ The original JVM was designed as an interpreter for byte-code.
- ❖ A Java program is executed by the JVM helps solve the major problems associated with web-based programs.
- ❖ Translating a Java program into byte-code to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- ❖ The JVM will differ from platform to platform; all understand the same Java byte-code.
- ❖ The execution of byte-code by the JVM is the easiest way to create truly portable programs.
- ❖ A Java program is executed by the JVM also helps to make it secure.
- ❖ It can contain the program and prevent it from generating side effects outside of the system.



Working:

1. First we will write the java code with .java extension using any editor Eclipse IDE(Integrated Development Environment)
2. Then, compiler will convert the source file to Byte code with .class extension.
3. The Byte code then converts to Machine code by using Java Virtual Machine (JVM).
4. We can run this bytecode on any other platform as well. But the bytecode is a non-runnable code that requires or relies on an interpreter. This is where JVM plays an important part.

Note: Difference between Machine code and Byte code

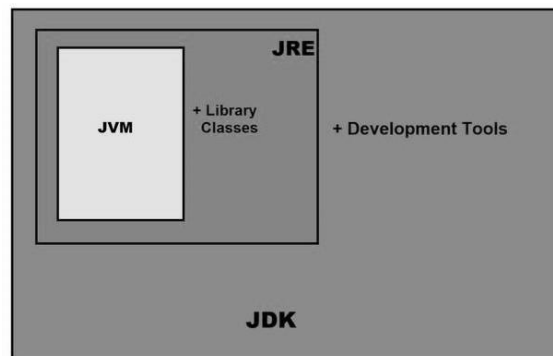
The main difference between the machine code and the bytecode is that the machine code is a set of instructions in machine language or binary which can be directly executed by the CPU.

While the bytecode is a non-runnable code generated by compiling a source code that relies on an interpreter to get executed.

Java Development Kit (JDK)

- ❖ The Java Development Kit (JDK) is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications and applets.
- ❖ It is a core package used in Java, along with the **JVM (Java Virtual Machine)** and the **JRE (Java Runtime Environment)**.
- ❖ Beginners often get confused with JRE and JDK, if you are only interested in running Java programs on your machine then you can easily do it using Java Runtime Environment. However, if you would like to develop a Java-based software application then along with JRE you may need some additional necessary tools, which is called JDK.

➤ **JDK=JRE+Development Tools**



Steps to Run java program using Notepad and JDK –

1. Install JDK , Set the JDK path in the system environmental variable location.
2. Write program using Notepad editor and save file name as class name with .java extension (First.java) (Pretend we saving our java file in desktop)
3. Open command prompt , change the directory where the java file present.
C:\> cd desktop
4. Compile the java file using command –
C:\desktop> javac First .java
5. If any errors present compiler will show and programmer should clear those bugs.
6. Run the java file to see output using command
C:\desktop> java First

First Java Program

```
//java program to display name
//save as First.java

import java.io.* ;
class First
{
    public static void main(String args[ ] )
    {
        System.out.println("Welcome to NCET");
    }
}
```

Working

- ❖ Save file name as **First.java** (should be same as class name)
- ❖ In the above program , first line shows the comment
 - Single line comment starts with the symbol `//`
 - Multi line comment starts with `/*` and ends with `*/`
- ❖ Next we used import statement to import an entire package or sometimes import certain classes and interfaces inside the package. The import statement is written before the class definition and after the package statement (if there is any). Also, the import statement is optional.

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

```
import java.util.Scanner ;
```

- ❖ Class definition starts with keyword class followed by name

```
class First {
```

- ❖ The next line of code is shown here:

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

- This line begins the **main()** method at which the program will begin executing.
- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members.
- The keyword **static** allows **main()** to be called without having to create object (instance of the class).
- The keyword **void** simply tells the compiler that **main()** does not return a value.
- In **main()**, there is only one parameter, **String args[]** declares a parameter named **args**, which is an array of instances of the class **String**. (Arrays are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.
- The last character on the line is the **{**. This signals the start of **main()**'s body.

Note: Our program may contain more than one class also but only one class contains main method to get things started.

- ❖ The next line of code is shown here. Notice that it occurs inside main().

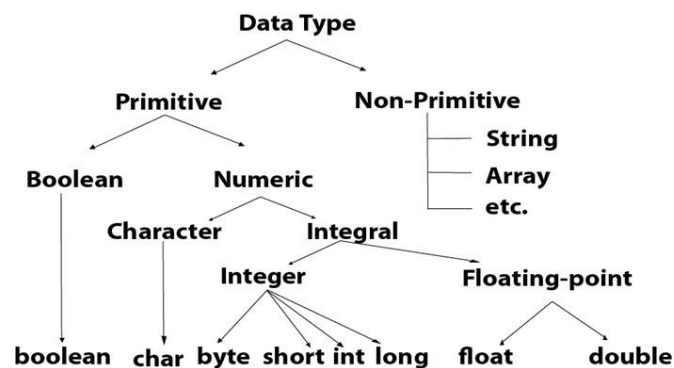
System.out.println("Welcome to NCET ");

- ❖ This line outputs the string “Welcome to NCET” followed by a new line on the screen. Output is actually accomplished by the built-in println() method. In this case, println() displays the string which is passed to it. As you will see, println() can be used to display other types of information, too. The line begins with
 - **System.out.** System is a predefined class that provides access to the system, and out is the output stream that is connected to the console.
- ❖ The first } in the program ends main(), and the last } ends the First class definition.

Data types, Variables and Arrays

Data Types:

- ❖ Specifies type of data we can store in computer memory.
- ❖ Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:
 - Primitive data types – boolean, char, byte, short, int, long, float and double
 - Non-Primitive data types (Derived) – String, Array, classes, interfaces etc.,



❖ Boolean –

- The Boolean data type is used to store only two possible values: true and false.
- The Boolean data type specifies **one bit** of information, but its "size" can't be defined precisely.

Example:

```
Boolean one = false ;
System.out.println(one) ;           //false
```

❖ Byte –

- It is an (1 byte)8-bit signed two's complement integer. Its value-range lies between -128 to 127 (-2^8 to 2^8-1).

- Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required.
- It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: **byte a = 10, b = -20 ;**

❖ Short

- The short data type is a (2 bytes) 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (-2^{16} to $2^{16}-1$). Its default value is 0.
- The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: **short s=10000 , r=-5000;**

❖ Int

- The int data type is a (4 bytes) 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^{32}) to 2,147,483,647 ($2^{32}-1$) . Its default value is 0.
- The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: **int a=10000, b=-5000;**

❖ Long

- The long data type is a 64-bit (8 bytes) two's complement integer. Its value-range lies between -2^{64} to $2^{64}-1$.
- Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: **long a=100000L**

❖ Float

- Variables of type float are useful when we need to store fractional component.
- float is a keyword.
- Size is 4 bytes (32 bits) and its value range lies between -2^{32} to $2^{32}-1$
- Example: **float a=10.6;**

❖ Double

- The double data type is a 64-bit floating point. Its value range is unlimited.
- The double data type is generally used for decimal values just like float.
- Example: **double a=10.6;**

❖ char

- The char data type is used to store characters.
- The char data type is a single 16-bit Unicode (ASCII) character.
- The range of a char is 0 to 65,536. There are no negative chars.

- ASCII(American Standard Code for Information Interchange), a set of digital codes representing letters, numerals, and other symbols, widely used as a standard format in the transfer of text between computers.
- Example:

```
char a='K';           //K
char a='75';          //75
char a=75;            //K
```

Note: in the 3rd example above, 75 is not inside pair of single quotes so we will get answer as ASCII code of that

Arrays

- ❖ Array is group of similar type of elements referenced by 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 in Java work differently than they do in other languages like C,C++.

One-Dimensional Array -

- ❖ Elements stored either in single row or single column we called 1-dimensional array.

Declaration:

Syntax: **type name [] ;**

Example: **int a [] ;**

Instantiation of an Array in Java:

Syntax: **name = new type [size] ;**

Example: **a=new int[5] ;**

We can also combine both declaration and instantiation as –

Syntax : **type name []= new type [size] ;**

Example: **int a [] = new int [5] ;**

Note: 1. we can also declare like this – **int [] a = new int[5];**

2. we can initialize values of array directly using , **int a[] = {10,20,30,40};** //no need of new

Example:

```
class First
{
public static void main( String args[ ] )
{
    int a[ ] = new int[ 5 ] ;    //declaration and instantiation
    a[0]=10;                    //initialization
    a[1]=20;
    //printing array
    for(int i=0 ; i<a.length ; i++)    //length is the property of array
        System.out.println( a[i] );
    }
}
```

Output:

```
10
20
0
0
0
```


Multi-Dimensional Array -

- ❖ Elements stored in the form of more than one row and more than one column we called multi-dimensional array.

Declaration:

Syntax: **type name [] [] ;** Example: **int a [] [] ;**

Instantiation of an Array in Java:

Syntax: **name = new type [row_size][col_size] ;** Example: **a=new int[5][4] ;**

We can also combine both declaration and instantiation as –

Syntax : **type name [][]= new type [row] [col] ;**

Example: **int a [][] = new int [5][5] ;**

Strings

- ❖ String is group of characters with in pair of double quotations (“ and “).
- ❖ String is not a simple type. Nor is it simply an array of characters. Rather, String defines an object. (Later we will discuss).
- ❖ The String type is used to declare string variables. We can also declare arrays of strings.
- ❖ A quoted string constant can be assigned to a String variable.
- ❖ A variable of type String can be assigned to another variable of type String. We can use an object of type String as an argument to println().

For example,

```
String s = “Kiran” ;
```

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

Here, s is an object of type String. It is assigned the string “Kiran”. This string is displayed by the println() statement.

Variable

- ❖ Variable is a name given to memory location to store data while java program executed.
- ❖ It is a combination of "vary + able" which means its value can be changed.

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

Local Variable

- ❖ A variable declared inside the body of the method is called local variable.
- ❖ You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
- ❖ A local variable cannot be defined with "static" keyword.

Instance Variable

- ❖ A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.
- ❖ To access instance variable we need to create object of the class.

Static variable

- ❖ A variable that is declared as static is called a static variable. It cannot be local.
- ❖ To access static variable no need to create object of the class.
- ❖ You can create a single copy of the static variable and share it among all the instances of the class.
- ❖ Memory allocation for static variables happens only once when the class is loaded in the memory.

Example:

```
public class A
{
    int data=50;           //instance variable
    static int m=100;       //static variable
    void demo( )
    {
        int n=90;          //local variable
    }
    public static void main(String args[])
    {
        int b=20;           //instance variable
    }
}
```

In the above example ,

- ❖ variable 'data' is a instance variable because to access 'data' variable we should create object of the class.
- ❖ variable 'm' is declared as static that means, we can access 'm' anywhere without creating object.
- ❖ variable 'n' is a local variable that means variable 'n' is visible(valid) only inside method demo() and we can't access in other methods.

Read and Display value of a variable**Display (output) :**

To **Display** value or string or combination of value and string, we will use the statement,

```
System.out.println("Kiran");           //prints just a string
System.out.println(a);                  //prints just value of a string
System.out.println("Result is"+a);      //prints combination of value and string
```

Where,

- ❖ **println()** is used to display along with newline . We can also use just **print()** but without newline.
- ❖ **out** is an instance of **PrintStream** type, which is a public and static member field of the **System** class.

Read (Input):

- ❖ To **Read** value of a variable at compile time we will use ,

Example: **int a=10,b=20;**

- ❖ To Read value of a variable at Run time (from the keyboard) we will use **Scanner class**

Scanner:

- Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the **java.util.Scanner** class is one of them.
- The Java Scanner class provides **nextXXX()** methods to return the type of value such as **nextInt()**, **nextByte()**, **nextShort()**, **next()**, **nextLine()**, **nextDouble()**, **nextFloat()**, **nextBoolean()**, etc.
- To get a single character from the scanner, you can call **next().charAt(0)** method which returns a single character.

Example:

```
Scanner sc = new Scanner(System.in);
```

```
System.out.println("Enter a name" );
```

```
String a = sc.next( ) ;           //Read a String
```

```
System.out.println("Enter value of n" );
```

```
int a = sc.nextInt( ) ;           //Read an interger value
```

```
System.out.println("Enter percentage" );
```

```
float a = sc.nextFloat( ) ;       //Read an floating point number
```

Operators

- ❖ Arithmetic Operators
- ❖ Relational Operators
- ❖ Logical Operators
- ❖ Increment/Decrement Operators
- ❖ Assignment Operators
- ❖ Conditional Operators
- ❖ Bitwise Operators

Arithmetic operators

Operator used to perform arithmetic operations.

Type	Symbol	Example	Description
Addition	+	5+4	Addition of 2 numbers
Subtraction	-	5-4	Subtraction of 2 numbers
Multiplication	*	5*4	Product
Division	/	5/4	Quotient is answer
Modulus	%	5%4	Remainder is answer

Write a Java program to perform Arithmetic operations.

```
class First
{
    public static void main(String s[ ] )
    {
        int a=10,b=5;
        System.out.println(a+b);
        System.out.println(a-b);
        System.out.println(a*b);
        System.out.println(a/b);
        System.out.println(a%b);
    }
}
```

Output :

15
5
50
2
0

Type casting

- We can called as **Explicit** type conversion.
- Conversion from one data type to other **done externally by the programmer**.
- This occurs when expression contains **two operands of same data type and expecting result is other data type**.
- Syntax: **(type)expression**
- **int / int=float** → expecting decimal result after dividing two integers then do type casting.

Ex: 2/3=?

(float)2/(float)3 → 2.0 /3.0 → 0.66→answer.

Relational operators

- operators used to find relationship between two operands are called relational operator.
- Relational operator result is **either true or false**

Operator	Symbol	Priority	Example	Result	Associativity
Less than	<	1	10 > 20	0	L-R
Greater than	>	1	10 < 20	1	L-R
Lesser or Equal	<=	1	10 <= 20	1	L-R
Greater or Equal	>=	1	10 >= 20	0	L-R
Equal	= =	2	10 == 10	1	L-R
Not Equal	!=	2	10 != 10	0	L-R

Logical operators

- Operator used to combine 2 or more relational expressions are called logical operators.
- Logical operators are used in comparisons and results the boolean value either true or false.

Operator	Description	Example
&& (AND)	This operator returns true if and only if both operands are true otherwise, returns false.	boolean a=true, b=false; a && b // returns false
(OR)	This operator returns true if and only if any one of the operand is true otherwise, returns false.	boolean a=true, b=false; a b // returns true
^ (XOR)	This operator returns false if and only if both operands are either true or false otherwise, returns true.	boolean a=true, b=true; a b // returns false
! (NOT)	As this is unary operator it will returns true if the operand is false and vise versa.	boolean a=true; !a //returns false

Assignment operator

- ❖ Operator used to assign some values to the variables.
 - ❖ This operator is used to assign some value present in the RHS side to LHS variable.
 - ❖ Types:
 - **Simple** Assignment: Assigning RHS value to LHS variable.
Example : C=a+b;
 - **Multiple** Assignment: if more than one variable having same value then we will prefer multiple assignment operator.
Example: C=D=a+b;
 - **Compound** Assignment: if LHS variable name and RHS first operand name is same then we will prefer compound assignment(+, -, *, /=).
- Example:** a=a+b; can be rewrite as a += b;
C=C-b; can be rewrite as C -= b;

Increment / Decrement operators

These operators are used to increment /decrement a value of the variable by one.

Operator	Description	Types	Example
Increment (++)	This operator used to increment variable value by 1	Pre-increment Y=++X ;	int X=5,Y; Y=++X; //output: X=6,Y=6
		Post- increment Y=X++;	int X=5,Y; Y=X++; //output: Y=6 X=5
Decrement (--)	This operator used to decrement variable value by 1	Pre-increment Y= -- X ;	int X=5,Y; Y= -- X; //output: X=4,Y=4
		Post- increment Y=X --;	int X=5,Y; Y=X --; //output: Y=5 X=4

Example:

```
public class One
{
    public static void main(String args[ ])
    {
        int x=10;
        System.out.println(x++);    //10 (11)
        System.out.println(++x);    //12
        System.out.println(x--);    //12 (11)
        System.out.println(--x);    //10
    }
}
```

Output :

10
12
12
10

Conditional operator

- ❖ It is also called ternary operator.
- ❖ Symbol ? and :
 - Syntax: **(expression1)?expression2:expression3**

Means, if expression1 is true then expression2 is answer.

If expression1 is false then expression3 is answer.

Ex : **(4>5)?4:5**

Here expression1 is false so 5 is answer.

Bit wise operators

- ❖ Bit wise operators are used to manipulate the bits.
- ❖ When compared to logical operators, Bit-wise operators are not used for comparisons rather these operators are used to manipulate the bits.

Operator	Symbol	Description	Example
Bitwise AND	&	This operator returns result true if and only if both operands are true.	<pre>int a=5,b=6,c; c=a&b; // 5→0101 6→0110 ----- c→ 0100=4</pre>
Bitwise OR	 	This operator returns result true if and only if any one operand is true.	<pre>int a=5,b=6,c; c=a b; // 5→0101 6→0110 ----- c→ 0111=7</pre>
Bitwise XOR	^	This operator returns result false if and only if either both operands are true or both operands are false.	<pre>int a=5,b=6,c; c=a^b; // 5→0101 6→0110 ----- c→ 0011=3</pre>
Bitwise NOT	~	As this is unary operator, if operand is true then result will be false and vice versa.	<pre>int a=6,c; c=~a; // 6→0110 ----- ~6→ 1001=9</pre>
Left shift	<<	Shift the user specified number of bits towards left.	<pre>int a=6,c ; c=a<<1; // 6→ 0110 Shift 1 bit towards left side. Output:1100=12</pre>
Right shift	>>	Shift the user specified number of bits towards right.	<pre>int a=6,c ; c=a>>1; // 6→ 0110 Shift 1 bit towards right side. Output:0011=3</pre>

Control statements

- **Conditional statements**
 - Conditional Branch statements – if ,if-else, else-if ladder,switch.
 - Conditional iterative(looping) statements – for , while, do-while.
- **Unconditional statements**
 - break, continue

if

- keyword is if
- This is used to check only one condition at a time. So it is called as one way selection statement.

Syntax: if expression is true then it will execute set of statements

```
if( expression )  
{ statements; }
```

Example:

```
int a=5 ,b=3;  
if(a>b)  
    System.out.println("a is Big:" +a); //output: a is Big:5
```

if - else

- This is 2 way selection statement.
- This is used to check 2 conditions at a time.
- if expression is true , it will execute some set of statements else ,it will executes other set of statements.

Syntax:

```
if(expression)  
{ statements; }  
else  
{ statements; }
```

Example:To check largest of 2 numbers.

```
int a=4,b=6;  
if(a>b)  
    System.out.println("a is big");  
else  
    System.out.println("b is big");
```

Write a Java program to check given number is even or odd .

```
class First  
{  
    public static void main( String s[ ] )  
    {  
        int n = 13;  
  
        if (n%2== 0)  
        {  
            System.out.println("even number");  
        }  
        else  
        {  
            System.out.println ("odd number");  
        }  
    }  
}
```


else if ladder

- ❖ This is a Multiway selection statement.
- ❖ That is more than one condition we can check at a time.

Syntax:

```
if(expression1)
{   statements;   }
else if(expression2)
{   statements;   }
else if(expression3)
{   statements;   }
.....
.....
else
{   statements;   }
```

Example: To check largest of 3 numbers.

```
int a=4,b=6,c=8;

if(a>b && a>c)
    System.out.println ("a is big");
else if(b>a && b>c)
    System.out.println ("b is big");
else
    System.out.println ("c is big");
```

Nested if statement

“An if or if –else statements present within another if or if-else statement is called nested if statement”.

Syntax:

```
if(condition1)
{
    if(condition 2)
    {
        Statements;
    }
}
```

Example: to check largest of 3 numbers

```
if(a>b)
{
    if(a>c)
    {
        System.out.println ("a is big");
    }
}
else if(b>c)
{
    System.out.println ("b is big");
}
else
{
    System.out.println ("c is big");
}
```

switch statement

- ❖ switch statement is used to make selection between many alternatives.
- ❖ Instead of else-if statement we are using switch to reduce program complexity.

Syntax:

```
switch( choice)
{
    case 1: statements;
           break;
    case 2: statements;
           break;
    .....
    .....
    default : System.out.println("give error message");
}
```

// Java program to perform Arithmetic operations using switch (Lab Program)

```
package jk;
import java.util.Scanner;
public class First
{
    public static void main(String[] args)
    {
        int a,b;
        String ch;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter a and b values");
        a=sc.nextInt();
        b=sc.nextInt();
        System.out.println("Enter choice");
        ch=sc.next();
        switch(ch)
        {
            case "+": System.out.println(a+b); break;
            case "-": System.out.println(a-b); break;
            case "*": System.out.println(a*b); break;
            case "/": System.out.println((float)a/(float)b); break;
            case "%": System.out.println(a%b); break;
            default: System.out.println("Invalid operator"); break;
        }
    }
}
```

```
Enter a and b values
10
3
Enter choice
/
3.3333333
```

Looping statements

- ❖ We can also known as **repetition** or **iteration** statements.
- ❖ A set of statements may have to be repeatedly executed for specified number of time or till the condition satisfied called looping statements.

Types or three ways of Loops

- **while loop**
- **for loop**
- **do-while loop**

while loop

- A set of statements may have to be repeatedly executed till the condition is true once the condition becomes false control comes out of the loop.
- It is **pre-test loop** or **top testing looping** and hence condition testing occurs at the top.

Syntax:

```
initialization;
while(condition)
{
    Statements;
    increment / decrement ;
}
```

Example:

//Java program to print numbers from 1 to n

```
package jk;

import java.util.Scanner;

public class Numbers
{
    public static void main(String[] args)
    {
        int n,i;

        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();
        i=1;
        while(i<=n)
        {
            System.out.println(i);
            i++;
        }
    }
}
```

```
enter the value of n
5
1
2
3
4
5
```

// Java program to find sum of natural numbers

```
package jk;

import java.util.Scanner;

public class Sum
{
    public static void main(String[] args)
    {
        int n, sum,i;
        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        sum=0;
        i=1;

        while(i<=n)
        {
            sum =sum+i;
            i++;
        }
        System.out.println("sum="+sum);
    }
}
```

```
enter the value of n
5
sum=15
```

for loop

- ❖ A set of statements may have to be repeatedly executed for specified number of time or till the condition satisfied.
- ❖ Once specified number of times loop executed, control comes out of the loop.
- ❖ This method is also top testing loop i.e condition is test at the top and body of the loop executed only if the condition is true.

Syntax:

```
for(init ; condition ;incr/decr)
{
    Statement 1;
    Statement 2;
    .....
    Statement n;
}
```



Where,
init → is a initial value
condition → check the condition
incr/decr → increment/ decrement

Examples:

//Java Program to print squares of a natural numbers.

```
package jk;

import java.util.Scanner;

public class Square
{
    public static void main(String[] args)
    {
        int n,sum,i;
        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        sum=0;
        for(i=1;i<=n;i++)
        {
            sum=sum+i*i;
        }
        System.out.println("sum="+sum);
    }
}
```

```
enter the value of n
4
sum=30
```

//Java Program to find factorial of a number.

```
package jk;

import java.util.Scanner;

public class Fact
{
    public static void main(String[] args)
    {
        int n, i , fact;
        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        fact=1;
        for(i=1;i<=n;i++)
        {
            fact= fact*i;
        }
        System.out.println(fact);
    }
}
```

```
enter the value of n
5
120
```

//Java Program to print the first n terms of Fibonacci series. (Lab Program)

```
package jk;
import java.util.Scanner;
public class Fib
{
    public static void main(String[] args)
    {
        int n,i,first,second,next;

        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        first=0;
        second=1;

        System.out.println("Fibonacci numbers are:\n");

        System.out.print(first+"\t"+second);

        for(i=2;i<=n-1;i++)
        {
            next=first+second;
            System.out.print("\t"+next);
            first=second;
            second=next;
        }
    }
}
```

```
enter the value of n
6
Fibonacci numbers are:
0      1      1      2      3      5
```

// Java Program to find given number is prime or not

```
package jk;
import java.util.Scanner;
public class Prime
{
    public static void main(String[] args)
    {
        int n , counter = 0,i;

        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        for(i=1; i<=n ; i ++ )
        {
            if( n % i == 0 )
                counter ++ ;
        }

        if(counter == 2)
            System.out.println("The Given Number " +n+ " is a prime" ) ;
        else
            System.out.println("The Given Number " +n+ " is not a prime");
    }
}
```

```
enter the value of n
6
The Given Number 6 is not a prime
enter the value of n
13
The Given Number 13 is a prime
```

do while loop

- ❖ do while loop is similar to while loop but condition is test at bottom.
- ❖ This loop is called post-testing or bottom- testing loop.
- ❖ As bottom testing loop , all the statements within body are **executed at least once** .

Syntax:

```
do
{
    Statements;
}while(condition);
```

➔ semicolon (;) is must at the end of do-while loop.

Example:

// Java program to find sum of natural numbers

```
package jk;

import java.util.Scanner;

public class DOWHILE
{
    public static void main(String[] args)
    {
        int n, sum,i;
        System.out.println("enter the value of n");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();

        sum=0;
        i=1;

        do
        {
            sum =sum+i;
            i++;
        } while(i<=n) ;

        System.out.println("sum="+sum);
    }
}
```

```
enter the value of n
5
sum=15
```

foreach loop

- ❖ The Java for-each loop or enhanced for loop.
- ❖ It provides an alternative approach to **traverse the array** in Java.
- ❖ The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable.
- ❖ It is known as the for-each loop because it traverses each element one by one.
- ❖ The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order.
- ❖ But, it is recommended to use the Java for-each loop for traversing the elements of array because it makes the code readable.

Syntax

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array.

```
for (type var_name : Array_name)
{
    //Body of the loop
}
```

Example:

```
package jk;

public class Temp
{
    public static void main(String[] args)
    {
        int a[] = {1,2,3,4,5,6};

        for(int i:a)
            System.out.println(i);
    }
}
```

```
1
2
3
4
5
6
```


break

- break we can use in **switch** statement(branching) and also in **looping** statements.
- Use break in switch statement to terminate it.
- Use break in looping statement, to make control come out of that loop.

Examples : break used in loop

```
for(i=1;i<=5;i++)
{
    if (i= = 3)
        break;
    System.out.print("\t"+i);
}
Output:
1 2
means control break when i is 3
```

break used in switch

```
switch(operator)
{
    case '+': System.out.println(a+b);
              break;
    case '-': System.out.println(a-b);
              break;
}
```

continue

- continue used only in looping statements.
- This keyword used to **skips** execution of **present** statement and **continue** execution from **next** statement.

```
for(i=1;i<=5;i++)
{
    if (i= = 3)
        continue;
    System.out.print ("\t"+i);
}
Output:
1 2 4 5
```

MODULE 2

CLASSES

METHODS

- ❖ CLASS
- ❖ OBJECTS
- ❖ REFERENCE VARIABLE
- ❖ METHODS
- ❖ CONSTRUCTORS
- ❖ GARBAGE COLLECTION
- ❖ METHOD OVERLOADING
- ❖ CONSTRUCTOR OVERLOADING
- ❖ OBJECTS AND METHODS
- ❖ NESTED CLASSES
- ❖ ACCESS MODIFIER
- ❖ SETTER AND GETTER

Class

- Keyword is **class**.
- class is a collection of fields, methods, constructors, and certain properties.
- class acts like a blueprint.
- When defining class , it should starts with keyword class followed by name of the class; and the class body, enclosed by a pair of curly braces.

Syntax:

```
<public> class class_name  
{  
    //variables(fields or members)  
  
    // methods or functions  
}
```

Example: to find area of a circle.

```
class First  
{  
    int r;  
    public static void main( String s[ ] )  
    {  
        First obj=new First( ) ;  
        obj.r=2;  
        System.out.println(MATH.PI * obj.r * obj.r );  
    }  
}
```

Where, public is an Access modifier of the class.

In the above example class definition, class name is First ,r is a member(instance variable means need to create object to access) which is used to find area of circle, and main is a method or function to perform operation and Math.PI gives the mathematical constant that is 3.1412.

Object

- Object is a instance of the class.
- Object is used to access members and member functions.
- Object should be created for instance variable, but for local variables and static variable object creation is not required.
- Object is created using **new** keyword , which allocates the memory for object when its created.

Syntax: classname objectname = new classname() ;

Example: **First obj = new First () ;**

Where First is class name followed by object name **obj**(any identifier) followed by assignment operator followed by new keyword followed by constructor (we will discuss later).

Example:

```
public class First
{
    int a;                                //member

    void display( )                       //method
    {
        System.out.println("your inside a display method");
    }

    public static void main( String s[ ] )
    {
        First obj=new First( ) ;         //object creation
        obj.a=2;                          //access variable a using obj
        System.out.println(obj.a);
        obj.display();                    //access display method using obj
    }
}
```

Output:

```
2
your inside a display method
```

In the above example, **a** is instance variable and **display** is a method.

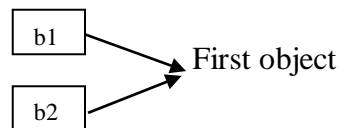
obj is created as instance of the class **First** so that we can access instance variable and method as shown in the above program.

Reference variables

- A reference variable is used to access the object of a class. Reference variables are created at the program compilation time.
- Reference variable is just a alias name for object.
- Object reference variables act differently than you might expect when an assignment takes place.

For example,

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



We might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, we might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the same object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

Methods

- We already know that class contains members (instance variables) and member functions (methods).
- A method is a module or function, which is used to declare the elements of its structure.
- A method is used to reduce the whole problem into smaller problems to make it easy programming.

```
<Access modifier><Return type><Method name>(parameter List)
{
    // Method body
}
```

Following are the elements of a method –

Access modifier: this determines the visibility of a variable or a method from another class.

Return type: A method may return a value. If a method is not returning any value then method return data type should be void, if method returning integer value then method return type should be int, etc.,

Method name: It's a unique identifier and it's case sensitive. It cannot be same as any other identifier.

Parameter List: enclosed between parenthesis. Parameter List is optional that is, a method may contain no parameters.

Method Body: this contains the set of instructions need to complete the required activity.

Scope

- **Local Scope:**
 - The variables which are declared inside one method, we cannot use the same variables in other method.
 - The left and right curly braces that form the body of a method create scope.
- **Class Scope:**
 - The variable which are declared inside the class and outside all the methods, which are used in any methods known as Class Scope.

We can write method in different ways depends on return value –

- **Writing method –**
 - **Without parameters and without return value.**
 - **Without parameters and with a return value.**
 - **With parameters and without return value.**
 - **With parameters and with return value.**

Writing a Method without parameters and without return value

- In this type A Method will not accept any parameters and not return any value to the Main method.

Example: program to find the Sum of 2 numbers

```
public class First
{
    public void add()
    {
        int a=20, b=30, sum;
        sum=a+b;
        System.out.println("Addition is: " +sum );
    }
    public static void main(String[] args)
    {
        First obj = new First( ) ;
        obj.add( ) ;
    }
}
```

Addition is: 50

Writing a Method without parameters and with return value

- In this type A Method will not accept any parameters and return value to the Main method.

Example: program to find the Sum of 2 numbers

```
public class First
{
    public int add()
    {
        int a=20, b=30, sum;
        sum=a+b;
        return sum;
    }
    public static void main(String[] args)
    {
        First obj = new First( ) ;
        int sum=obj.add( ) ;
        System.out.println("Addition is: " +sum );
    }
}
```

Addition is: 50

Writing a Method with parameters and without return value

- In this type A Method will accept parameters and not return any value to the Main method.

Example: program to find the Sum of 2 numbers

```
public class First
{
    public void add(int a,int b)
    {
        int sum;
        sum=a+b;
        System.out.println("Addition is: " +sum );
    }
    public static void main(String[] args)
    {
        First obj = new First( ) ;
        obj.add(20,30 ) ;
    }
}
```

Addition is: 50

Writing a Method by using with parameters and return value

- In this type A Method will accept parameters and return value to the Main method.

Example: program to find the Sum of 2 numbers

```
public class First
{
    public int add(int a,int b)
    {
        int sum;
        sum=a+b;
        return sum;
    }
    public static void main(String[] args)
    {
        First obj = new First( ) ;
        int sum =obj.add(20,30 ) ;
        System.out.println("Addition is: " +sum );
    }
}
```

Addition is: 50

Constructors

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to **initialize** the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type.
3. A Java constructor cannot be static, final, abstract.

Types of constructors:

- 1. Default or parameter less constructor.
- 2. Parameterized constructor.
- 3. Copy constructor.

Default / parameter less constructor

- If a constructor method doesn't have any parameters then we call it as parameter less constructor or default constructor.
- These methods can be defined by the programmer explicitly, or else will be defined implicitly provided by the compiler.
- In Implicit constructors, the default value for integer is 0, for boolean is false, for string is null., etc.
- Syntax: **[Access_Modifier]<class_name> () { }**

```
public class First
{
    int i; boolean b;

    public First()                //parameter less constructor --here , public is optional
    {
        i=800;
    }

    public static void main(String[] args)
    {
        First obj = new First( ) ;

        System.out.println(obj.i+ " and "+ obj.b );
    }
}
```

800 and false

Parameterized constructor

- If a Constructor method is defined with parameters, we call that as parameterized constructor.
- Parameterized constructor should be defined by the programmer but never defined by the compiler implicitly.

Example:

```
public class First
{
    int a, b ;

    First(int x, int y)           //parameterized constructor
    {
        a=x; b=y;
    }

    public static void main(String[] args)
    {
        First obj1 = new First(10,20) ;
        First obj2 = new First(30,40) ;

        System.out.println(obj1.a+ " "+ obj1.b );
        System.out.println(obj2.a+ " "+ obj2.b );
    }
}
```



```
10 20
30 40
```

Copy Constructor

- If we want to create multiple instances with the same value then we can use copy constructor.
- Copy constructor is used to copy data of one object into another object.

In copy constructor, the constructor takes the same class as a parameter to it.

```
public class First
{
    int a, b ;

    First(int x, int y)           //parameterized constructor
    {
        a=x; b=y;
    }

    First(First obj)             //copy constructor
    {
        a = obj.a; b = obj.b;
    }

    public static void main(String[] args)
    {
        First obj1 = new First(10,20) ;
        First obj2 = new First(obj1) ;

        System.out.println(obj1.a+ " "+ obj1.b );
        System.out.println(obj2.a+ " "+ obj2.b );
    }
}
```



```
10 20
10 20
```

this keyword

- In Java, this is a **reference variable** that refers to the current object.
- Uses –
 - ❖ this can be used to refer current class instance variable.
 - ❖ this can be used to invoke current class method (implicitly)
 - ❖ this() can be used to invoke current class constructor.
 - ❖ Etc.,
- The **this** keyword can be used to refer current class instance variable. If there is ambiguity (confusion) between the instance variables and local parameters, this keyword resolves the problem of **ambiguity**.
- **Example:**
//without using this keyword

```
public class First
{
    int a, b ; String name;

    First(int a, int b, String name)
    {
        a=a; b=b; name=name;
    }

    public static void main(String[] args)
    {
        First obj1 = new First(10,20,"kiran") ;

        System.out.println(obj1.a+ " "+ obj1.b + " "+obj1.name );
    }
}
```

```
0 0 null
```

In the above program , instance variables and local variables are with same name so, compiler will returns default values of their datatypes.

//Using this keyword

```
public class First
{
    int a, b ; String name;

    First(int a, int b, String name)
    {
        this.a=a; this.b=b; this.name=name;
    }

    public static void main(String[] args)
    {
        First obj1 = new First(10,20,"kiran") ;

        System.out.println(obj1.a+ " "+ obj1.b + " "+obj1.name );
    }
}
```

```
10 20 kiran
```

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, we might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as **C++**, dynamically allocated objects must be manually released by use of a **delete** operator. **Java** takes a different approach; it handles deallocation for you **automatically**.
- The technique that accomplishes this is called garbage collection. It works like this: 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. There is no explicit need to destroy objects as in **C++**.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

Method overloading

- Java allows us to create **more than one method with same name, but with different parameter** list and different definitions. This is called method overloading.
- Method overloading is used when methods are required to perform similar tasks but using different input parameters. Overloaded methods must differ in number and/or type of parameters they take.
- This enables the compiler to decide which one of the definitions to execute depending on the type and number of arguments in the method call.

Example:

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

        obj.Addition(10, 20);
        obj.Addition(10, 20, 30);
    }
}
```

30
60

Constructor Overloading

The process of creating more than one constructor with same name, which is similar to class name, but with different parameters is called constructor overloading.

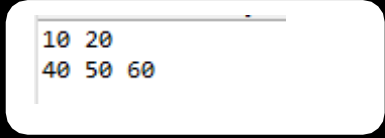
Example:

```
public class First
{
    int a, b ,c ;

    First(int x, int y)
    {
        a=x; b=y;
    }
    First(int x, int y,int z)
    {
        a = x; b = y; c = z;
    }

    public static void main(String[] args)
    {
        First obj1 = new First(10,20) ;
        First obj2 = new First(40,50,60) ;

        System.out.println(obj1.a+ " "+ obj1.b );
        System.out.println(obj2.a+ " "+ obj2.b+ " " +obj2.c );
    }
}
```



```
10 20
40 50 60
```

Objects and methods (Using Objects as Parameters to methods)

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

```
public class First
{
    int a, b ,c ;

    First(int x, int y)                //constructor
    {
        a=x; b=y;
    }

    void addition(First obj1)          //method with object as a parameter
    {
        System.out.println(obj1.a+obj1.b);
    }
    public static void main(String[] args)
    {
        First obj = new First(10,20) ;

        obj.addition(obj);
    }
}
```



```
30
```

Objects and arrays

- Array is a set of similar type of elements.
- In Java, **Arrays are implemented as objects**.
- Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable.
- All arrays have this variable, and it will always hold the size of the array.
- Here is a program that demonstrates this property:

```
public class First
{
    public static void main(String[] args)
    {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};

        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
    }
}
```

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

- As shown above, the size of each array is displayed. Keep in mind that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
- The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
- The following program illustrates how to define and use an inner class.

```
public class Temp
{
    int a=10;

    class inner
    {
        void display( )
        { System.out.println("value of a is "+a); }
    }

    void test( )
    {
        inner obj=new inner();
        obj.display();
    }

    public static void main( String s[ ] )
    {
        Temp obj=new Temp();
        obj.test();
    }
}
```

```
value of a is 10
```

Access Modifiers (Access Specifiers)

- The access modifier's in Java specifies the accessibility or scope of a field, method, constructor, or class.
- We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Types:

- ❖ Public
- ❖ Private
- ❖ Protected

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

Public

- The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
- It has the widest scope among all other modifiers.

Example:

```
package jk;                                     //save as A.java in jk package

public class A
{
    public void msg()                           //public method
    {
        System.out.println("Hello Kiran");
    }
}
```

```
package k1;                                     //save as B.java in k1 package

import jk.*;                                    //import jk package to access public method

public class B
{
    public static void main(String[] args)
    {
        A obj=new A();
        obj.msg();
    }
}
```

Hello Kiran

Private

- The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Example:

```
package jk;                                //save as A.java in jk package

public class A
{
    private void msg()                    //private method
    {
        System.out.println("Hello Kiran");
    }
}
```

```
package k1;                                //save as B.java in k1 package
import jk.*;                               //import jk package to access public method

public class B
{
    public static void main(String[] args)
    {
        A obj=new A();
        obj.msg();
    }
}
```

Compile error : cant access private method outside the class

Similarly we can't access private variables(members) also from outside the class.

Protected

- The protected access modifier is accessible within package and outside the package but through **inheritance** only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example:

```
package jk;                                //save as A.java in jk package

public class A
{
    protected void msg()                //protected method
    {
        System.out.println("Hello Kiran");
    }
}
```

```
package k1;                                //save as B.java in k1 package
import jk.*;                               //import jk package to access public method

public class B extends A                  //class B inherits class A
{
    public static void main(String[] args)
    {
        B obj=new B();                    //create object for child class
        obj.msg();
    }
}
```

Hello Kiran

Setters and getters

- Getter and Setter are methods used to protect your data and make your code more secure.
- **Getter returns** the **value**(that is, it returns the value of data type int, String, double, float, etc.) While **Setter sets** or updates the **value**. It sets the value for any variable used in a class's programs.
- **Private** members (variables) of one class can access in another class using **set** and **get** methods.

Example:

```
package k1;                                //save as C1.java in package k1

public class C1
{
    private String name;                    //name variable is private

    public String getName()
    { return name; }

    public void setName(String N)
    {
        name = N;                          // This keyword refers to current instance itself
    }
}
```

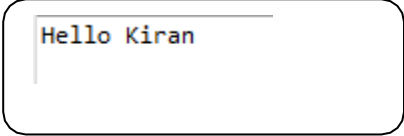
```
package k1;                                //save as C2.java in package k1

public class C2 {

    public static void main(String[] args)
    {
        C1 obj= new C1();                  //create object for C1
        obj.setName("Hello Kiran");         //"Hello kiran" set to private variable name in class c1

        System.out.println(obj.getName());  //get private value and displayed
    }
}
```

Output:



Hello Kiran

Lab Programs:

2.a. Write a Java program to demonstrate method overloading to create a class called Calculator with three methods named Demo with different type of parameters and print the computed results.

```
package jk;

import java.util.Scanner;

public class Calculator
{
    int a;

    void Demo(int a,int b)
    {
        System.out.print((a+b)+"\t"+(a-b)+"\t"+(a*b)+"\t"+((float)a/(float)b)+"\n");
    }
    void Demo(int a,int b,int c)
    {
        System.out.print((a+b+c)+"\t"+(a-b-c)+"\t"+(a*b*c)+"\t"+((float)a/(float)b/(float)c)+"\n");
    }
    void Demo(int a,int b,int c,int d)
    {
        System.out.print((a+b+c+d)+"\t"+(a-b-c-d)+"\t"+(a*b*c*d)+"\t"+((float)a/(float)b/(float)c/(float)d));
    }

    public static void main( String s[ ] )
    {
        Calculator obj = new Calculator();
        int a,b,c,d;

        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the values of a,b,c,d");

        a=sc.nextInt();    b=sc.nextInt();
        c=sc.nextInt();    d=sc.nextInt();

        obj.Demo(a,b);
        obj.Demo(a,b,c);
        obj.Demo(a,b,c,d);
    }
}
```

Output:

```
Enter the values of a,b,c,d
1 2 3 4
3      -1      2      0.5
6      -4      6      0.16666667
10     -8      24     0.041666668
```

2.b. Write a Java program to demonstrate constructor overloading to calculate the area of a rectangle and circle.

```
package jk;

public class Area
{
    int r,l,b;
    Area (int r)
    {
        this.r=r;
    }
    Area (int l,int b)
    {
        this.l=l;
        this.b=b;
    }
    public static void main(String[] args)
    {
        Area obj1=new Area(2);
        Area obj2=new Area(3,4);

        System.out.println("Area of a Circle:");
        System.out.println(3.142*obj1.r*obj1.r);
        System.out.println("Area of a Rectangle:");
        System.out.println(obj2.l*obj2.b);
    }
}
```

//we can also write without using this keyword

```
Area (int rad)
{
    r = rad ;
}
Area(int len , int br )
{
    l=len ; b=br ;
}
```

Output:

```
Area of a Circle:
12.568
Area of a Rectangle:
12
```

MODULE 3

INHERITANCE
INTERFACES
PACKAGES

- ❖ INHERITANCE BASICS
- ❖ USING SUPER
- ❖ CREATING MULTI-LEVEL HIERARCHY
- ❖ METHOD OVERRIDING
- ❖ USING ABSTRACT CLASSES
- ❖ USING FINAL
- ❖ INTERFACES.
- ❖ PACKAGES: ACCESS PROTECTION, IMPORTING PACKAGES

Inheritance

- One of the most important concepts in object-oriented programming is inheritance.
- To inherit a class, we simply incorporate the definition of one class into another by using the **extends** keyword.
- Definition:
“**Creating a new class (child class) from existing class (parent class) is called as inheritance**”.

Or

“**Acquiring (taking/Consuming) the properties of one class into another class is called inheritance.**”

Or

“**When a new class needs same members as an existing class, then instead of creating those members again in new class, the new class can be created from existing class, which is called as inheritance.**”

Main advantage of inheritance is **reusability** of the code and Method overriding (polymorphism) can be achieved.

Base class: is the class from which features are to be inherited into another class.

Syntax:

```
class Base_Class
{
    Members of class
}
```

Derived class: it is the class in which the base class features are inherited.

Syntax:

```
class Derived_class extends Base_class
{
    Members of class
}
```

Example:

```
class One //parent class
{
    int a=10;
}
class Second extends One //child class
{
    public static void main(String[] args)
    {
        Second obj=new Second();
        System.out.print(obj.a);
    }
}
```

10

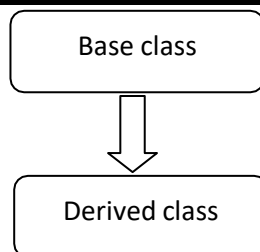
As shown above child class can access properties (variables or methods) of the parent class.

Types of Inheritance:

Inheritance can be classified into 5 types

1. Single Inheritance
2. Multi-Level Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance

Single Inheritance



When a single derived class is created from a single base class then the inheritance is called as single inheritance.

Syntax:

```
class A
{
    ----
    ----
}
class B extends A
{
    ----
    ----
}
```

Example:

```
package jk;

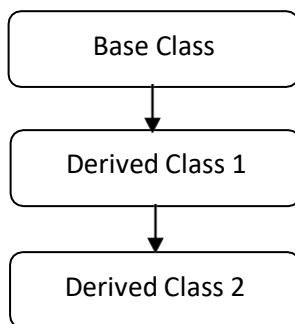
class One                                //save as Test.java
{
    void display1()
    {
        System.out.println("Ha ha ha");
    }
}
class Two extends One
{
    void display2()
    {
        System.out.println("Hi hi hi");
    }
}

class Test
{
    public static void main(String[] args)
    {
        Two obj = new Two();
        obj.display1();
        obj.display2();
    }
}
```

```
Ha ha ha
Hi hi hi
```

Multi Level Inheritance

When a derived class is created from another derived class, then that inheritance is called as multi-level inheritance.



Syntax:

```
class A
{-----}
```

```
class B extends A
{-----}
```

```
class C extends B
{-----}
```

Example:

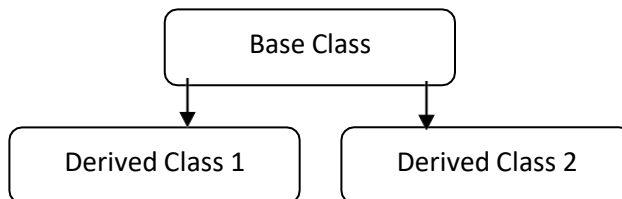
```
package jk;

class One
{
    void display1()
    {
        System.out.println("Ha ha ha");
    }
}
class Two extends One
{
    void display2()
    {
        System.out.println("Hi hi hi");
    }
}
class Three extends Two
{
    void display3()
    {
        System.out.println("La La La");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Three obj=new Three();
        obj.display1();
        obj.display2();
        obj.display3();
    }
}
```

```
Ha ha ha
Hi hi hi
La La La
```

Hierarchical Inheritance

When more than one derived class is created from a single base class, then that inheritance is called as hierarchical inheritance.



Syntax:

```
class A
{ - - - - }
```

```
class B extends A
{ - - - - }
```

```
class C extends A
{ - - - - }
```

```
package jk;

public class One
{
    void display1()
    {
        System.out.println("Ha ha ha");
    }
}

public class Two extends One
{
    void display2()
    {
        System.out.println("Hi hi hi");
    }
}

public class Three extends One
{
    void display3()
    {
        System.out.println("La La La");
    }
}

public class Test {

    public static void main(String[] args)
    {
        Two obj1=new Two();
        Three obj2=new Three();

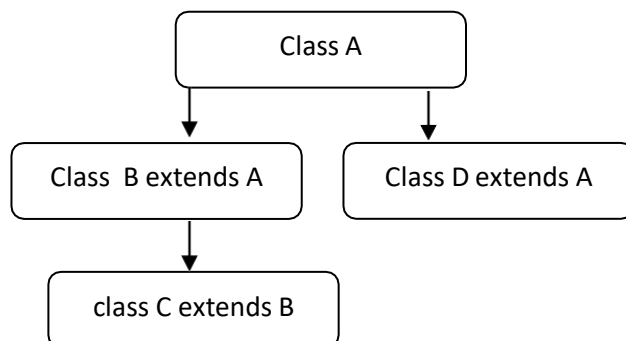
        obj1.display1();    // we can access properties of class one and two
        obj1.display2();

        obj2.display1();    // we can access properties of class one and three
        obj2.display3();
    }
}
```

```
Ha ha ha
Hi hi hi
Ha ha ha
La La La
```

Hybrid Inheritance

Any combination of single, hierarchical, multi-level and Multiple inheritances is called as hybrid inheritance.



```
package jk;

public class One
{
    void display1()
    {
        System.out.println("\n Ha ha ha");
    }
}

public class Two extends One
{
    void display2()
    {
        System.out.println("Hi hi hi");
    }
}

public class Three extends Two
{
    void display3()
    {
        System.out.println("La La La");
    }
}

public class Four extends One
{
    void display4()
    {
        System.out.println("wow wow wow");
    }
}

public class Test {

    public static void main(String[] args)
    {

        Three obj1=new Three();
        Four obj2=new Four();

        obj1.display1();    // we can access properties of class one, two, three
        obj1.display2();    //Multilevel Inheritance
        obj1.display3();

        obj2.display1();    // we can access properties of class one and four
        obj2.display4();    //single Inheritance

    }
}
```

```
Ha ha ha
Hi hi hi
La La La

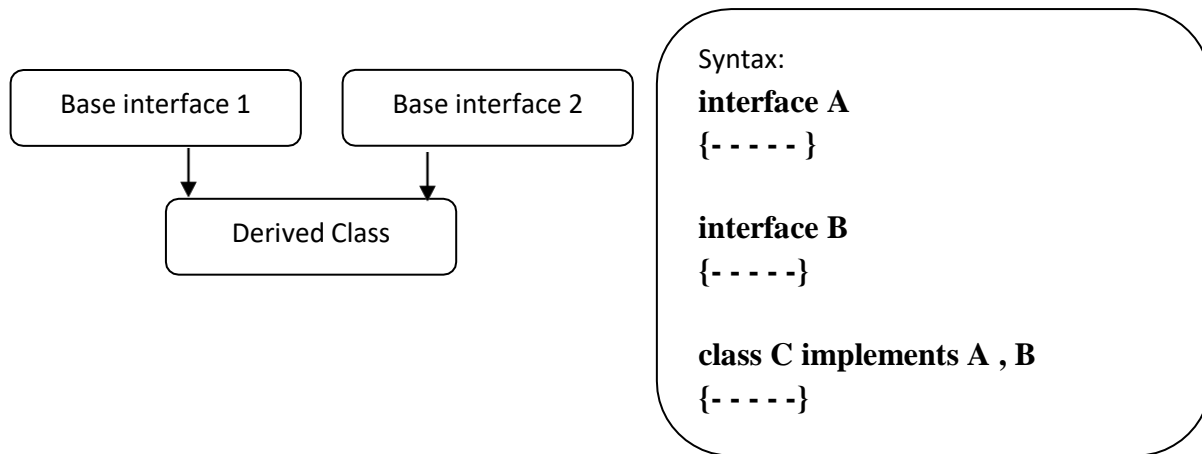
Ha ha ha
WOW WOW WOW
```

Note: Students should note that the concept explained above is multi level hierarchy (if students got question in exam Explain how to create multilevel hierarchy)

Multiple Inheritances

In multiple inheritance, one class can have more than one super class (Base class) and inherits features from all parent classes.

- **Note:** Java doesn't support Multiple Inheritance with class and it can be achieved by using **Interfaces**.
- **Note:** Students need to explain in the exam by using interface if marks are more.



super keyword

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

super keyword in java we can use in different ways

- ❖ **super** can be used to refer immediate **parent class instance variable**.
- ❖ **super** can be used to invoke immediate **parent class method**.
- ❖ **super()** can be used to invoke immediate **parent class constructor**.

super can be used to refer immediate parent class instance variable.

- ❖ We can use super keyword to access the data member or field of parent class.
- ❖ It is used if parent class and child class have same fields.

```
package superexample;

public class One
{
    int a=10;           //parent class variable name a=10
}

public class Two extends One
{
    int a=20;           //child class variable name a=20
    void display()
    {
        System.out.println("child class variable a=" + a);
        System.out.println("parent class variable a=" + super.a);
    }

    public static void main(String[] args)
    {
        Two obj=new Two();
        obj.display();
    }
}
```

child class variable a=20
parent class variable a=10

super can be used to invoke immediate parent class method.

- ❖ The super keyword can also be used to invoke parent class method.
- ❖ It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
package superexample;

public class One
{
    void display()
    {
        System.out.println("My name is kiran");
    }
}

public class Two extends One
{
    void display()
    {
        super.display();
        System.out.println("Updated: My name is Kiran Jayaram");
    }
}

public static void main(String[] args)
{
    Two obj=new Two();
    obj.display();
}
```

```
My name is kiran
Updated: My name is Kiran Jayaram
```

super can be used to invoke immediate **parent class constructor**.

- ❖ The super keyword can also be used to invoke the parent class constructor.

```
package superexample;

public class One
{
    One()
    {
        System.out.println("class one is created ");
    }
}

public class Two extends One
{
    Two()
    {
        super(); //access parent class constructor
        System.out.println("Class two is created");
    }
}

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

```
class one is created
Class two is created
```

Method overriding

- The process of re-implementing the parent class method under the child class with same name and same number of parameters is called method overriding.
- Method overriding means same method names with same signatures(parameters) in different classes.
- Method overriding is used to provide the specific implementation of a method which is already provided by its super class.
- Method overriding is used for runtime polymorphism.

```
package jk;

public class One
{
    void display()
    {
        System.out.println("My name is kiran");
    }
}

public class Two extends One
{
    void display()
    {
        super.display();
        System.out.println("Updated: My name is Kiran Jayaram");
    }
}

public static void main(String[] args)
{
    Two obj=new Two();
    obj.display();
}
```

```
My name is kiran
Updated: My name is Kiran Jayaram
```

Differences between Method overloading and Method overriding

Method overloading	Method overriding
It is used to implement multiple method with same name and different number of parameters	It is used to implement multiple method with same name and same number of parameters
This can be performed in one single class as parent/child classes also.	This can be performed only in child class
This is all about providing multiple behavior to a method.	This is all about changing the behavior of the method.
Inheritance not required to implement.	Inheritance is required to implement.
Example:	Example:

<pre> class A { public void Display(int x, int y) { System.out.println(x+y); } public void Display(int x,int y, int z) { System.out.println (x+y+z); } } class Test { public static void main(String s[]) { A obj = new A(); obj.Display(10,20); obj.Display(10,20,30); } } </pre> <p>Output: 30 60</p>	<pre> class A { public void Display(int x, in y) { System.out.println (x+y); } } class B extends A { public void Display(int x, int y) { System.out.println (x*y); } } class Test { public static void main(String s[]) { B obj= new B(); Obj.Display(10,20); } } </pre> <p>Output: 200</p>
--	---

Abstract classes

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract methods (method without body) and non-abstract methods (method with the body).
- Abstract class cannot be instantiated.
- Abstract method:
 - ❖ A method without method body is known as abstract method.
 - ❖ It contains only declaration of the method.
 - ❖ We can define abstract method using abstract keyword.
 - ❖ Abstract method must be implemented in child class.

Note: Abstraction is the process of hiding the implementation details and showing only functionality to the user.

Abstraction is achieved in java using 2 ways

- ❖ Using abstract class we can achieve partial abstraction because this class may contains non-abstract methods also
- ❖ Using interface we can achieve 100% abstraction because interface contains only abstract classes.

```
package jk;

public abstract class One
{
    abstract void display1();           //abstract method

    void display2()                     //Non abstract method
    {
        System.out.println("This is non abstract method from parent class");
    }
}

public class Two extends One
{
    void display1()
    {
        System.out.println("Implementing abstract method in child class");
    }

    public static void main(String[] args)
    {
        One obj=new Two();              //Note: we can't write like    One obj=new One()
        obj.display1();                  //because abstract class One can't be instantiated
        obj.display2();
    }
}
```

Implementing abstract method in child class
This is non abstract method from parent class

final keyword

- ❖ final keyword is used to restrict the user to change value of the variable or method or class.
- ❖ That is, if the variable is declared as final then we cannot change throughout the program.

final variable

```
package jk;

public class Two
{
    final int a=20 ;                    //a is final so we can't change later
    void display()
    {
        a=10;                          //unable to change value
        System.out.println(a);
    }

    public static void main(String[] args)
    {
        Two obj=new Two();
        obj.display();
    }
}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The final field Two.a cannot be assigned

final method

If we make any method as final, we cannot override it.

```
package jk;

public class One
{
    final void display()
    {
        System.out.println("This is final method and you can't override");
    }
}

public class Two extends One
{
    void display()           // we can't override final method
    {
        System.out.println("we can't override");
    }
    public static void main(String[] args)
    {
        Two obj = new Two();
        obj.display();
    }
}
```

Output:

Compile Time Error

final class

If we make any class as final, we cannot Inherits it.

```
package superexample;

public final class One
{
    void display()
    {
        System.out.println("This is final class");
    }
}

public class Two extends One           //we can't extends final class One
{
    public static void main(String s[])
    {
        //Body
    }
}
```

Output:

Compile Time Error

Interfaces

An Interface is a collection of abstract methods (methods without definition / only method declarations) and abstract methods should be **implemented** by the derived/child class.

- Keyword here we used is , **implements** in child class instead of **extends** keyword
- In the interface, method declaration is not used abstract modifier i.e by default its abstract.
- Interface should not contain any Non-abstract methods (methods with body).
- All interface members by default public.
- By using Interface we can achieve 100% abstraction because interface contains only abstract methods.
- By using Interface we can achieve Multiple inheritance.

Syntax:

```
interface interfaceName
{
    Method declarations ;           //only method protocols- No Implementation
}
```

Example:

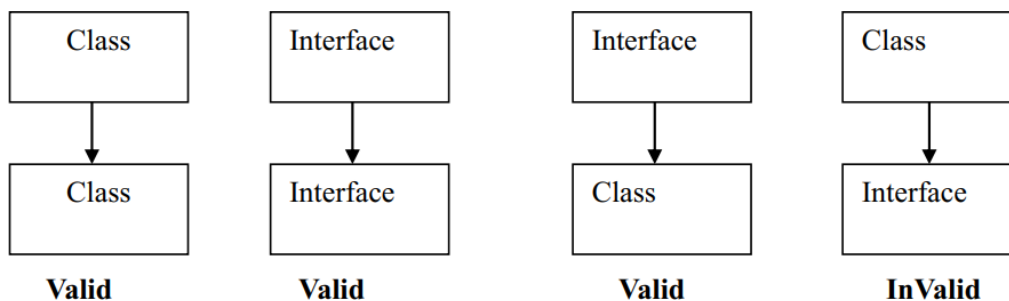
```
package hi;

public interface One
{
    void display();
}

public class Two implements One
{
    public void display()
    {
        System.out.println("abstract method of interface is implemented in child class");
    }
    public static void main(String[] args)
    {
        Two obj=new Two();
        obj.display();
    }
}
```

abstract method of interface is implemented in child class

Note:



Keywords: extends

extends

implements

can't implement nor extends

Multiple Inheritance using Interface

Java does not support multiple inheritance with class. However, we can use interfaces to implement multiple inheritance.

The following program demonstrates this:

```
package hi;

public interface One
{
    void display1();
}

public interface Two
{
    void display2();
}

public class Three implements One , Two
{
    public void display1()
    {
        System.out.println("Implementing method display1 in child class Three");
    }
    public void display2()
    {
        System.out.println("Implementing method display2 in child class Three");
    }

    public static void main(String[] args)
    {
        Three obj=new Three();
        obj.display1();
        obj.display2();
    }
}
```

Implementing method display1 in child class Three
Implementing method display2 in child class Three

Note: Students can write any examples of their wish

Packages

- ❖ A java package is a group of similar types of classes, interfaces and sub-packages.
- ❖ Keyword is **package**
- ❖ Package in java can be categorized in two form, built-in package and user-defined package.
- ❖ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- ❖ Advantages:
 - Java package is used to categorize the classes and interfaces so that they can be easily maintained.
 - Java package provides access protection.
 - Java package removes naming collision.

❖ Here, we will have the detailed learning of creating and using user-defined packages.

Simple Package example:

```
package mypack ;                                //keyword package followed by name of the package

class One
{
    public static void main(String s[ ])
    {
        System.out.println("package is created");
    }
}
```

Let us discuss how to access package from another package (Access Protection)

There are 3 ways to access the package from outside the package.

1. **import package.*;**
2. **import package.classname;**
3. **fully qualified name.**

import package.*

If we use **package.*** then all the classes and interfaces of this package will be accessible.

The import keyword is used to make the classes and interface of another package accessible to the current package.

```
package mypack1;                                //save One.java in the package mypack1

public class One
{
    public void display()
    {
        System.out.println("Hello Kiran from mypack1");
    }
}
```

```
package mypack2;                                //save Two.java in the package mypack2

import mypack1.*;                               //importing mypack1 all classes, interfaces ect.,

public class Two
{
    public static void main(String[] args)
    {
        One obj=new One();
        obj.display();
    }
}
```

Hello Kiran from mypack1

import package.classname

If you import **package.classname** then only declared class of this package will be accessible.

```
package mypack1;                                //save One.java in the package mypack1

public class One
{
    public void display()
    {
        System.out.println("Hello Kiran from mypack1");
    }
}
```

```
package mypack2;                                //save Two.java in the package mypack2

import mypack1.One;                             //importing mypack1 class One Only

public class Two
{
    public static void main(String[] args)
    {
        One obj=new One();
        obj.display();
    }
}
```

```
Hello Kiran from mypack1
```

Fully Qualified name

If we use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But we need to use fully qualified name every time when we are accessing the class or interface. It is generally used when two packages have same class name.

```
package mypack1;                                //save One.java in the package mypack1

public class One
{
    public void display()
    {
        System.out.println("Hello Kiran from mypack1");
    }
}
```

```
package mypack2;                                //save One.java in the package mypack2

public class One
{
    public static void main(String[] args)
    {
        mypack1.One obj= new mypack1.One();    //using name of the package followed by class name
        obj.display();
    }
}
```

```
Hello Kiran from mypack1
```

MODULE 4

EXCEPTIONS AND
APPLETS

- ❖ Exception handling fundamentals
- ❖ Exception types
- ❖ uncaught exceptions
- ❖ using try and catch
- ❖ using multiple catch clauses
- ❖ nested try statements
- ❖ throw, throws, finally
- ❖ Applets

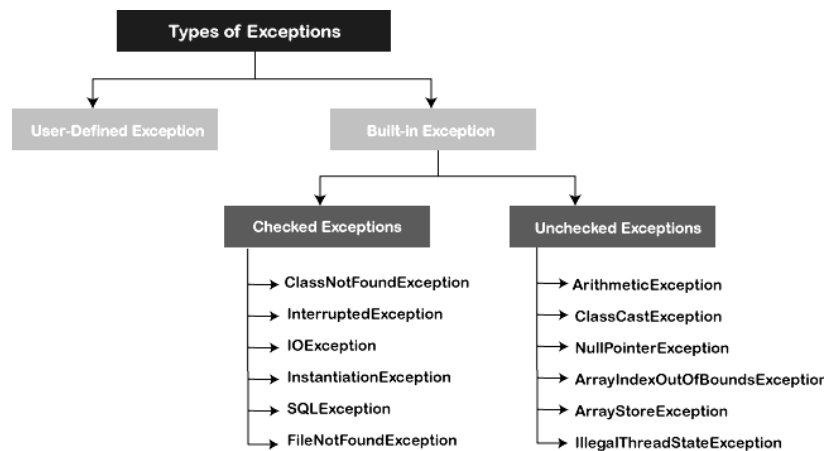
Exception Handling

- ❖ “Exception is a class responsible for abnormal termination of the program whenever run time errors occurred in a program”.
- ❖ A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- ❖ When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- ❖ Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
 - Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Our 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 absolutely must be executed after a try block completes 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 forExceptionType1
}
catch (ExceptionType2 exOb)
{
    // exception handler forExceptionType2
}
// ...
finally
{
    // block of code to be executed after try block ends
}
```

Exception Types



Exceptions can be categorized into two ways:

1. Built-in Exceptions
 - Checked Exception
 - Unchecked Exception
2. User-Defined Exceptions

Built-in Exception

- ❖ Exceptions that are already available in Java libraries are referred to as built-in exception.
- ❖ It can be categorized into two broad categories, i.e., checked exceptions and unchecked exception.

Checked exceptions

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

Example:

Sl.No	Exception	Description
1	ClassNotFoundException	Class not found.
2	CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException	Access to a class is denied.
4	InstantiationException	Attempt to create an object of an abstract class or interface.
5	InterruptedException	One thread has been interrupted by another thread.
6	NoSuchFieldException	A requested field does not exist.
7	NoSuchMethodException	A requested method does not exist.
8	FileNotFoundException	A requested file does not exist.
	Etc,...	

Un-Checked exceptions

- ❖ An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- ❖ The unchecked exceptions defined in **java.lang**

Sl.No	Exception	Description
1	ArithmeticException	Arithmetic error, such as divide-by-zero.
2	ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
3	IndexOutOfBoundsException	Some type of index is out-of-bounds.
4	NullPointerException	Invalid use of a null reference.
5	NumberFormatException	Invalid conversion of a string to a numeric format.
6	StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
7	ArrayStoreException	Assignment to an array element of an

		incompatible type.
8	ClassCastException	Invalid cast.
9	EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
10	IllegalArgumentException	Illegal argument used to invoke a method.
11	IllegalStateException	Environment or application is in incorrect state.
12	IllegalThreadStateException	Requested operation not compatible with current thread state.
13	NegativeArraySizeException	Array created with a negative size
14	SecurityException	Attempt to violate security.
15	TypeNotPresentException	Type not found

User Defined Exception

- ❖ In Java, we can write our own exception class by extends the **Exception** class. We can throw our own exception on a particular condition using the **throw** keyword. For creating a user-defined exception, we should have basic knowledge of the **try-catch** block and **throw** keyword.

Example:

```
// A Class that represents user-defined exception
package kiran;
class MyException extends Exception
{
    public MyException( )
    {
        System.out.println("My exception occurs");
    }
}

// A Class that uses above MyException
package kiran;
public class First
{
    // Driver Program
    public static void main(String args[])
    {
        try {
            // Throw an object of user defined exception
            throw new MyException( ) ;
        }
        catch (MyException ex) {
            System.out.println(ex);
        }
    }
}
```

Output:

My exception occurs

Uncaught Exceptions

- ❖ In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints an exception message with the help of uncaught exception handler.
- ❖ The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.
- ❖ Java programming language has a very strong exception handling mechanism. It allows us to handle the exception use the keywords like try, catch, finally, throw, and throws.
- ❖ When an uncaught exception occurs, the JVM calls a special private method known **dispatchUncaughtException()**, on the Thread class in which the exception occurs and terminates the thread.
- ❖ The **Division by zero exception** is one of the **example** for uncaught exceptions. Look at the following code.

```
import java.util.Scanner;

public class First
{
    public static void main(String[] args)
    {
        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a / b;
        System.out.println(a + "/" + b + " = " + c);
    }
}
```

Output:

```
Enter the a and b values:
5
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at jk.First.main(First.java:13)
```

In the above example code, we are not used try and catch blocks, but when the value of b is zero the division by zero exception occurs and it caught by the default exception handler.

Using try, catch - Exception handling keywords

- ❖ Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.
- ❖ To handle a run-time error, simply enclose the code that we want to monitor inside a **try** block. Immediately following the try block, includes a **catch** clause that specifies the exception type that you wish to catch.

Example:

```
package jk;

import java.util.Scanner;

public class First
{
    public static void main(String[] args)
    {
        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();

        try
        {
            int c = a / b;
            System.out.println(a + "/" + b + " = " + c);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

Output 1: (Exception handled by using try and catch)

```
Enter the a and b values:
5
0
java.lang.ArithmeticException: / by zero
```

Output 2: (No Error)

```
Enter the a and b values:
10
2
10/2 = 5
```

Multiple catch Clauses

- ❖ In some cases, more than one exception could be raised by a single piece of code.
- ❖ To handle this type of situation, you can 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.

Example:

Let us consider an example contain array of 3 elements and we trying to print 5th index element which is not present and this error match with the second catch clause that print exception error.

```
package jk;

public class First
{
    public static void main(String[] args)
    {
        int arr[ ]= {10,20,30};
        try
        {
            System.out.println( arr[5] );           //error
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
```

Nested try statements

- ❖ The **try** statement can be nested. That is, a **try** statement 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, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the **Java run-time system(JVM)** will handle the exception.

Example:

```
package jk;

import java.util.Scanner;

public class First
{
    public static void main(String[] args)
    {
        int arr[] = {10,20,30};
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter values of a and b");
        int a = sc.nextInt();
        int b = sc.nextInt();
        try
        {
            int c = a/b;
            System.out.println(c);
            try
            {
                System.out.println(arr[5]);
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println(e);
            }
        }
        catch (ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

Output: (Error in index)

```
Enter values of a and b
6
2
3
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
```

Output: (consider b value as zero)

```
Enter values of a and b
5
0
java.lang.ArithmeticException: / by zero
```

throw , throws , finally - Exception handling keywords

throw keyword

- ❖ So far, we have only been catching exceptions that are thrown by the Java run-time system.
- ❖ However, it is possible for our program to **throw an exception explicitly**, using the **throw** statement.
- ❖ The general form of **throw** is shown here:

```
throw new exception_class( ) ;           //we can pass parameter also
```

Where the throw Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Example: Below program shows using throw keyword in User Defined Exception similarly we can also use throw keyword in checked and unchecked Exceptions also.

```
// A Class that represents user-defined exception
package kiran;
class MyException extends Exception
{
    public MyException( )
    {
        System.out.println("My exception occurs");
    }
}

// A Class that uses above MyException
package kiran;
public class First
{
    // Driver Program
    public static void main(String args[])
    {
        try {
            // Throw an object of user defined exception
            throw new MyException( ) ;
        }
        catch (MyException ex) {
            System.out.println(ex);
        }
    }
}
```

Output:
My exception occurs

throws keyword

- ❖ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. we do this by including a **throws** clause in the method's declaration.
- ❖ **throws** keyword should use with method signature.
- ❖ If we use throws keyword then no need of try and catch blocks.
- ❖ A throws clause lists the types of exceptions that a method might throw.
- ❖ Syntax:

The 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.

Example:

```
package jk ;

public class First
{
    static void display ( int age ) throws ArithmeticException
    {
        if (age < 18)
        {
            throw new ArithmeticException("Access denied - You must be at least 18 years old.");
        }
        else
        {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main( String[] args )
    {
        display(15);           // Set age to 15 (which is below 18...)
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
    at jk.First.display(First.java:10)
    at jk.First.main(First.java:20)
```

finally keyword

- ❖ **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.
- ❖ If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- ❖ The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- ❖ Syntax:

```
finally { //Statements; }           // after try block
```

Example:

```
package jk;

import java.util.Scanner;

public class First
{
    public static void main(String[] args)
    {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter values of a and b");
        int a=sc.nextInt();
        int b=sc.nextInt();
        try
        {
            int c = a / b;
            System.out.println( c );
        }
        catch(ArithmeticException e)
        {
            System.out.println( e );
        }
        finally
        {
            System.out.println("finally block executed if exception occure or not");
        }
    }
}
```

Output:

```
Enter values of a and b
5
0
java.lang.ArithmeticException: / by zero
finally block executed if exception occure or not
```

Applets

- ❖ Still now we discussed console based applications but to create window based applications we will use applets.
- ❖ Applet is a window based application works with java and html programs (files).
- ❖ Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.
- ❖ Some Important points –
 - An applet is a Java class that extends the `java.applet.Applet` class.
 - Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called `applet viewer`.
 - In general, execution of an applet does not begin at `main()` method.
 - Output of an applet window is not performed by `System.out.println()`. Rather it is handled with various AWT methods, such as `drawString()`.

Steps to Write and execute applet program

1. Open new notepad write java source code and save as **First.java**
2. Open command prompt for the same path where our java and html files present.
3. Compile java program to create class file by using command, **javac First.java**
4. Open new notepad in same folder of First.java , write html code that contains applet tag with `code="First.class"` and save as **test.html**
5. To create applet window type the command in the cmd prompt , **appletviewer test.html**
6. Finally our applet window is created and opened.

Simple Example:

```
//First.java
import java.awt.*;
import java.applet.*;

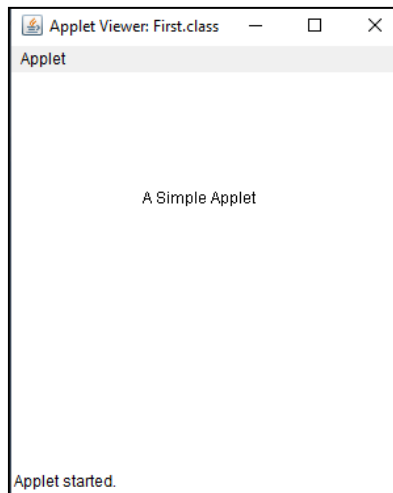
public class First extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("A Simple Applet", 100, 100);
    }
}
```

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

Output:

```
C:\Windows\System32\cmd.exe - appletviewer test.html
Microsoft Windows [Version 10.0.19045.2364]
(c) Microsoft Corporation. All rights reserved.

F:\jk>javac First.java
F:\jk>appletviewer test.html
```



Explanation of the java program:

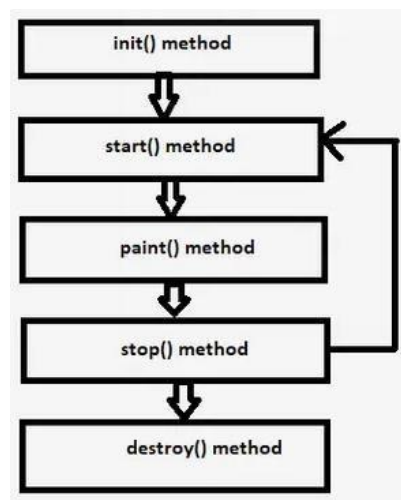
- ❖ This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes that provides GUI(Graphical user Interface) by using **paint ()** method. The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet.
- ❖ The next line in the program declares the class First. This class must be declared as public, because it will be accessed by code that is outside the program.
- ❖ Inside First, **paint()** is declared. This method is defined by the AWT. **paint()** is called each time that the applet must redisplay its output.
- ❖ **paint ()** method is used for several purposes like to draw string , to draw rectangle , to draw Line, show status etc.,.
- ❖ Inside **paint()** is a call to **drawString()**, which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location on the screen.
- ❖ Notice that the applet does not have a **main()** method. Unlike Java programs, applets **do not** begin execution at **main()**.

Explanation of the html program:

- ❖ Html program always starts with the tag **<html>**
- ❖ Next line we starts with the tag
<applet code="First.class" width=300 height=300>
means, in the html file we connect to our java compiled file that is class file with applet window width and height values.
- ❖ Next line we will close applet tag by **</applet>**
- ❖ Next line we will close html tag by **</html>**

Applet Life Cycle

- ❖ Applet life cycle contains total 5 stages or methods –
 1. initialization – public void init()
 2. start – public void start()
 3. paint – public void paint(graphics g)
 4. stop – public void stop()
 5. destroy – public void destroy()
- ❖ The **java.applet.Applet** class has **4** life cycle methods – init(), start(), stop() and destroy()
java.awt.Component class provides **1** life cycle method – paint().



- ❖ It is important to understand the order in which the various methods shown in the above image are called.
- ❖ When an applet begins, the following methods are called, in this sequence:
 - init()
 - start()
 - paint()
- ❖ When an applet is terminated, the following sequence of method calls takes place:
 - stop()

➤ `destroy()`

Let's look more closely at these methods.

1. **init()** : The **init()** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
2. **start()** : The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. Note that **init()** is called once i.e. when the first time an applet is loaded whereas **start()** is called each time an applet's HTML document is displayed onscreen.
3. **paint()** : The **paint()** method is called each time an AWT-based applet's output must be redrawn. The **paint()** method has one parameter of type **Graphics**. And create object **g** for the graphics class.
4. **stop()** : The **stop()** method is called when an applet window is **minimized** that is suspended for a while.
5. **destroy()** : The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop()** method is always called before **destroy()**.

Example program that shows applet life cycle

```
import java.applet.*;           // save as First.java
import java.awt.*;

public class First extends Applet
{
    int length, breadth;

    public void init()           // Used to initialize
    {
        System.out.println("Welcome to Init method "); //observe in cmd prompt while run html
        setBackground(Color.green);
        setForeground(Color.black);
        length=10;
        breadth=30;
    }

    public void start()          //Executed when applet started
    {
        System.out.println("Welcome to Start method "); //observe in cmd prompt while maximize applet window
    }

    public void paint(Graphics g) //GUI shown in applet window
    {
        System.out.println("Welcome to Paint method "); //observe in cmd prompt while maximize applet window

        int area=length*breadth;
        g.drawString("Area of Rectangle is "+area ,100,100);
    }

    public void stop()           //suspend current task
    {
        System.out.println("Welcome to Stop method "); //observe in cmd prompt while minimize applet window
    }

    public void destroy()        //close or destroy the applet window
    {
        System.out.println("Welcome to destroy method "); //observe in cmd prompt while close applet window
    }
}
```

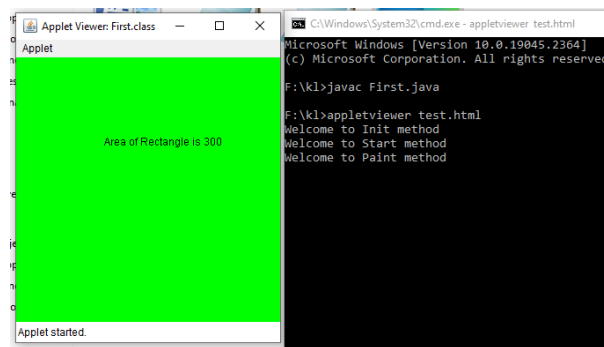
```
}  
}  
<html>  
<applet code="First.class" width=300 height=300>  
</applet>  
</html>
```

//save as test.html

Output:

```
C:\Windows\System32\cmd.exe  
Microsoft Windows [Version 10.0.19045.2364]  
(c) Microsoft Corporation. All rights reserved.  
  
F:\kl>javac First.java  
  
F:\kl>appletviewer test.html
```

Now Applet window is created and shown to you (same time observe cmd prompt init, start paint methods are invoked)



Minimize the applet and observe cmd prompt , stop method is invoked.

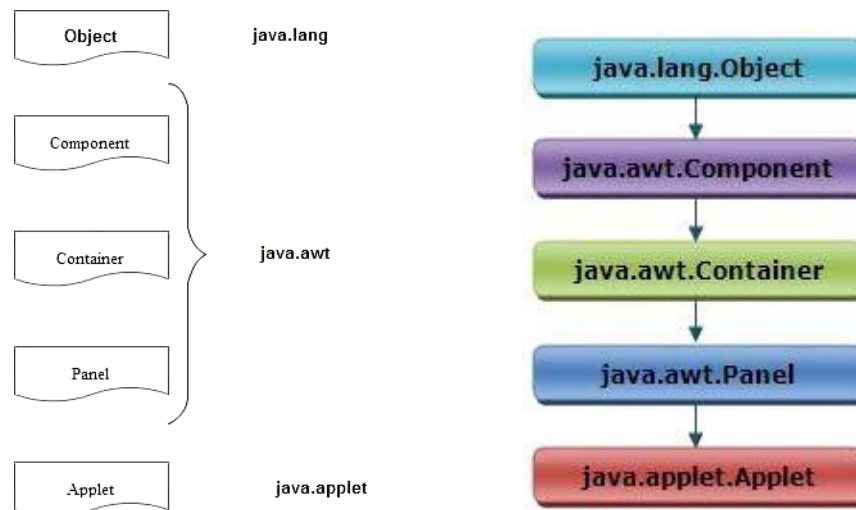
```
C:\Windows\System32\cmd.exe - appletviewer test.html  
Microsoft Windows [Version 10.0.19045.2364]  
(c) Microsoft Corporation. All rights reserved.  
  
F:\kl>javac First.java  
  
F:\kl>appletviewer test.html  
Welcome to Init method  
Welcome to Start method  
Welcome to Paint method  
Welcome to Stop method
```

Close the applet window then stop and destroy method will be invoked.

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2364]
(c) Microsoft Corporation. All rights reserved.

F:\k1>javac First.java
F:\k1>appletviewer test.html
Welcome to Init method
Welcome to Start method
Welcome to Paint method
Welcome to Stop method
Welcome to Start method
Welcome to Paint method
Welcome to Stop method
Welcome to destroy method
F:\k1>
```

Applet Heirarchy



- ❖ The AWT allows us to use various graphical components. When we start writing any applet program we essentially import two packages namely – **java.awt** and **java.applet**.
- ❖ The **java.applet** package contains a class **Applet** which uses various interfaces such as AppletContext, AppletStub and AudioClip.
- ❖ The applet class is a sub class of **Panel** class belonging to **java.awt** package.
- ❖ To create a user friendly graphical interface we need to place various components on GUI window. There is a **Component** class from **java.awt** package which derives several classes for components. These classes include Check box, Choice, List, buttons and so on.
- ❖ The Component class in java.awt is an abstract class.
- ❖ Component class is an extension of object class that is **java.lang.Object**.

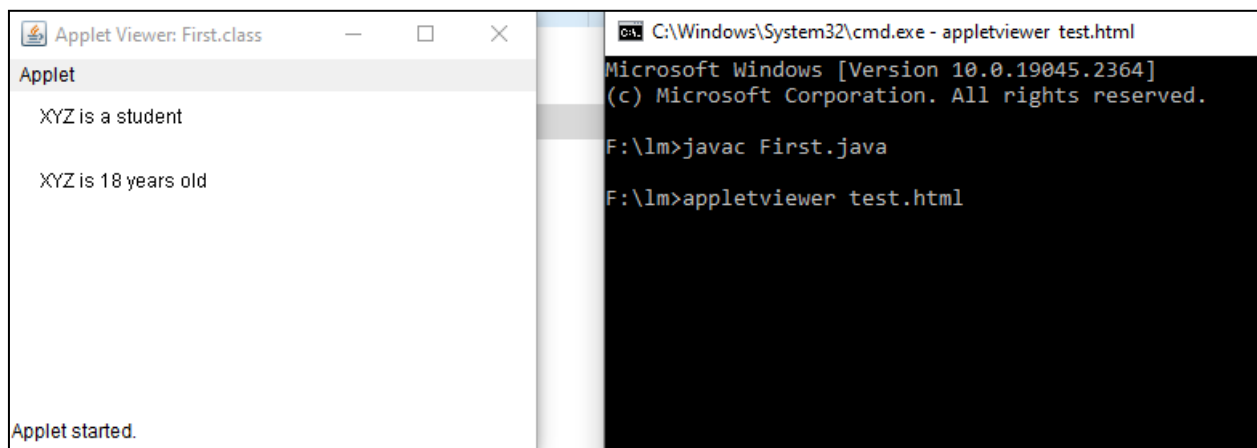
Passing parameters and getting parameters in applet

```
import java.applet.*;                                //save as First.java
import java.awt.*;

public class First extends Applet
{
    String n , a;
    public void init( )
    {
        n = getParameter("Name");
        a = getParameter("Age");
    }
    public void paint(Graphics g)
    {
        g.drawString(n + " is a student" , 20 , 20 );
        g.drawString(n + " is " + a + "years old" , 20 , 60 );
    }
}
```

```
<html>                                                //save as test.html
<body>
    <applet code="First.class" height=200 width=200>
        <param name="Name" value="XYZ">
        <param name="Age" value="18">
    </applet>
</body>
</html>                                              //param tag should be used to pass values
```

Output:



MODULE 5

EVENT HANDLING
AND MULTI-
THREADED
PROGRAMMING

- ❖ **Event Handling and Multi-Threaded Programming:**
- ❖ Two event handling mechanisms, the delegation event model,
- ❖ Event classes,
- ❖ Sources of events,
- ❖ Event listener interfaces,
- ❖ Using the delegation event model,
- ❖ Adapter classes, Inner classes.
- ❖ **Multi-Threaded Programming:**
- ❖ What are threads?
- ❖ How to make the classes threadable,
- ❖ Extending threads,
- ❖ Implementing runnable,
- ❖ Synchronization

Multi-Threaded Programming

- Multithreading in Java is a process of executing multiple threads simultaneously for maximum utilization of CPU.
- Each part of such a program is called a thread.
- Multithreading is used to achieve multitasking.
- There are two distinct types of multitasking:
 - Process based multitasking.
 - Thread-based multitasking.
- Process-based multitasking (Multiprocessing):
 - Process-based multitasking is to run two or more programs concurrently
 - Each process has an address in memory. In other words, each process allocates a separate memory area.
- Thread-based multitasking (Multithreading):
 - In thread-based multitasking, a single program can perform two or more tasks simultaneously.
 - Threads share the same address space.

What are threads?

- Java provides built-in support for multithreaded programming.
- 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.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Multitasking threads require less overhead than multitasking processes.

Thread Model

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- **New:**
 - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
- **Runnable**
 - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
- **Blocked**
 - This is the state when the thread is still alive, but is currently not eligible to run.

➤ **WAITING:**

A thread enters this state if it waits to be notified by another thread, which is the result of calling `Object.wait()` or `Thread.join()`. The thread also enters waiting state if it waits for a Lock or Condition in the `java.util.concurrent` package. When another thread calls `Object.notify()/notifyAll()` or `Condition's signal()/signalAll()`, the thread comes back to the runnable state.

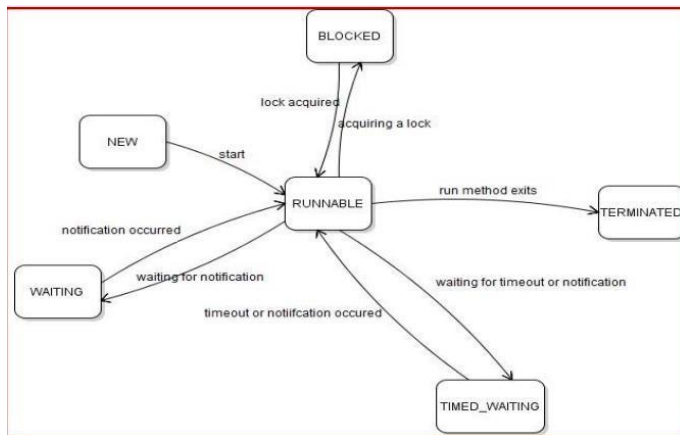
➤ **Timed-Waiting**

A thread enters this state if a method with timeout parameter is called: `sleep()`, `wait()`, `join()`, `Lock.tryLock()` and `Condition.await()`. The thread exits this state if the timeout expires or the appropriate notification has been received.

➤ **Terminated**

A thread enters terminated state when it has completed execution. The thread terminates for one of two reasons:

- The `run()` method exits normally.
- The `run()` method exits abruptly due to an uncaught exception occurs.



How to make the classes threadable

Java defines two ways in which this can be accomplished:

- By implementing the Runnable interface.
- By extending the Thread class

Implementing the Runnable Interface

To create a thread is to create a class that implements the Runnable interface.

- Runnable abstracts a unit of executable code.
- Construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run()
- **Syntax;**
public void run()
- **run()** method introduces a concurrent thread into your program. This thread will end when run() method terminates.
- we must specify the code that our thread will execute inside run() method.
- **run()** method can call other methods, can use other classes and declare variables just like any other normal method.
- **Example:**

```
class first implements Runnable
{
    public void run( )
    {
        System.out.println("Thread started ..");
    }
}
public class Multithreading
{
    public static void main(String [ ] s)
    {
        first obj=new first();           //or – Thread t1=new Thread ( new first ( ) );
        Thread t1 = new Thread ( obj );
        t1.start( ) ;
    }
}
```

Thread started ..";

- To call the run() method, start() method is used.
- On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Extending Thread class

- Create a thread by a new class that **extends Thread class** and create an instance of that class.
- The extending class must **override run()** method which is the entry point of new thread.
- Example:

```
class first extends Thread
{
    public void run( )
    {
        System.out.println("Thread started ..");
    }
}
public class Multithreading
{
    public static void main(String [ ] s)
    {
        first t1=new first( );
        t1.start( );
    }
}
```

Thread started ..");

Example program that demonstrates Multi-threading programming which is the combination of extends Thread and Implements runnable (Lab Program)

```
import java.util.Scanner;
public class Multithreading
{
    public static void main(String[] args)
    {
        first t1=new first();
        t1.start();
    }
}

class first extends Thread
{
    public void run()
    {
        int num=0;
        Random r=new Random();
        try
        {
            for(int i=0;i<=2;i++)
            {
                num=r.nextInt(10);
                System.out.println("first thread generated num is "+num);

                Thread t2 = new Thread(new second(num));
                t2.start();
                Thread.sleep(1000);
                Thread t3 = new Thread(new third(num));
                t3.start();
                Thread.sleep(1000);
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}

class second implements Runnable
{
    int x;
    public second(int x)
    {
        this.x=x;
    }
    public void run()
    {
        System.out.println("Second thread: Square of 2 num is "+(x*x));
    }
}
```

Write a Java program that implements a multi-thread application that has three threads. First thread generates a random integer forevery 1 second; second thread computes the square of the number and prints; third thread will print the value of cube of the number.

```
class third implements Runnable
{
    public int x;
    public third(int x)
    {
        this.x=x;
    }
    public void run( )
    {
        System.out.println("Third thread: Cube of num is" +(x*x*x));
    }
}
```

Output:

first thread generated num is =2
Second thread: Square of 2 num is4
Third thread: Cube of num is8

first thread generated num is =3
Second thread: Square of 2 num is9
Third thread: Cube of num is27

first thread generated num is =8
Second thread: Square of 2 num is64
Third thread: Cube of num is512

Synchronization

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- Example:

```
package jk;

public class Sample
{
    synchronized void print(int n)           //synchronized method
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println(n+i);
            try{ Thread.sleep(500); }
            catch(Exception e){ System.out.println(e); }
        }
    }
}
```

```
package jk;

public class Thread1 extends Thread
{
    Sample s;
    Thread1(Sample s)
    {
        this.s=s;
    }
    public void run()
    {
        s.print(10);
    }
}
```

```
package jk;

public class Thread2 extends Thread
{
    Sample s;
    Thread2(Sample s)
    {
        this.s=s;
    }
    public void run()
    {
        s.print(50);
    }
}
```

```
package jk;

public class Test
{
    public static void main(String[] args)
    {
        Sample obj=new Sample();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);

        t1.start();
        t2.start();
    }
}
```

Output:

Without Thread Synchronization (Random output with both threads - we will get different outputs with different runs)

Run1	Run2	Run3
11	51	51
51	11	11
52	52	52
12	12	12
53	13	53
13	53	13
54	54	14
14	14	54
55	55	15
15	15	55

With Thread Synchronization (Always gets same output that is Thread1 followed by Thread2)

11
12
13
14
15
51
52
53
54
55

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can also preempt a lower-priority one.

To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`.

This is its general form:

```
final void setPriority(int level)
```

Here, `level` specifies the new priority setting for the calling thread. The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`.

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non preemptive platform. One thread is set two levels above the normal priority, as defined by `Thread.NORM_PRIORITY`, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
    long click = 0;
    Thread t;
    private volatile boolean running = true;
    public clicker(int p) {
        t = new Thread(this);
        t.setPriority(p);
    }
    public void run() {
        while (running) {
            click++;
        }
    }
    public void stop() {
        running = false;
    }
    public void start() {
        t.start();
    }
}
class HiLoPri {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
        lo.start();
        hi.start();
    }
}
```

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
    hi.t.join();
    lo.t.join();
} catch (InterruptedException e) {
    System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}
```

The output of this program, shown as follows when run under Windows, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got the majority of the CPU time.

Low-priority thread: 4408112

High-priority thread: 589626904

Event Handling

What is an Event?

- ❖ Change in the state of an object is known as event i.e. event describes the change in state of source.
- ❖ Events are generated as result of user interaction with the graphical user interface components.
- ❖ For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

What is Event Handling?

- ❖ Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- ❖ These mechanisms have the code which is known as event handler that is executed when an event occurs.
- ❖ Java Uses the Delegation Event Model to handle the events.
- ❖ This model defines the standard mechanism to generate and handle the events

Two event handling mechanisms;

- ❖ The way in which events are handled changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1.
- ❖ The 1.0 method of event handling is still supported, but it is not recommended for new programs.
- ❖ Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by Programs.

The Delegation Event Model

- ❖ The delegation event model is defining standard and consistent mechanisms to generate and process events.
- ❖ A source generates an event and sends it to one or more listeners.
- ❖ The listener simply waits until it receives an event.
- ❖ Once an event is received, the listener processes the event and then returns.
- ❖ In the delegation event model, listeners must register with a source in order to receive an event notification.
- ❖ The notifications are sent only to listeners that want to receive them.
- ❖ **Events:**
 - An event is an object that describes a state change in a source.
 - Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
 - Events may also occur that are not directly caused by interactions with a user interface.
 - *Example:*
An event may be generated when

- ✓ A timer expires
- ✓ A counter exceeds a value
- ✓ A software or hardware failure occurs, or an operation is completed

❖ Event Sources

- A source is an object that generates an event, occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- Syntax:

public void addTypeListener(TypeListener el)

- ✓ **Type** is the name of the event
- ✓ **el** is a reference to the event listener.

Example

- ❖ The method that registers a keyboard event listener is called **addKeyListener()**
- ❖ The method that registers a mouse motion listener is called **addMouseMotionListener()**
- ❖ When an event occurs, all registered listeners are notified and receive a copy of the event object is known as **multicasting** the event.

Some sources may allow only one listener to register.

Syntax:

public void addTypeListener(TypeListener el)

throws java.util.TooManyListenersException

- ✓ **Type** is the name of the event
- ✓ **el** is a reference to the event listener.

When such an event occurs, the registered listener is notified. This is known as **unicasting** the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event.

Syntax:

public void removeTypeListener(TypeListener el)

- ✓ **Type** is the name of the event
- ✓ **el** is a reference to the event listener.

Example:

To remove a keyboard listener, you would call **removeKeyListener()**.

Sources of Events

- ❖ Some of the user interface components that can generate the events.
- ❖ In addition to these graphical user interface elements, any class derived from *Component*, such as *Applet*, can generate events.
- ❖ *Example*
 - You can receive key and mouse events from an applet. (You may also build your own components that generate events.)

Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, iconified, opened, or quit.

Event Listener Interfaces

- ❖ The delegation event model has two parts: *sources* and *listeners*.
- ❖ Listeners are created by implementing one or more of the interfaces defined by the *java.awt.event* package.
- ❖ When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

<i>Interface</i>	<i>Description</i>
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.

MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

❖ The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent ae)

❖ The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent ce)

void componentMoved(ComponentEvent ce)

void componentShown(ComponentEvent ce)

void componentHidden(ComponentEvent ce)

❖ The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent ce)

void componentRemoved(ContainerEvent ce)

❖ The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked. When the mouse enters a component, the **mouseEntered()** method

is called. When it leaves, `mouseExited()` is called. The `mousePressed()` and `mouseReleased()` methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Explain KeyEvent and MouseEvent class.

KeyEvent :

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember not all keypresses result in characters. For example, pressing shift does not generate a character.

There are many other integer constants that are defined by **KeyEvent**.

For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT, **VK_CANCEL**, **VK_CONTROL**, **VK_DOWN**, **VK_ENTER**, **VK_ESCAPE**, **VK_LEFT**
VK_PAGE_DOWN, **VK_PAGE_UP**, **VK_RIGHT**, **VK_SHIFT**, **VK_UP**

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt. **KeyEvent** is a subclass of **InputEvent**.

Syntax:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

Here, **src** is a reference to the component that generated the event.

The type of the event is specified by **type**.

The system time at which the key was pressed is passed in **when**.

The **modifiers** argument indicates which modifiers were pressed when this key event occurred.

The virtual key **code**, such as **VK_UP**, **VK_A**, and so forth, is passed in **code**.

The character equivalent (if one exists) is passed in **ch**.

The MouseEvent Class :

There are eight types of mouse events.

The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED : The user clicked the mouse.

MOUSE_DRAGGED : The user dragged the mouse.

MOUSE_ENTERED : The mouse entered a component.

MOUSE_EXITED : The mouse exited from a component.

MOUSE_MOVED : The mouse moved.

MOUSE_PRESSED : The mouse was pressed.

MOUSE_RELEASED : The mouse was released.

MOUSE_WHEEL : The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**.

Syntax:

MouseEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup)

Here, **src** is a reference to the component that generated the event.

The type of the event is specified by **type**.

The system time at which the mouse event occurred is passed in **when**.

The **modifiers** argument indicates which modifiers were pressed when a mouse event occurred.

The coordinates of the mouse are passed in **x** and **y**.

The click count is passed in **clicks**.

The **triggersPopup** flag indicates if this event causes a pop-up menu to appear on this platform.

Develop and demonstrate how delegation model used to handle mouse events in java.

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Handling Mouse Events

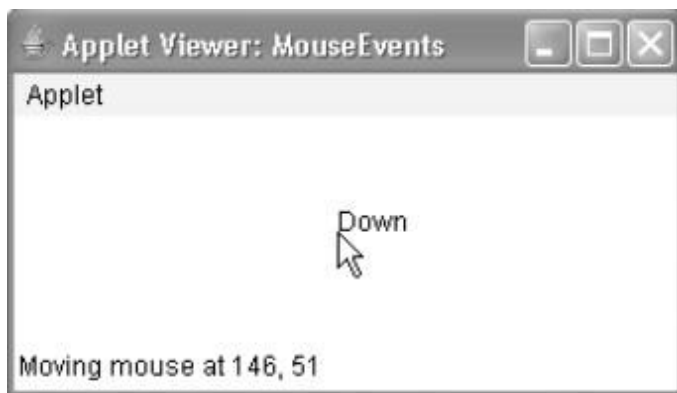
To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces.

The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.

```
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; // coordinates of mouse
    public void init ( ) {
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
        repaint( );
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered.";
        repaint ( );
    }
    // Handle mouse exited.
    public void mouseExited(MouseEvent me) {
        // save coordinates
        mouseX = 0;
    }
}
```

```
mouseY = 10;
msg = "Mouse exited.";
repaint ( );
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint( );
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
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 at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```

Sample output from this program is shown here:



Develop and demonstrate how delegation model used to handle key events in java.

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListenerinterface**.

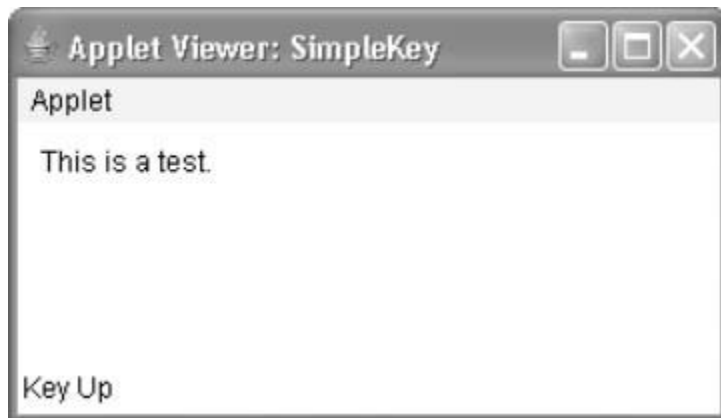
Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased()** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped()** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the keypress and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed()** handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

// Demonstrate the key event handlers.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

Output:



Adapter Classes

Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged()** and **mouseMoved()**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged()**. The empty implementation of **mouseMoved()** would handle the mouse motion events for you.

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init()** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events.

Both of the constructors take a reference to the applet as an argument. **MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked()** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged()** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}
class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked");
    }
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
    }
}
```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods.