**Introduction Of Java**

* Java technology is both a programming language and a platform.
* The Java programming language is a high-level language.
* James Gosling - founder of java.
* It was first released by Sun Microsystems in 1995 and later acquired by Oracle Corporation.
* In the Java programming language, all source code is first written in plain text files ending with the java extension.
* Those source files are then compiled into .class files by the javac compiler.
* A class file does not contain code that is native to your processor; it instead contains bytecodes the machine language of the Java Virtual Machine (Java VM).
* The java launcher tool then runs your application with an instance of the Java Virtual Machine.



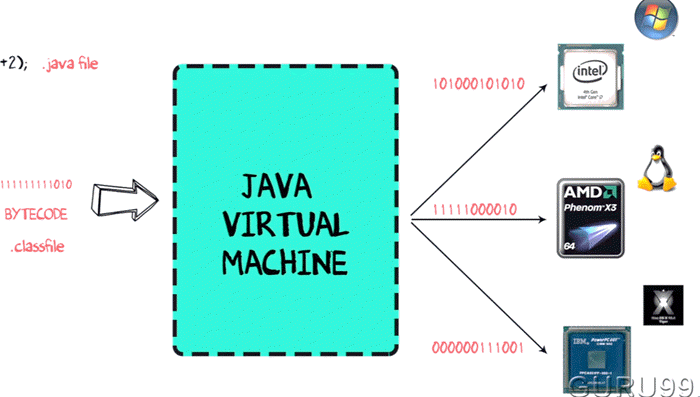
**Features Of Java**

There is given many features of java. They are also known as java buzzwords.   
**1.ObjectOriented:-**  
Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:-  
i)Object  
ii)Class  
ii)Inheritance  
iv)Polymorphism  
v)Abstraction  
vi)Encapsulation  
  
**2.Platform-independent:-**Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions ofUNIX.  
  


**3.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.  
Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.   
  
**4.Secured:-**Java is secured because:  
i) No explicit pointer  
ii) Programs run inside virtual machine sandbox.   
  
**5.Robust:-**Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points make java robust.   
  
**6.Architectural-neutral:-**There is no implementation dependent features e.g. size of primitive types is set.   
  
**7.Portable: -**We may carry the java bytecode to any platform.  
  
**8.Dynamic: -**Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.   
  
**9.Interpreted:-**As described earlier, 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.   
  
**10. High-performance:-**Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)   
  
**11.Multi-threaded:-**A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.   
  
**12.Distributed:-**We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

**How Java Virtual Machine works?**

By using **Java Virtual Machine**, this problem can be solved. But how it works on different processors and O.S. Let's understand this process step by step.

[](https://www.guru99.com/images/ccna/061516_1256_WhatisJava9.png)

**Step 1:-**

The code to display addition of two numbers is System.out.println(1+2), and saved as .java file.

**Step 2:-**

Using the java compiler the code is converted into an intermediate code called the **bytecode.** The output is a **.class file.**

**Step 3:-**

This code is not understood by any platform, but only a virtual platform called the **Java Virtual Machine.**

**Step 4:-**

This Virtual Machine resides in the RAM of your operating system. When the Virtual Machine is fed with this bytecode, it identifies the platform it is working on and converts the bytecode into the native machine code.

**Variable**

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

### Types of Variables

There are three types of variables in java:

* local variable
* instance variable
* static variable

#### 1) 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.

#### 2) Instance Variable

A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.

It is called instance variable because its value is instance specific and is not shared among instances.

#### 3) Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

### *Example*:

class A{

int data=50;//instance variable

static int m=100;//static variable

void method(){

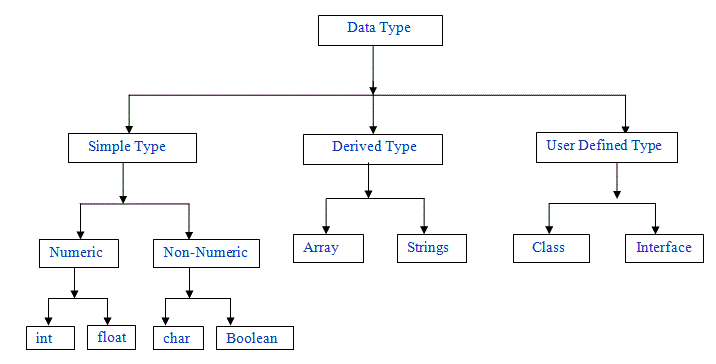
int n=90;//local variable

}

}//end of class

Java Data Types

The main purpose of Data Types in java is to determine what kind of value we can stored in to the variable.  
Ex: int x;  
like, int=12;



|  |  |  |
| --- | --- | --- |
| **Data Types** | **Memory Size** | **Minimum and Maximum Values** |
| Byte | 1 byte | -128 to +127 |
| Short | 2 bytes | -32768 to + 32767 |
| Int | 4 bytes | -2147483648 to + 2147483647 |
| Long | 8 bytes | -9223372036854775808 to +9223372036854775807 |
| Float | 4 bytes | -3.4e38 to -.4e-45 for negative values and 1.4e-45 to 3.4e+038 for positive values. |
| Double | 8 bytes | -1.8e08 to -4.9e-324 for negative values and 4.9e-324 to 1.8e+308 for positive values. |
| Char | 2 bytes | 0 to 65535 |
| boolean |  | true or false |

Java Basic Tools

|  |  |
| --- | --- |
| **Tool Name** | **Brief Description** |
| Javac | The compiler for the Java programming language. |
| Java | The launcher for Java applications. |
| Javadoc | API documentation generator. |
| Appletviewer | Run and debug applets without a web browser. |
| Jar | Create and manage Java Archive (JAR) files. |
| Jdb | The Java Debugger. |
| Javah | C header and stub generator. Used to write native methods. |
| Javap | Class file disassembler |
| Extcheck | Utility to detect Jar conflicts. |

Java Class Path

Java Class Path is required for using tools such as javac, java etc. If you are saving the java file in jdk/bin folder, path is not required.But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK.

There are two ways to set java class path of JDK:

1. Temporary

2. Permanent

***1. Temporary:-*** Temporary java class path set of JDK in windows :-

You need to follow the some steps are:-

i) Open Command Prompt

ii) Copy the path of bin folder

iii) Write in command prompt: set path=copiedpath

Ex:-

set path=C:\Program Files\Java\jdk1.6.0\_23\bin

### *2. Permanent*:- Permanent java class path set of JDK in windows :-

You need to follow the some steps are:-

i) Go to My Computer Properties

ii) Advanced System Settings

iii) Advanced Tab

iv) Environment Variables

v) New Tab of User Variable

vi) Write path in variable name

vii) Write path of bin folder in variable value

viii) Ok

ix) Ok

x) Ok

Java Simple Program

public class DemoProgram

{

//Start of Main Method

public static void main(string args[])

{

System.out.println(" Simple demo program");

} //End of Main Method

} //End of DemoProgram Class

**Output:-**Simple demo program

### How to Compile and Run the java program:- i) Save java program with extension .java  ii) Compile java program javac DemoProgram.java iii) Run java program java DemoProgram

Parameters used in First Java Program:-

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

* **class** keyword is used to declare a class in java.
* **public** keyword is an access modifier which represents visibility. It means it is visible to all.
* **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.
* **void** is the return type of the method. It means it doesn't return any value.
* **main** represents the starting point of the program.
* **String[] args** is used for command line argument. We will learn it later.
* **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

Arrays

Array is a collection of similar type of elements that have contagious memory location.

Array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed elements in an array.

Array is index based, first element of the array is stored at 0 index.

**Creation of arrays:**

Arrays are data structures that can store large amount of information of the same data type grouped together and known by a common name. Each member is called an element of the array.

Arrays are capable of storing primitive data types as well as objects. The elements of the array can be accessed by its index value that starts from 0. Once array is declared, its size cannot be altered dynamically.

Arrays can be :-

a) declared and later assigned or

b) initialized.

// declaration of an array

int subject[ ] = new int[ 10 ] ;

// if not assigned, default 0 is assigned for int element

System.out.println( subject[ 1 ] ) ;

// assigning a value to an element

subject[ 1 ] = 45 ;

// now, prints 45

System.out.println( subject[ 1 ] ) ;

**Advantages of Array**

1. Code Optimization: It makes the code optimized, we can retrive or sort the data easily.

2. Random access: We can get any data located at any index position.

**Disadvantage of Array**

Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array

There are two types of array.

1. Single Dimensional Array

2. Multidimensional Array

Example:-Single Dimensional Array

//Start of ArrayDemo class

class ArrayDemo

{

//Start of Main Method

public static void main(String args[])

{

//declaration and instantiation

int array[]=new int[6];

//initialization

array[0]=100;

array[1]=200;

array[2]=300;

array[3]=400;

array[4]=500;

array[5]=600;

//length is the property of array

for(int i=0;i< array.length;i++)

//printing array

System.out.println(array[i]);

}//End of Main Method

}//End of ArrayDemo class

Save this file as ArrayDemo.java

To compile: javac ArrayDemo.java

To execute: java ArrayDemo

Output:-

100

200

300

400

500

600

Example:-Multidimensional Array

//Start of MultiArray class

class MultiArray

{

//Start of Main Method

public static void main(String args[])

{

//declaring and initializing 2D array

int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//Start of outer for loop

for(int i=0;i< 3;i++)

{

//Start of inner for loop

for(int j=0;j< 3;j++)

{

//printing 2D array

System.out.print(arr[i][j]+" ");

}//End of inner for loop

System.out.println();

}//End of outer for loop

}//End of Main Method

}//End of MultiArray class

Save this file as MultiArray.java

To compile: javac MultiArray.java

To execute: java MultiArray

Output:-

1 2 3

2 4 5

4 4 5

Java Constructors

A constructor is a special member method which will be called by the JVM implicitly(automatically) for placing user/programmer defined values instead of placing default values.  
  
Constructors are meant for initializing the object. Constructor is a special type of method that is used to initialize the state of an object.  
  
Constructor is invoked at the time of object creation. It constructs the values i.e. data for the object that is why it is known as constructor.  
  
Constructor is just like the instance method but it does not have any explicit return type.  
  
**Advantages of Constructors:**1. A constructor eliminates placing the default values.  
2. A constructor eliminates callling the normal method implicitly.  
  
RULES/CHARACTERISTICS of a Constructor:1. Constructor name must be same as its class name.  
2. Constructor should not return any value even void also.  
3. Costructors should not be static .  
4. Constructors should not be private.  
5. Constructors will not be inherited at all.  
6. Constructors are called automatically whenever an object is cereating.  
  
Types of Constructors:  
There are two types of constructors:-  
1. Default constructor (no-argument constructor)  
2. Parameterized constructor  
  
**1. Default constructor (no-argument constructor):-**

A constructor is one which will not take any parameter.  
A constructor that have no parameter is known as default constructor.  
  
**Syntax:-**

class< class name >

{

classname() //default constructor

{

Block of statements;

...................;

...................;

}

..................;

..................;

};

Example:-

//Start Of TestConstructor

class TestConstrucutor

{

int a, b;

TestConstructor()

{

System.out.println(" Default Constructor !!!");

a = 10;

b = 20;

System.out.println("Value of a = " +a);

System.out.println("Value of b = " +b);

}

};

class MainConstructor

{

public static void main(String args[])

{

TestConstructor obj = new TestConstructor();

}

};

**2. Parameterized Constructor:-**

A constructor is one which takes some parameters.  
  
Syntax:-

class< class name >

{

classname(list of parameters) //paramiterized constructor

{

Block of statements;

...................;

...................;

}

..................;

..................;

};

Example:-

//Start Of TestConstructor

class TestConstrucutor

{

int a, b;

TestConstructor(int x, int y)

{

System.out.println(" Parameterized Constructor !!!");

a = x;

b = y;

System.out.println("Value of a = " +a);

System.out.println("Value of b = " +b);

}

};

class MainConstructor

{

public static void main(String args[])

{

TestConstructor obj = new TestConstructor(10,20);

}

};

BufferedReader, InputStreamReader, and System.in:-

System.in refers InputStream object and corresponds to keyboard input or any input source specified by the host environment or user.  
System.out refers OutputStream object and corresponds to display output(monitor) or any output destination specified by the host environment or user.

**HOW DATA IS ACCEPTED FROM KEYBOARD ?**

We need three objects,

* System.in
* InputStreamReader
* BufferedReader

InputStreamReader and BufferedReader are classes in java.io package.

The data is received in the form of bytes from the keyboard by System.in which is an InputStream object.

Then the InputStreamReader reads bytes and decodes them into characters.

Then finally BufferedReader object reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

|  |
| --- |
| InputStreamReader inp = new InputStreamReader(system.in); BufferedReader br = new BufferedReader(inp); |
|  |

**READ() ANDREADLINE() METHODS:**

BufferedReader class contains methods such as read() and readline() which are used to read the data from BufferedReader object.  
read() – reads a single character

br.read() reads a single character from the BufferedReader object ‘br’ but returns its ASCII value which is an integer. so we use typecast to convert an integer to character by using (char) before br.read().

readline() – reads a line of text

|  |
| --- |
| String s = br.readline() |

br.readLine() reads a line of text from the BufferedReader object ‘br’ and returns string. So no need of casting here.

### Storing numeric data from bufferedreader object:

All numeric datas such as integer, float and double are read by readLine in the form of string data.

|  |
| --- |
| String no = br.readline() |

To retrieve integer, we use

|  |
| --- |
| int value = Integer.parseInt(no) |

Since typecasting is done only between data types and String is a class. we cannot typecast String to an integer so we use parseInt method of Integer Wrapper class. Similarly for other data types, refer the below table

|  |  |
| --- | --- |
| Data Type | Statement |
| Integer | int value = Integer.parseInt(br.readline() ); |
| Long | long value = Long.parseLong(br.readline() ); |
| Float | float value = Float.parseFloat(br.readline() ); |
| Double | double value = Double.parseDouble(br.readline() ); |
| Boolean | boolean value = Boolean.parseBoolean(br.readline() ); |
| Byte | byte value = Byte.parseByte(br.readline() ); |
| Short | short value = Short.parseShort(br.readline() ); |

java.util.Scanner and System.in:-

Combining System.in and java.util.Scanner provides a way to read user input that can run inside an IDE. It also provides a way to read different data types.

**1.read string input:**

import java.util.Scanner;

public class Test {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("What is your favorite color? ");

String name = scanner.next();

System.out.println("Your favorite color is: " + name);

}

}

Sample output:

What is your favorite color? blue

Your favorite color is: blue

**2.read byte input:**

import java.util.Scanner;

public class Test {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a small number: ");

byte number = scanner.nextByte();

System.out.println("The number is: " + number);

}

}

Sample output:

Enter a small number: 5

The number is: 5

**3.read short input:**

short number = scanner.*nextShort();*

**4.read int input:**

int number = scanner.nextInt();

**5.read long input:**

long number = scanner.nextLong();

**6.read float input:**

float number = scanner.nextFloat();

**7.read double input:**

double number = scanner.nextDouble();

**8.read boolean input:**

boolean bool = scanner.nextBoolean();

Encapsulation

**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

public class Encapsulation

{

public static void main(String[] args)

{ Emp e=new Emp();

e.setEmpId(6);

e.setEmpNm("aarti");

System.out.println(e.getEmpId());

System.out.println(e.getEmpNm());

}

}

class Emp

{

private int empId;

private String empNm;

public int getEmpId()

{

return empId;

}

public void setEmpId(int empId)

{

this.empId = empId;

}

public String getEmpNm()

{

return empNm;

}

public void setEmpNm(String empNm)

public class Encapsulation

{

public static void main(String[] args)

{

Emp e=new Emp();

e.setEmpId(6);

e.setEmpNm("aarti");

System.out.println(e.getEmpId());

System.out.println(e.getEmpNm());

}

}

class Emp

{

private int empId;

private String empNm;

public int getEmpId()

{ return empId; }

public void setEmpId(int empId)

{ this.empId = empId; }

public String getEmpNm()

{ return empNm; }

public void setEmpNm(String empNm)

{ this.empNm = empNm; }

}

Inheritance.

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

**Child Class:**  
The class that extends the features of another class is known as child class, sub class or derived class.

**Parent Class:**  
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Syntax

To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

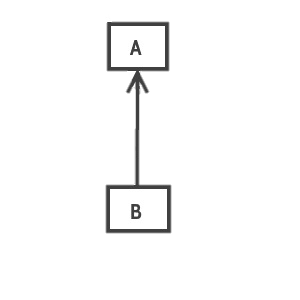
class XYZ extends ABC

{

}

**Types of inheritance**

1.Single Inheritance:

To learn types of inheritance in detail, refer: [Types](https://beginnersbook.com/2013/05/java-inheritance-types/) of Inheritance in Java.  
**Single Inheritance**: refers to a child and parent class relationship where a class extends the another class.  


public class Singleinheritance

{

public static void main(String[] args)

{

Sub a1=new Sub();

a1.num1=5;

a1.num2=50;

a1.sum();

a1.sub();

}

}

class Add

{

int num1,num2,result;

public void sum()

{

result=num1+num2;

System.out.println(result);

}

}

class Sub extends Add

{

public void sub()

{

result=num1-num2;

System.out.println(result);

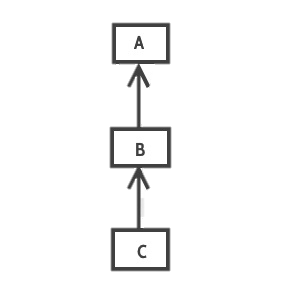
}

}

*2.Multilevelinheritance:*

[Multilevel inheritance](https://beginnersbook.com/2013/12/multilevel-inheritance-in-java-with-example/): refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class

extends class A.



public class Multilevelinheritance

{

public static void main(String[] args)

{

child3 x =new child3();

x.show();

}

}

class child1

{

String str ="this is ";

}

class child2 extends child1

{

child2()

{

str =str.concat("multilevel inheritance");

}

}

class child3 extends child2

{

child3()

{

str =str.concat(" Example");

}

void show()

{

System.out.println(str);

}

public class Multilevelinheritance

{

public static void main(String[] args)

{

child3 x =new child3();

x.show();

}

}

class child1

{

String str ="this is ";

}

class child2 extends child1

{

child2()

{

str =str.concat("multilevel inheritance");

}

}

class child3 extends child2

{

child3()

{

str =str.concat(" Example");

}

void show()

{

System.out.println(str);

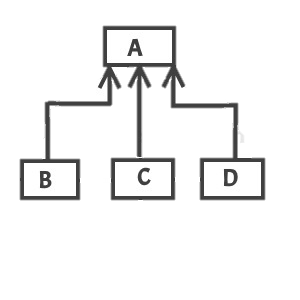
}

}

}

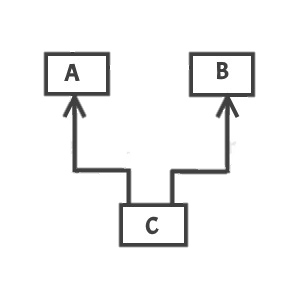
[*Hierarchical inheritance*](https://beginnersbook.com/2013/10/hierarchical-inheritance-java-program/)*:*

refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.



***Multiple Inheritance:***

refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn’t support multiple inheritance, read more about it [here](https://beginnersbook.com/2013/05/java-multiple-inheritance/).



Keywords in java:

**1.‘this’ reference in Java**

‘this’ is a reference variable that refers to the current object.  
  
Following are the ways to use ‘this’ keyword in java :  
  
 **1. Using ‘this’ keyword to refer current class instance variables**

|  |
| --- |
| class Test  {      int a;      int b;           // Parameterized constructor      Test(int a, int b)      {          this.a = a;          this.b = b;      }       void display()      {       //Displaying value of variables a and b          System.out.println("a = " + a + "  b = " + b);      }       public static void main(String[] args)      {          Test object = new Test(10, 20);          object.display();      }  } |

Run on IDE

Output:

a = 10 b = 20

**2. Using this() to invoke current class constructor**

|  |
| --- |
| // Java code for using this() to  // invoke current class constructor  class Test  {      int a;      int b;        //Default constructor      Test()      {          this(10, 20);          System.out.println("Inside  default constructor \n");      }        //Parameterized constructor      Test(int a, int b)      {          this.a = a;          this.b = b;          System.out.println("Inside parameterized constructor");      }        public static void main(String[] args)      {          Test object = new Test();      }  } |

Run on IDE

Output:

Inside parameterized constructor

Inside default constructor

**3. Using ‘this’ keyword to return the current class instance**

|  |
| --- |
| //Java code for using 'this' keyword  //to return the current class instance  class Test  {      int a;      int b;        //Default constructor      Test()      {          a = 10;          b = 20;      }        //Method that returns current class instance      Test get()      {          return this;      }        //Displaying value of variables a and b      void display()      {          System.out.println("a = " + a + "  b = " + b);      }        public static void main(String[] args)      {          Test object = new Test();          object.get().display();      }  } |

Run on IDE

Output:

a = 10 b = 20

*4. Using ‘this’ keyword as method parameter*

|  |
| --- |
| // Java code for using 'this'  // keyword as method parameter  class Test  {      int a;      int b;        // Default constructor      Test()      {          a = 10;          b = 20;      }        // Method that receives 'this' keyword as parameter      void display(Test obj)      {          System.out.println("a = " + a + "  b = " + b);      }        // Method that returns current class instance      void get()      {          display(this);      }        public static void main(String[] args)      {          Test object = new Test();          object.get();      }  } |

Run on IDE

Output:

a = 10 b = 20

*5. Using ‘this’ keyword to invoke current class method*

|  |
| --- |
| // Java code for using this to invoke current  // class method  class Test {        void display()      {          // calling fuction show()          this.show();           System.out.println("Inside display function");      }        void show() {          System.out.println("Inside show funcion");      }          public static void main(String args[]) {          Test t1 = new Test();          t1.display();      }  } |

Run on IDE

Output :

Inside show funcion

Inside display function

*6. Using ‘this’ keyword as an argument in the constructor call*

|  |
| --- |
| // Java code for using this as an argument in constructor  // call  // Class with object of Class B as its data member  class A  {      B obj;        // Parameterized constructor with object of B      // as a parameter      A(B obj)      {          this.obj = obj;         // calling display method of class B          obj.display();      }    }    class B  {      int x = 5;        // Default Contructor that create a object of A      // with passing this as an argument in the     // constructor      B()      {          A obj = new A(this);      }        // method to show value of x      void display()      {          System.out.println("Value of x in Class B : " + x);      }        public static void main(String[] args) {          B obj = new B();      }  } |

Run on IDE

Output :

Value of x in Class B : 5

**2.Super keyword in java:**

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

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

**1.super is 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.

public class Superwithvariable {

public static void main(String[] args)

{

Dog d =new Dog();

d.display();

}

}

class Animal

{

String color="White";

}

class Dog extends Animal

{

String color="black";

void display()

{

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

## 2. super can be used to invoke 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.

public class Superwithmethod

{

public static void main(String[] args)

{

CirclePeremeter c=new CirclePeremeter();

c.cal();

}

}

class CircleArea

{

float pi=3.14f;

int r=1;

float result;

void cal()

{

result=pi\*r\*r;

System.out.println("area of circle"+result);

}

}

class CirclePeremeter extends CircleArea

{

void cal()

{

super.cal();

result=2\*pi\*r;

System.out.println("Perimeter of circle :"+result);

}

}

## 3.super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

public class Superwithconstructor

{

public static void main(String[] args)

{

B b =new B();

}

}

class A

{

A()

{

System.out.println("defalut constructor of class A");

}

A(int x)

{

System.out.println("parameterise constructor of class A");

}

}

class B extends A

{

B()

{

super(2);

System.out.println("default constructor of class B");

}

B(int x)

{

System.out.println("parameterised constructor of class B");

}

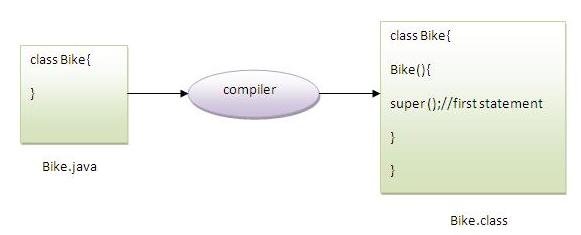
}

# Output:-

parameterise constructor of class A

default constructor of class B

#### Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

# super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor

class Person

{

int id;

String name;

Person(int id,String name)

{

this.id=id;

this.name=name;

}

}

class Emp extends Person

{

float salary;

Emp(int id,String name,float salary)

{

super(id,name);//reusing parent constructor

this.salary=salary;

}

void display()

{

System.out.println(id+" "+name+" "+salary);}

}

class TestSuper5

{

public static void main(String[] args)

{

Emp e1=new Emp(1,"ankit",45000f);

e1.display();

}

}

Output:-

1 ankit 45000

**3.The‘static’ can be:**

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

* variable (also known as class variable)
* method (also known as class method)
* block

**1) Java static variable:**

If you declare any variable as static, it is known static variable.

The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.

The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program memory efficient (i.e it saves memory).

Understanding problem without static variable

class Student{

int rollno;

String name;

String college="ITS";

}

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

Java static property is shared to all objects.

**Example of static variable**

//Program of static variable

class Student8{

int rollno;

String name;

static String college ="ITS";

Student8(int r,String n){

rollno = r;

name = n;

}

void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){

Student8 s1 = new Student8(111,"Karan");

Student8 s2 = new Student8(222,"Aryan");

s1.display();

s2.display();

}

}

Output:

111 Karan ITS

222 Aryan ITS

**Program of counter by static variable**

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

class Counter2{

static int count=0;//will get memory only once and retain its value

Counter2(){

count++;

System.out.println(count);

}

public static void main(String args[]){

Counter2 c1=new Counter2();

Counter2 c2=new Counter2();

Counter2 c3=new Counter2();

}

}

Output:

1

2

3

**2) Java static method**

If you apply static keyword with any method, it is known as static method.

A static method belongs to the class rather than object of a class.

A static method can be invoked without the need for creating an instance of a class.

static method can access static data member and can change the value of it.

**Example of static method**

//Program of changing the common property of all objects(static field).

class Student9{

int rollno;

String name;

static String college = "ITS";

static void change(){

college = "BBDIT";

}

Student9(int r, String n){

rollno = r;

name = n;

}

void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){

Student9.change();

Student9 s1 = new Student9 (111,"Karan");

Student9 s2 = new Student9 (222,"Aryan");

Student9 s3 = new Student9 (333,"Sonoo");

s1.display();

s2.display();

s3.display();

}

}

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

**example of static method that performs normal calculation**

//Program to get cube of a given number by static method

class Calculate{

static int cube(int x){

return x\*x\*x;

}

public static void main(String args[]){

int result=Calculate.cube(5);

System.out.println(result);

}

}

Output:125

***Restrictions for static method***

There are two main restrictions for the static method. They are:

The static method can not use non static data member or call non-static method directly.

this and super cannot be used in static context.

class A{

int a=40;//non static

public static void main(String args[]){

System.out.println(a);

}

}

Test it Now

Output:Compile Time Error

**Q) why java main method is static?**

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

**3) Java static block**

Is used to initialize the static data member.

It is executed before main method at the time of classloading.

Example of static block

class A2{

static{System.out.println("static block is invoked");}

public static void main(String args[]){

System.out.println("Hello main");

}

}

Output:static block is invoked

Hello main

**Q) Can we execute a program without main() method?**

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

class A3{

static{

System.out.println("static block is invoked");

System.exit(0);

}

}

Test it Now

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

Output:Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

**4.Final Keyword**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

*1.variable*

If you make any variable as final, you cannot change the value of final variable(It will be constant).

public class Finalwithvariable {

public static void main(String[] args)

{

final int i=3;

i++;

System.out.println("Value of i : "+i);

}

}

Output:Compile Time Error

*2.Java final method*

f you make any method as final, you cannot override it.

**Example of final method**

class Bike{

final void run(){System.out.println("running");}

}

class Honda extends Bike{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda honda= new Honda();

honda.run();

}

}

Test it Now

Output:Compile Time Error

*3. Java final class*

If you make any class as final, you cannot extend it.

### Example of final class

final class Bike{}

class Honda1 extends Bike{

  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){

  Honda1 honda= new Honda1();

  honda.run();

  }

}

Output:Compile Time Error

Polymorphism:

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

Types of polymorphism

1.Compile time Polymorphism/method overloading:

public class Polycompiletime

{

public static void main(String[] args)

{

overload o=new overload();

o.show();

o.show(5.6);

o.show(8);

o.show(4, 6);

}

}

class overload

{

void show()

{

System.out.println("hi");

}

void show(int a)

{

System.out.println("a :"+a);

}

void show(double b)

{

System.out.println("b :"+b);

}

void show(int a,int b)

{

System.out.println("Addotion of a and b : "+(a+b));

}

}

# Output:-

2.Run time Polymorphism/method overriding/dynamic binding/letbinding

package polyruntime;

public class Polyruntime

{

public static void main(String[] args)

{

A obj1=new A();

B obj2=new B();

A a;

a=obj2;

a.show();

}

}

class A

{

public void show()

{

System.out.println("in show A");

}

}

class B extends A

{

public void show()

{

System.out.println("in show B");

}

}

#### Note:if you have to pass obj1 to reference a then it will display show() of class A.

# Output:-

in show B

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.

It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

1) By nulling a reference:

Employee e=new Employee();

e=null;

2) By assigning a reference to another:

Employee e1=new Employee();

Employee e2=new Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

3) By annonymous object:

new Employee();

*finalize() method*

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

*protected void finalize(){}*

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

*gc() method*

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

*public static void gc(){}*

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

public class TestGarbage1{

public void finalize(){System.out.println("object is garbage collected");}

public static void main(String args[]){

TestGarbage1 s1=new TestGarbage1();

TestGarbage1 s2=new TestGarbage1();

s1=null;

s2=null;

System.gc();

}

}

Output:-

object is garbage collected

object is garbage collected

Note: Neither finalization nor garbage collection is guaranteed.

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.

How to create String object?

|  |
| --- |
| There are two ways to create String object:   1. By string literal 2. By new keyword |

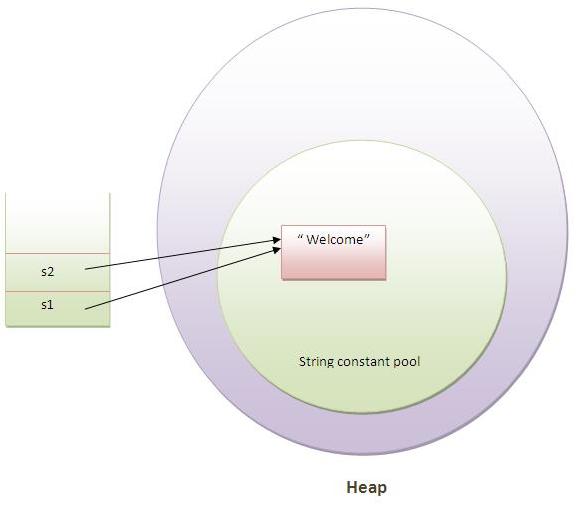
1) String Literal

Java String literal is created by using double quotes. For Example:

String s="welcome";

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

**Note: String objects are stored in a special memory area known as string constant pool.**

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=StringExample)

java

strings

example

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** |  |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) |  | |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index | |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char  value | |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string | |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string | |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with object | |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty | |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatinates specified string | |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified char value | |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of specified CharSequence | |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. | |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex | |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns splitted string matching regex and limit | |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns interned string | |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index | |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified char value index starting with given index | |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index | |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns specified substring index starting with given index | |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase. | |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns string in lowercase using specified locale. | |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase. | |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns string in uppercase using specified locale. | |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. | |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is overloaded. | |

Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

**Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.**

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

**class** StringBufferExample{

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

StringBuffer sb=**new** StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

**class** StringBufferExample2{

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

StringBuffer sb=**new** StringBuffer("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

**class** StringBufferExample3{

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

StringBuffer sb=**new** StringBuffer("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

**class** StringBufferExample4{

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

StringBuffer sb=**new** StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

**class** StringBufferExample5{

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

StringBuffer sb=**new** StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK Java StringBuilder Examples

The StringBuilder append() method concatenates the given argument with this string.

**class** StringBuilderExample{

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

StringBuilder sb=**new** StringBuilder("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

 toString() method

The toString() method returns the string representation of the object.

If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depends on your implementation.

## Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

### Understanding problem without toString() method

Let's see the simple code that prints reference.

**class** Student{

**int** rollno;

 String name;

 String city;

 Student(**int** rollno, String name, String city){

**this**.rollno=rollno;

**this**.name=name;

**this**.city=city;

 }

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

   Student s1=**new** Student(101,"Raj","lucknow");

   Student s2=**new** Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()

   System.out.println(s2);//compiler writes here s2.toString()

 }

}

Output:

Student@1fee6fc

Student@1eed786

|  |  |
| --- | --- |
| As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below: |  |

## Example of Java toString() method

Now let's see the real example of toString() method.

**class** Student{

**int** rollno;

 String name;

 String city;

 Student(**int** rollno, String name, String city){

**this**.rollno=rollno;

**this**.name=name;

**this**.city=city;

 }

**public** String toString(){//overriding the toString() method

**return** rollno+" "+name+" "+city;

 }

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

   Student s1=**new** Student(101,"Raj","lucknow");

   Student s2=**new** Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()

   System.out.println(s2);//compiler writes here s2.toString()

 }

}

Output:101 Raj lucknow

102 Vijay ghaziabad

StringTokenizer in Java

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

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

### example1

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

**import** java.util.StringTokenizer;

**public** **class** Simple{

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

   StringTokenizer st = **new** StringTokenizer("my name is khan"," ");

**while** (st.hasMoreTokens()) {

         System.out.println(st.nextToken());

     }

   }

}

Output:my

name

is

khan

### example 2

**import** java.util.\*;

**public** **class** Test {

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

       StringTokenizer st = **new** StringTokenizer("my,name,is,khan");

      // printing next token

      System.out.println("Next token is : " + st.nextToken(","));

   }

}

Output:Next token is : my

#### StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

Interface in Java

1. [Interface](https://www.javatpoint.com/interface-in-java)
2. [Example of Interface](https://www.javatpoint.com/interface-in-java#interfaceex)
3. [Multiple inheritance by Interface](https://www.javatpoint.com/interface-in-java#interfacemultiple)
4. [Why multiple inheritance is supported in Interface while it is not supported in case of class.](https://www.javatpoint.com/interface-in-java#interfacewhynot)
5. [Marker Interface](https://www.javatpoint.com/interface-in-java#interfacemarker)
6. [Nested Interface](https://www.javatpoint.com/nested-interface)

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

## Why use Java interface?

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

## How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

**interface** <interface\_name>{

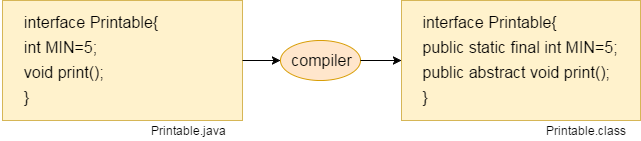
    // declare constant fields

    // declare methods that abstract

    // by default.

}

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



#### The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



## Java Interface Example

## Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

//Interface declaration: by first user

**interface** Drawable{

**void** draw();

}

//Implementation: by second user

**class** Rectangle **implements** Drawable{

**public** **void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle **implements** Drawable{

**public** **void** draw(){System.out.println("drawing circle");}

}

//Using interface: by third user

**class** TestInterface1{

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

Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()

d.draw();

}}

Output:

drawing circle

Multiple inheritance in Java by interface:-

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

**interface** Printable{

**void** print();

}

**interface** Showable{

**void** show();

}

**class** A7 **implements** Printable,Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **void** show(){System.out.println("Welcome");}

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

A7 obj = **new** A7();

obj.print();

obj.show();

 }

}

Output:Hello

Welcome

## *Interface inheritance:-*

A class implements an interface, but one interface extends another interface.

**interface** Printable{

**void** print();

}

**interface** Showable **extends** Printable{

**void** show();

}

**class** TestInterface4 **implements** Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **void** show(){System.out.println("Welcome");}

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

TestInterface4 obj = **new** TestInterface4();

obj.print();

obj.show();

 }

}

Output:

Hello

Welcome

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

### Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

**Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

**Example of abstract class**

abstract class A{}

**Abstract Method in Java**

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

**abstract** **void** printStatus();//no method body and abstract

### Example of Abstract class that has an abstract method

**abstract** **class** Shape{

**abstract** **void** draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

**class** Rectangle **extends** Shape{

**void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle1 **extends** Shape{

**void** draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

**class** TestAbstraction1{

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

Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method

s.draw();

}

}

Output:-

drawing circle

**Example2:-**

**abstract** **class** Bike{

   Bike(){System.out.println("bike is created");}

**abstract** **void** run();

**void** changeGear(){System.out.println("gear changed");}

 }

//Creating a Child class which inherits Abstract class

**class** Honda **extends** Bike{

**void** run(){System.out.println("running safely..");}

 }

//Creating a Test class which calls abstract and non-abstract methods

**class** TestAbstraction2{

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

  Bike obj = **new** Honda();

  obj.run();

  obj.changeGear();

 }

}

Output:-

bike is created

running safely..

gear changed

#### Rule: If there is an abstract method in a class, that class must be abstract.

#### Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Java Inner Classes

**Java inner class** or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

#### Syntax of Inner class

**class** Java\_Outer\_class{

 //code

**class** Java\_Inner\_class{

  //code

 }

}

**Types of Nested classes:-**

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing interface or extending class.  Its name is decided by the java compiler. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class created within method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class created within class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

## 1.Java Member inner class:-

A non-static class that is created inside a class but outside a method is called member inner class.

**Syntax:**

**class** Outer{

 //code

**class** Inner{

  //code

## }

}

## example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

**class** TestMemberOuter1{

**private** **int** data=30;

**class** Inner{

**void** msg(){System.out.println("data is "+data);}

 }

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

  TestMemberOuter1 obj=**new** TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.**new** Inner();

  in.msg();

 }

}

## Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

**Example:-**

**public** **class** localInner1{

**private** **int** data=30;//instance variable

**void** display(){

**class** Local{

**void** msg(){System.out.println(data);}

  }

  Local l=**new** Local();

  l.msg();

 }

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

  localInner1 obj=**new** localInner1();

  obj.display();

 }

}

Output:

30

## Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

### example

**abstract** **class** Person{

**abstract** **void** eat();

}

**class** TestAnonymousInner{

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

  Person p=**new** Person(){

**void** eat(){System.out.println("nice fruits");}

  };

  p.eat();

 }

}

Output:

nice fruits

## Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

## Java static nested class example with instance method

**class** TestOuter1{

**static** **int** data=30;

**static** **class** Inner{

**void** msg(){System.out.println("data is "+data);}

  }

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

  TestOuter1.Inner obj=**new** TestOuter1.Inner();

  obj.msg();

  }

}

Output:

data is 30

## Java Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

### Syntax :-

**interface** interface\_name{

 ...

**interface** nested\_interface\_name{

  ...

## }

}

## Example of nested interface which is declared within the interface

|  |
| --- |
| In this example, we are going to learn how to declare the nested interface and how we can access it. |

**interface** Showable{

**void** show();

**interface** Message{

**void** msg();

  }

}

**class** TestNestedInterface1 **implements** Showable.Message{

**public** **void** msg(){System.out.println("Hello nested interface");}

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

  Showable.Message message=**new** TestNestedInterface1();//upcasting here

  message.msg();

 }

}

Output:hello nested interface

|  |
| --- |
|  |

Exception Handling in Java

1. [Exception Handling](https://www.javatpoint.com/exception-handling-in-java)
2. [Advantage of Exception Handling](https://www.javatpoint.com/exception-handling-in-java#exceptionad)
3. [Hierarchy of Exception classes](https://www.javatpoint.com/exception-handling-in-java#exceptionhierarchy)
4. [Types of Exception](https://www.javatpoint.com/exception-handling-in-java#exceptiontypes)
5. [Exception Example](https://www.javatpoint.com/exception-handling-in-java#exceptionexample)
6. [Scenarios where an exception may occur](https://www.javatpoint.com/exception-handling-in-java#exceptionscenarios)

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

## What is Exception in Java

**Dictionary Meaning:** Exception is an abnormal condition.

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

## What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

## Difference between Checked and Unchecked Exceptions

### 1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| Try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| Catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| Finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| Throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

There are 5 keywords which are used in handling exceptions in Java.

## Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

**public** **class** JavaExceptionExample{

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

**try**{

      //code that may raise exception

**int** data=100/0;

   }**catch**(ArithmeticException e){System.out.println(e);}

   //rest code of the program

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

  }

}

Output:

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

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### *1) A scenario where ArithmeticException occurs*

If we divide any number by zero, there occurs an ArithmeticException.

**int** a=50/0;//ArithmeticException

### *2) A scenario where NullPointerException occurs*

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

String s=**null**;

System.out.println(s.length());//NullPointerException

### *3) A scenario where NumberFormatException occurs*

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

String s="abc";

**int** i=Integer.parseInt(s);//NumberFormatException

### *4) A scenario where ArrayIndexOutOfBoundsException occurs*

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

**int** a[]=**new** **int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Java Multi catch block

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

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

**public** **class** TestMultipleCatchBlock{

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

**try**{

**int** a[]=**new** **int**[5];

    a[5]=30/0;

   }

**catch**(ArithmeticException e){System.out.println("task1 is completed");}

**catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

**catch**(Exception e){System.out.println("common task completed");}

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

 }

}

Output:

task1 completed

rest of the code...

# *Java Nested try block*

The try block within a try block is known as nested try block in java.

### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### Syntax:

....

**try**

{

    statement 1;

    statement 2;

**try**

    {

        statement 1;

        statement 2;

    }

**catch**(Exception e)

    {

    }

}

**catch**(Exception e)

{

}

....

***Example:***

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

**class** Excep6{

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

**try**{

**try**{

     System.out.println("going to divide");

**int** b =39/0;

    }**catch**(ArithmeticException e){System.out.println(e);}

**try**{

**int** a[]=**new** **int**[5];

    a[5]=4;

    }**catch**(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);

  }**catch**(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");

 }

}

# *Java finally block*

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

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

Java finally block follows try or catch block.

#### Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

## Why use java finally

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

class TestFinallyBlock{

public static void main(String args[]){

try{

int data=25/5;

System.out.println(data);

}

catch(NullPointerException e){System.out.println(e);}

finally{System.out.println("finally block is always executed");}

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

}

}

Test it Now

Output:5

finally block is always executed

rest of the code...

Java throw keyword

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

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

## java throw keyword example

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

**public** **class** TestThrow1{

**static** **void** validate(**int** age){

**if**(age<18)

**throw** **new** ArithmeticException("not valid");

**else**

      System.out.println("welcome to vote");

   }

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

      validate(13);

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

  }

}

Output:

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

# Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java#multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java#multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java#multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java#thread)

Multithreading in java

is a process of executing multiple threads simultaneously.

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

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

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

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

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

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

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

Process-based Multitasking (Multiprocessing)

Thread-based Multitasking (Multithreading)

**1) Process-based Multitasking (Multiprocessing)**

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

**2) Thread-based Multitasking (Multithreading)**

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface. Java Thr

ead Methods

Java Thread Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | Void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | Void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing  thread object. |
| 5) | Void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | Int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | Void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | Void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | Long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | Boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |

Life cycle of a Thread (Thread States)

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

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

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

1. New

2. Runnable

3. Running

4. Non-Runnable (Blocked)

5. Terminated

**1) New**

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) 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.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

**How to create thread**

There are two ways to create a thread:

1. By extending Thread class

class hi extends Thread

{

public void run()

{

for(int i=0;i<=4;i++)

{

System.out.println("hi");

try{Thread.sleep(1000);}catch(Exception e){}

}

}

}

class hello extends Thread

{

public void run()

{

for(int i=0;i<=4;i++)

{

System.out.println("hello");

try{Thread.sleep(1000);}catch(Exception e){}

}

}

}

class threadedemo

{

public static void main(String ar[])

{

hi h=new hi();

hello ho=new hello();

h.start();

try{Thread.sleep(500);}catch(Exception e){}

ho.start();

}

}

1. By implementing Runnable interface.

/\*class hi implements Runnable

{

public void run()

{

for(int i=0;i<=4;i++)

{

System.out.println("hi");

try{Thread.sleep(1000);}catch(Exception e){}

}

}

}

class hello implements Runnable

{

public void run()

{

for(int i=0;i<=4;i++)

{

System.out.println("hello");

try{Thread.sleep(1000);}catch(Exception e){}

}

}

}

class threadedemo

{

public static void main(String ar[])

{

hi h=new hi();

hello ho=new hello();

Thread t=new Thread(h);

Thread t1=new Thread(ho);

t.start();

try{Thread.sleep(500);}catch(Exception e){}

t1.start();

}

}

# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

**public** **class** TestThreadTwice1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

 }

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

  TestThreadTwice1 t1=**new** TestThreadTwice1();

  t1.start();

  t1.start();

 }

}

**Output:-**

running

Exception in thread "main" java.lang.IllegalThreadStateException

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

### *Syntax:*

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

**class** TestJoinMethod1 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<=5;i++){

**try**{

    Thread.sleep(500);

   }**catch**(Exception e){System.out.println(e);}

  System.out.println(i);

  }

 }

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

 TestJoinMethod1 t1=**new** TestJoinMethod1();

 TestJoinMethod1 t2=**new** TestJoinMethod1();

 TestJoinMethod1 t3=**new** TestJoinMethod1();

 t1.start();

**try**{

  t1.join();

 }**catch**(Exception e){System.out.println(e);}

 t2.start();

 t3.start();

 }

}

Output:

1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

# getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

**class** TestJoinMethod3 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

  }

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

  TestJoinMethod3 t1=**new** TestJoinMethod3();

  TestJoinMethod3 t2=**new** TestJoinMethod3();

  System.out.println("Name of t1:"+t1.getName());

  System.out.println("Name of t2:"+t2.getName());

  System.out.println("id of t1:"+t1.getId());

  t1.start();

  t2.start();

  t1.setName("Sonoo Jaiswal");

  System.out.println("After changing name of t1:"+t1.getName());

 }

}

Output:

Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

**class** TestJoinMethod4 **extends** Thread{

**public** **void** run(){

  System.out.println(Thread.currentThread().getName());

 }

 }

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

  TestJoinMethod4 t1=**new** TestJoinMethod4();

  TestJoinMethod4 t2=**new** TestJoinMethod4();

  t1.start();

  t2.start();

 }

}

Output:

Thread-0

Thread-1

# Priority of a Thread (Thread Priority):

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

## 3 constants defined in Thread class:

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

|  |
| --- |
| Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |

### Example of priority of a Thread:

**class** TestMultiPriority1 **extends** Thread{

**public** **void** run(){

   System.out.println("running thread name is:"+Thread.currentThread().getName());

   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }

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

  TestMultiPriority1 m1=**new** TestMultiPriority1();

  TestMultiPriority1 m2=**new** TestMultiPriority1();

  m1.setPriority(Thread.MIN\_PRIORITY);

  m2.setPriority(Thread.MAX\_PRIORITY);

  m1.start();

  m2.start();

 }

}

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

### Advantage of Applet

There are many advantages of applet. They are as follows:

It works at client side so less response time.

Secured

It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

Plugin is required at client browser to execute applet.

# Hierarchy of Applet

# As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

# Lifecycle of Java Applet

# Applet is initialized.

# Applet is started.

# Applet is painted.

# Applet is stopped.

# Applet is destroyed.

# Lifecycle methods for Applet:

# The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

# java.applet.Applet class

# For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

# public void init(): is used to initialized the Applet. It is invoked only once.

# public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

# public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

# public void destroy(): is used to destroy the Applet. It is invoked only once.

# java.awt.Component class

# The Component class provides 1 life cycle method of applet.

# public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

# Who is responsible to manage the life cycle of an applet?

# Java Plug-in software.

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# How to run an Applet?

# There are two ways to run an applet

# By html file.

# By appletViewer tool (for testing purpose).

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Simple example of Applet by html file:

# To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

# //First.java

# import java.applet.Applet;

# import java.awt.Graphics;

# public class First extends Applet{

# public void paint(Graphics g){

# g.drawString("welcome",150,150);

# }

# }

# Note: class must be public because its object is created by Java Plugin software that resides on the browser.

# myapplet.html

# <html>

# <body>

# <applet code="First.class" width="300" height="300">

# </applet>

# </body>

# </html>

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Simple example of Applet by appletviewer tool:

# To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

# //First.java

# import java.applet.Applet;

# import java.awt.Graphics;

# public class First extends Applet{

# public void paint(Graphics g){

# g.drawString("welcome to applet",150,150);

# }

# }

# /\*

# <applet code="First.class" width="300" height="30">

# Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

1. **public** String getParameter(String parameterName)

Example of using parameter in Applet:

**import** java.applet.Applet;

**import** java.awt.Graphics;

**public** **class** UseParam **extends** Applet{

**public** **void** paint(Graphics g){

String str=getParameter("msg");

g.drawString(str,50, 50);

}

}

myapplet.html

<html>

<body>

<applet code="UseParam.class" width="300" height="300">

<param name="msg" value="Welcome to applet">

</applet>

</body>

</html>

</applet>

\*/

To execute the applet by appletviewer tool, write in comParameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter(). Syntax:

public String getParameter(String parameterName)

Example of using parameter in Applet:

import java.applet.Applet;

import java.awt.Graphics;

public class UseParam extends Applet{

public void paint(Graphics g){

String str=getParameter("msg");

g.drawString(str,50, 50);

}

}

myapplet.html

<html>

<body>

<applet code="UseParam.class" width="300" height="300">

<param name="msg" value="Welcome to applet">

</applet>

</body>

</html>mand prompt:

**c:\>**javac First.java

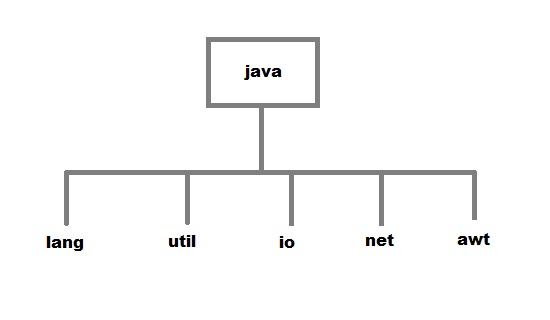
**c:\>**appletviewer First.java

Java Package

Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc. A package can be defined as a group of similar types of classes, interface, enumeration or sub-package. Using package it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

#### Types of Packages: Built-in and User defined

* **Built-in Package:** Existing Java package for example java.lang, java.util etc.
* **User-defined-package:** Java package created by user to categorize their project's classes and interface.



Creating a package

Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

package mypack;

public class employee

{

statement;

}

The above statement will create a package woth name **mypack** in the project directory.

Java uses file system directories to store packages. For example the .java file for any class you define to be part of **mypack** package must be stored in a **directory** called **mypack**.

**Additional points about package:**

* A package is always defined as a separate folder having the same name as the package name.
* Store all the classes in that package folder.
* All classes of the package which we wish to access outside the package must be declared public.
* All classes within the package must have the package statement as its first line.
* All classes of the package must be compiled before use (So that they are error free)

#### Example of Java packages

//save as FirstProgram.java

package learnjava;

public class FirstProgram{

public static void main(String args[]) {

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

}

}

#### How to compile Java programs inside packages?

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

javac -d directory javafilename

**Example:**

javac -d . FirstProgram.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like **d:/abc** (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

#### How to run Java package program?

You need to use fully qualified name e.g. learnjava.FirstProgram etc to run the class.

**To Compile:**

javac -d . FirstProgram.java

To Run:

java learnjava.FirstProgram

Output: Welcome to package

#### import keyword

import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

1. **Using fully qualified name (But this is not a good practice.)**

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface, which can look a little untidy if the package name is long.

This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

**Example :**

//save by A.java

package pack;

public class A {

public void msg() {

System.out.println("Hello");

}

}

//save by B.java

package mypack;

class B {

public static void main(String args[]) {

pack.A obj = new pack.A(); //using fully qualified name

obj.msg();

}

}

**Output:**

Hello

1. **To import only the class/classes you want to use**

If you import packagename.classname then only the class with name **classname** in the package with name **packagename** will be available for use.

**Example :**

//save by A.java

package pack;

public class A {

public void msg() {

System.out.println("Hello");

}

}

//save by B.java

package mypack;

import pack.A;

class B {

public static void main(String args[]) {

A obj = new A();

obj.msg();

}

}

Output:

Hello

1. **To import all the classes from a particular package**

If you use packagename.\*, then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use.

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

**Example :**

//save by First.java

package learnjava;

public class First{

public void msg() {

System.out.println("Hello");

}

}

//save by Second.java

package Java;

import learnjava.\*;

class Second {

public static void main(String args[]) {

First obj = new First();

obj.msg();

}

}

Output:

Hello

#### Points to remember

* When a package name is not specified, the classes are defined into the default package (the current working directory) and the package itself is given no name. That is why, you were able to execute assignments earlier.
* While creating a package, care should be taken that the statement for creating package must be written before any other import statements.

// not allowed

import package p1.\*;

package p3;

Below code is correct, while the code mentioned above is incorrect.

//correct syntax

package p3;

import package p1.\*;