|  |  |
| --- | --- |
| ***James Gosling*** | Java is a versatile and powerful programming language known for its platform independence, object-oriented design, robustness, and security. Its extensive ecosystem and large community of developers make it a popular choice for building a wide range of applications across different domains.  *Prepared By*  **PARANOIA TECHNOLOGIES** |

**Java**

**What is Java?**

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than **3 billion** devices run Java.

It is **used for**:

* Mobile applications (especially Android apps)
* Desktop applications
* Web applications
* Smart Card
* Robotics
* Games

**Types of Java Applications**

There are mainly 4 types of applications that can be created using Java programming:

* Standalone Application/Desktop Applications/Window-based Applications
* Web Application
* Enterprise Application
* Mobile Application

**Java Platforms / Editions**

* Java SE (Java Standard Edition)
* Java EE (Java Enterprise Edition)
* Java ME (Java Micro Edition)
* JavaFX - JavaFX stands for special effects in the Java language. Internet applications are made richer with different plugins with the help of JavaFX.

**Why Use Java?**

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
* It is one of the most popular programming language in the world
* It is easy to learn and simple to use
* It is open-source and free
* It is secure, fast and powerful
* Java is an object oriented language which gives a clear structure to programs and allows code to be reused.
* Reduce the development costs.
* As Java is close to languages like C++ and C#

**History of Java**

* The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.
* [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java)**, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

**Father of JAVA:** [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java)



* Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was **.gt.**
* After that, it was called **Oak** and was developed as a part of the Green project.
* **Oak** is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.



* In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
* **Java** is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

**Features of Java**

Features of java is also known as **Java buzzwords**

**Simple**

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

**Object-oriented**

* Java is an [**object-oriented**](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

**Platform Independent**

* Java is a write once, run anywhere language.

**Secured**

* Java is best known for its security. With Java, we can develop virus-free systems.
* Java programs runs on JRE(Java Runtime Environment) and no interaction with our system OS

**Robust**

Meaning of Robust is strong. Java is robust because:

* Strong memory management.
* Lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* Exception handling and the type checking mechanism in Java.

**Architecture-neutral**

* Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

**Portable**

* Java is portable because it facilitates you to carry the Java byte code to any platform. It doesn't require any implementation.

**High-performance**

* Java is faster than other traditional interpreted programming languages because Java byte code is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
* Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

**Distributed**

* Java is distributed because it facilitates users to create distributed applications in Java.
* RMI and `EJB are used for creating distributed applications. Compared to other languages it is easy to create network connection in Java.

**Multi-threaded**

* A thread is like a separate program, executing concurrently.
* Write Java programs that deal with many tasks at once by defining multiple threads.
* The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.
* Threads are important for multi-media, Web applications, etc.

**Dynamic**

* Java is a dynamic language.
* It supports the dynamic loading of classes.
* It means classes are loaded on demand.

**Source File**

**File name: Hello.java**

public class Hello{

      public static void main(String[] args) {

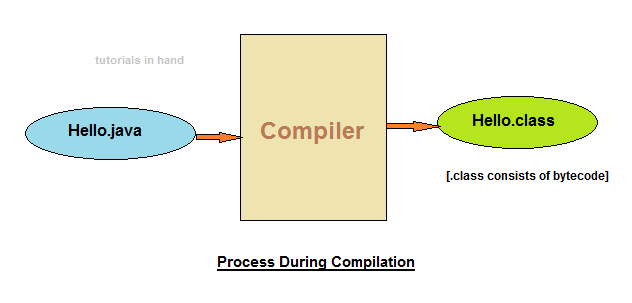
            System.out.print("My first program in java");

      }

}

**Internal working during compile time**

* When you compile the **Hello.java** file then the compiler creates a **.class file**
* **.class file** is created with the name **Hello.class**
* **Hello.class** contains **byte codes**



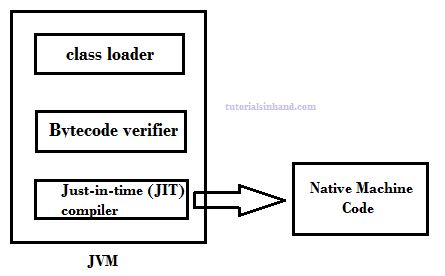
**Byte code (.class file)**is specialized set of instruction that **JVM** can read. Thus this .class file consisting of byte code can be transported to other system, irrespective of the operating system it has, and could be executed comfortably without any issue.

* This is why java a **portable language**.
* Because of the same reason java is also popularly known as **Write once, run anywhere (WORA) language**.

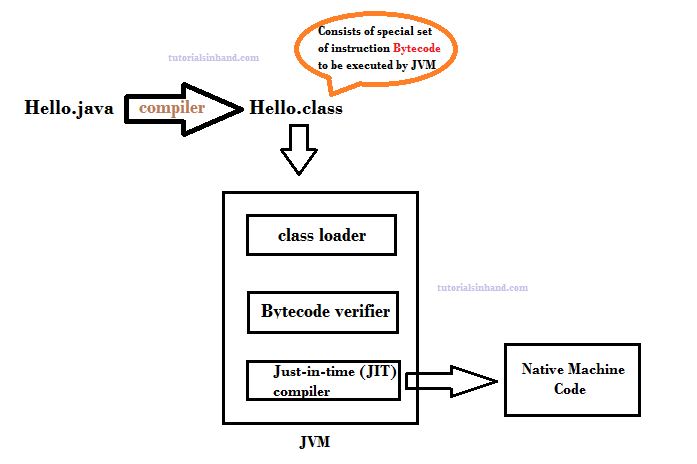
**JVM** is capable of reading, verifying the bytecode.

**Internal working during run time**

During run or execution time following tasks are performed within JVM:

* Class loader loads the .class files containing bytecode to the memory.
* Next the bytecode is verified and checked for any errors or bugs which may result in program exhibiting anomalous behaviour.
* At last, the just-in-time compiler converts the bytecode into machine code.

**Compile + Run diagram of java code**



**Difference between JVM JRE and JDK**

**Java Virtual Machine - JVM**

* JVM is a virtual machine which is capable of reading the ***.class file*** that contains bytecode.
* In java, compiler produces bytecode during compilation which can be run on any system that has JVM installed on it. This results in making java a **portable programming language**. It can be written on any system and run-on different system easily irrespective of operating system. Thus, java is also referred to as **write once, run anywhere**.

 JVM performs three major tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

**Java Runtime Environment - JRE**

* JRE provides environment to run java applications.
* JRE contains supporting libraries, core classes and other components that JVM uses during the runtime.
* JRE is part of JDK. It can also be downloaded separately to just only run java application and applet.



**Java Development Kit – JDK**

JDK comprises of JRE and other tools that helps in developing, debugging & monitoring the java application.

A JDK always comprises of:

* JRE
* Compiler (javac)
* Debbugger
* Java document



**What is a class?**

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**

**Syntax to declare a class:**

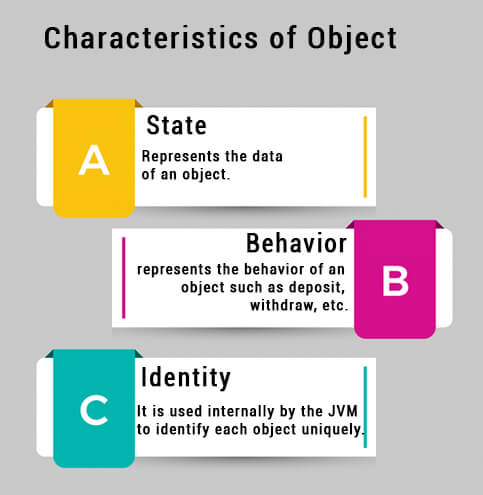
**class** <class name>

{    field;

    method;  }

**What is an object?**

* An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
* **An object is an instance of a class.**
* A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.



**How to create an object?**

**Syntax:**

ClassName object = **new** Constructor();

**Using new Keyword**

Using the **new** keyword is the most popular way to create an object or instance of the class. When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the **reference** of that object to that memory.

**Constructors**

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

**Rules for creating Java constructor**

There are two rules defined for the constructor.

* Constructor name must be the same as its class name
* A Constructor must have no explicit return type
* A Java constructor cannot be abstract, static, final, and synchronized

**Types of Java constructors**

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

**Default Constructor**

A constructor is called "Default Constructor" when it doesn't have any parameter.

**Purpose:**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

**Syntax:**

class name(){}

**Example**

**class** Fruit

{

//creating a default constructor

Fruit ()

{

System.out.println("Fruit is created");

}

//main method

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

{

//calling a default constructor

Fruit  f = **new**  Fruit ();

Constructor Name

Object Name

Class Name

New Keyword

}

}

**Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

**Purpose:**

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

**Example**

**class** Fruit

{

//creating a parameterized constructor

Fruit (String str)

{

System.out.println("Fruit is created as" +str);

}

//main method

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

{

//calling a parameterized constructor

Fruit f=**new** Fruit(“Apple”);

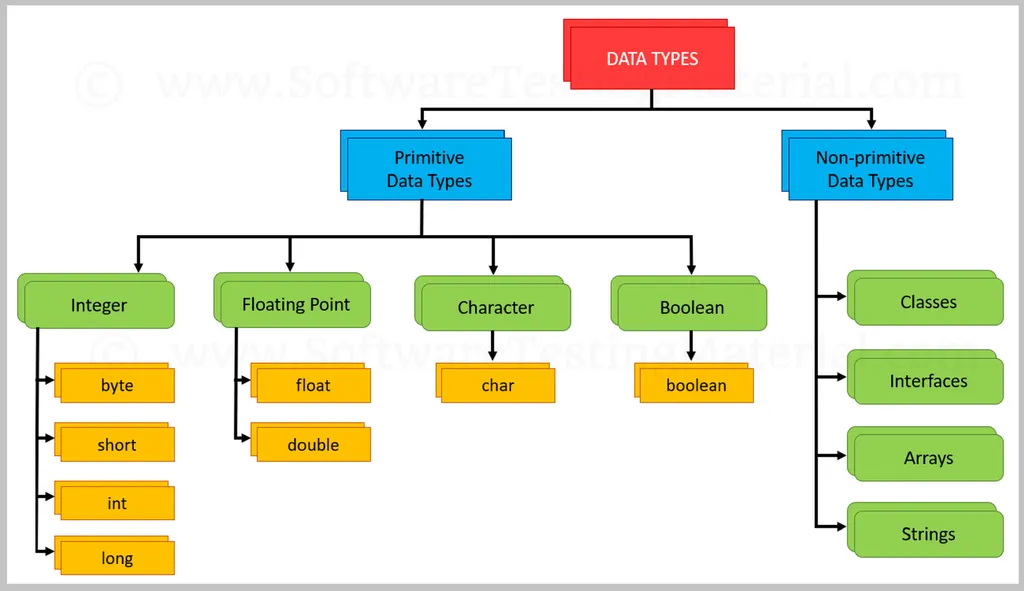
}

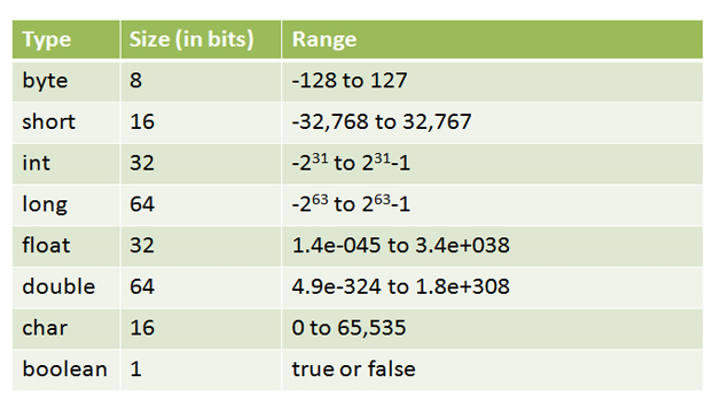
}

**Data Types**

Primarily “**Data types**” in java can be categorized into two types.

* **Primitive data types**
* **Non Primitive or Reference data types.**

****

****

**Variables**

* **Variables** are containers for storing data values.
* **Variables in java**can be thought of as a container that can hold certain values like int, long, byte, etc. during the life time of an application program.

**Syntax**

**type** variablename = value;

**Types of Variables in Java**

* Local Variables
* Instance Variables
* Static Variables

**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.

**Example**

public class A

{

public static void main(String args[])

    {

int a=90;

System.out.println(“value is :”+a);

     }

}//end of class

**2) Instance Variable**

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as [static](https://www.javatpoint.com/static-keyword-in-java).

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

**Example**

public class A

{

int d=50; **//instance variable**

public static void main(String args[])

    {

A a=new A();

System.out.println(“Instance Variable:”+a.d);

     }

}//end of class

**3) Static variable**

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

**Example**

public class A

{

 static int m=100; **//static variable**

   public static void main(String args[])

    {

System.out.println(“Static Variable:”+m);

     }

}//end of class

**Access Modifiers**

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

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

There are four types of access modifiers:

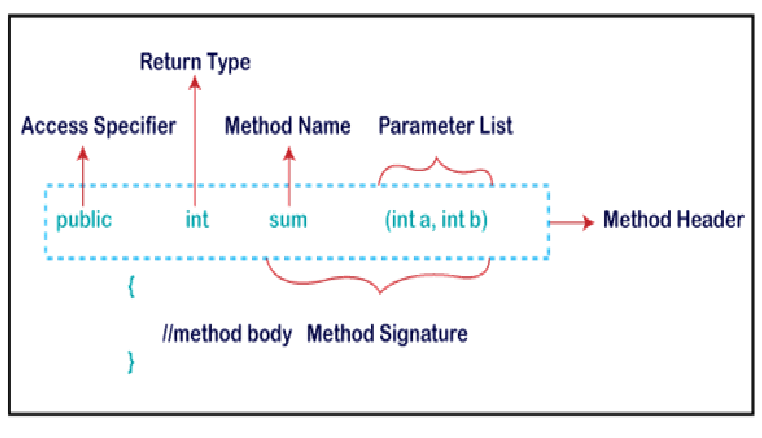
* **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
* **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
* **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
* **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

**What is a method?**

A **method** is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as **functions**.



**Types of Method**

There are two types of methods in Java:

* Predefined Method
* User-defined Method

**Predefined Method**

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**.

**Example**

**public** **class** Demo

{

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

{

**// using the max() method of Math class which is a predefined method**

System.out.print("The maximum number is: " + Math.max(9,7));

}

}

**User-defined Method**

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

**Example**

public class A

{

void show()  **//User-defined Method**

    {

        int n=90**;**

System.out.println(“Variable:”+n);

    }

public static void main(String args[])

    {

A a=new A(); **//Object Creation;**

a.show(); **//Method Calling**

     }

}**//end of class**

**Conditional & Control Statements**

A program written in Java programming language is generally executed by [JVM](https://www.scientecheasy.com/2021/03/what-is-jvm.html/) sequentially (one by one) in the order in which they appear.These statements are called sequential statements. The flow of execution takes place from top to bottom.

There are three main types of flow of execution (control) that occur in any computer programming. They are:

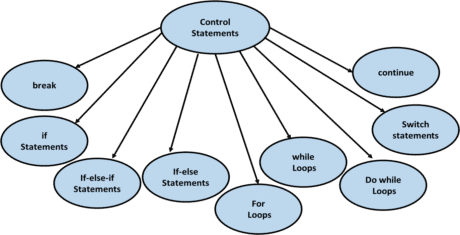
* **Sequential:** Statements execute from top to bottom one by one.
* **Conditional or selection:** Out of two instructions, only one will be executed successfully based on the specified condition. This is because the condition generates the result as either true or false.
* **Repetition or loop:** Group of statements repeats whenever the specified condition is true.

Here are the topics if you want to jump directly:

* [Conditional Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_conditional_control)
  + [If statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/%22#_if_statement)
  + [If-else statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_if_else)
  + [If-else-if statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_if_else_if)
  + Nested if statement
  + [Switch statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_switch_statement)
* [Looping Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_looping_control)
  + [For loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_for_loop)
  + [While loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_while_loop)
  + [Do-while loop](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_do_while)
* [Unconditional Statements/Jump Statements](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_unconditional_control)
  + [break statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_break_statement)
  + [continue statement](https://intellipaat.com/blog/tutorial/java-tutorial/control-statements-in-java/#_continue_statement)

**Control Statements**  
Control Statements are used to control the execution flow of the

program.

[](https://intellipaat.com/mediaFiles/2018/12/ja1.png)

There are three types of control statements:

* **Conditional Control Statements**
* **Looping Control Statements**
* **Unconditional Control Statements/Jump Statements**

**Conditional Statements**

Conditional Statements allows the program to select between the alternatives during the program execution.  
They are also called as **decision-making statements or selection statements.**

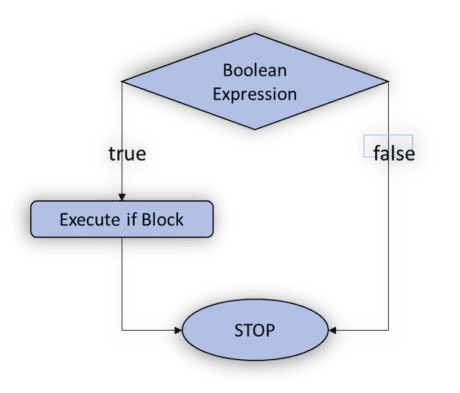
**If statement**

* It will go inside the block only if the condition is true otherwise, it will not execute the block.

**Syntax:**

if (condition)

{  
// statements (if Block)  
}  
//other statements.

**Execution Flow Chart of If Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja2.png)

**Example**

class Condif

{ static

{

System.out.println("Simple If Statement");

}

public static void main(String args[])

{

int n=5;//local variable

if(n==5)

{ System.out.println("n=" +n);

}

System.out.println("Done");

}

}

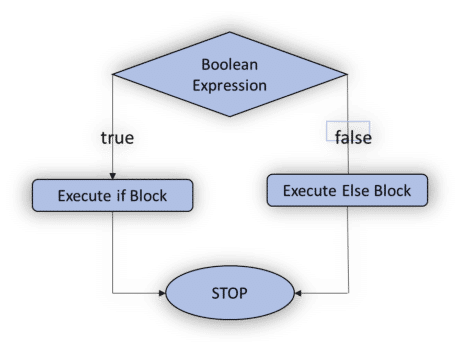
**If-Else Statement**

* If the condition is true then, it will execute the If block. Otherwise, it will execute the Else block.

**Syntax:**

if (condition)

{  
// statements (if Block)  
}  
else{  
// statements (Else block)  
}  
//other statements

**Execution flow chart of If-Else Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja3-1.png)

**Example**

class Condifelse

{

static

{

System.out.println("If else Statement");

}

public static void main(String args[])

{

int n=5;

if(n>4)

{

System.out.println("n=" +n);

}

else

{

System.out.println("Wrong number");

}

}

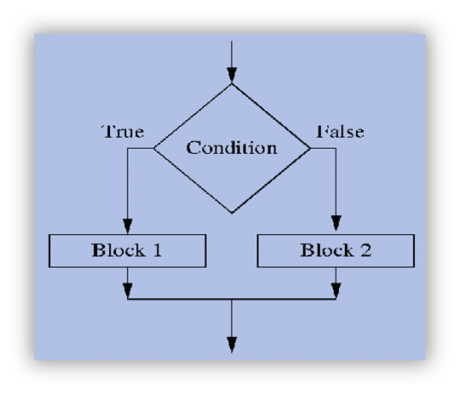
}

**If Else-If statement**

* If the condition is true, then it will execute the If block. Otherwise, it will execute the Else-If block. Again, if the condition is not met, then it will move to the else block.

**Syntax:**

if (condition){  
// statements (if Block)  
}  
else if(condition){  
// statements (Else-If block)  
}  
else{  
//statements(Else Block)  
}//other statements

**Execution flow chart of If Else-If Statement**  
[](https://intellipaat.com/mediaFiles/2018/12/ja4-2.png)

**Example**

class Condifelseif

{

static

{

System.out.println("If else if Statement");

}

public static void main(String args[])

{

int a=10,b=15,c=5;

if((a>b)&&(a>c))

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

else if((b>c)&&(b>a))

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

else

System.

out.println("C is greater:" +c);

}

}

**Nested if statement**

**Example**

class Nestedif

{

static

{

System.out.println("Nested if Statement");

}

public static void main(String args[])

{

int a=70,b=45,c=25,d=50;

if(a>d)

{

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

if(b>c) 45>25

{

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

}

else

{

System.out.println("C is greater:" +c);

}

}

else

{

System.out.println("D is greater:" +d);

}

}

}

**Switch Statement**

* Switch statement allows program to select one action among multiple actions during the program execution.

**Syntax:**

switch(variable/value/expression)

{  
case:  
//statements;

break;  
case:  
//statements;

break;  
default:  
//statements;  
}

* Based on the argument in the switch statement suitable case value will be selected and executed.
* If no matching case found, then the default will be executed.
* It is optional to write a break statement at the end of each case statement.

**Eample**

import java.util.\*;

class Switchprogram

{

public static void main(String []args)

{

System.out.println("Enter the choice from 0-2");

Scanner sc=new Scanner(System.in);

int x=sc.nextInt();

switch(0) switch(value/variable/expression)

{

case 0:

System.out.println("Hai");

break;

case 1:

System.out.println("Hello");

break;

case 2:

System.out.println("Bye");

break;

default:

System.out.println("No choice found");

}

}

}

**Looping Control Statements**

These are used to execute a block of statements multiple times. It means it executes the same code multiple times so it saves code. These are also called Iteration statements.

There are three types of looping control statements:

* **For loop**
* **While loop**
* **Do-while loop**

**For loop**

* It executes the code until condition is false.
* It is used when numbers of iterations are known.

**Syntax:**

for(initialization; condition; increment/decrement)

{  
//statements(For Body)

}

**Example**

public class Forloop

{

public static void main(String[] args)

{

int i;

for(i=0;i<=5;i++)

{

System.out.println("Hello");

}

}

}

**While loop**

* While loop executes till the condition becomes false.

**Syntax:**

while(condition)

{  
// statements

}

**Example**

public class Whileloop

{

public static void main(String[] args)

{

int i=1;

while(i<5)

{

System.out.println("While Loop");

i++;

}

}

}

**Do-while loop**

* When you are using for or while, then it will execute the loop body only if the condition if true.
* In do-while loop, it will execute the loop first, then it checks the condition. So, it will execute the loop atleast once.
* It is called **exit controlled loop** while **for & while loop** are called **entry controlled loop**.

**Syntax:**

do{  
//statements  
}while(condition);

**Example**

public class DoWhileloop {

public static void main(String[] args) {

int a=5;

do

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

a--;

}while(a!=0);

}

}

**Unconditional Control Statements/Jump Statements**

**break Statement**

* break is a keyword. It is used within any control statements. It is used to terminate the execution of the current loop or switch statements.
* **Syntax:** **break;**

**Example**

class Breakprogram

{

public static void main(String args[])

{

int i;

for(i=0;i<5;i++)

{

if(i==3)

break;

System.out.println("Break Statement" +i);

}

}

}

**continue Statement**

* continue is a keyword. It is used to continue the execution of the current loop with the next iteration.

**Syntax:** **continue;**

**Example**

class Continueprogram

{

public static void main(String args[])

{

for(int i=1;i<=4;i++)

{

System.out.println("i="+i);

if(i==3)

continue;

System.out.println("continue1");

}

}

}

**OOPS CONCEPTS**

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)

**Object**

* Any entity that has state and behavior is known as an object. It can be physical or logical.
* An Object can be defined as an instance of a class.
* An object contains an address and takes up some space in memory.
* **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.



**Class**

* Collection of objects is called class. It is a logical entity.
* A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

**Inheritance**

* **Inheritance** is a mechanism in which one class acquires the property of another class.
* With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.
* As we know, a child inherits the properties from his parents. A similar concept is followed in Java, where we have two classes:
* Parent class (Super or Base class)
* Child class (Subclass or Derived class)
* A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

**Inheritance Syntax:**

class subClass extends superClass

{

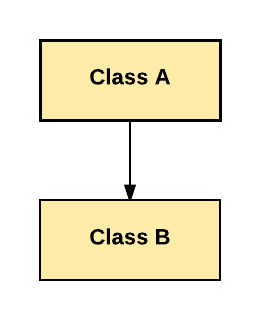
//methods and fields

}

**Types of Inheritance**

### Single Inheritance:

In Single Inheritance one class extends another class (one class only).



In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child extends Parent {

void Child\_Details() {

System.out.println("Child Detail...");

}

}

public class SingleInheritance {

public static void main(String args[])

{

Child c = new Child();

c.Parent\_Details();

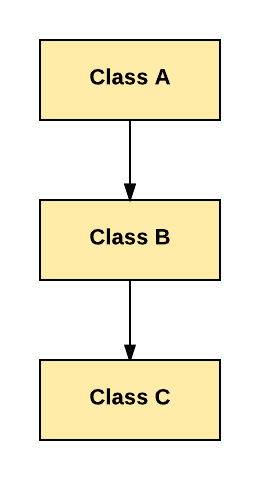
c.Child\_Details();

}

}

### Multilevel Inheritance:

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.



As per shown in diagram Class C is subclass of B and B is a of subclass Class A

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child1 extends Parent {

void Child1\_Details()

{

System.out.println("Child1 Detail...");

}

}

class Child2 extends Child1 {

void Child2\_Details() {

System.out.println("Child2 Detail...");

}

}

public class MultilevelInheritance

{

public static void main(String args[]) {

Child2 c = new Child2();

c.Parent\_Details();

c.Child1\_Details();

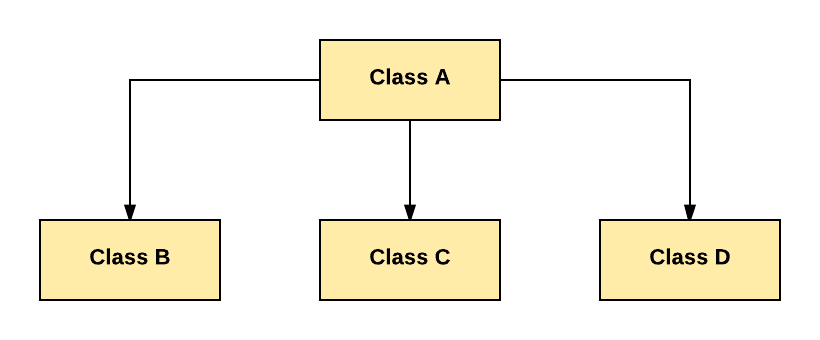
c.Child2\_Details();

}

}

### Hierarchical Inheritance:

In Hierarchical Inheritance, one class is inherited by many sub classes.



As per above example, Class B, C, and D inherit the same class A.

**Example**

class Parent{

void Parent\_Details() {

System.out.println("Parent Details...");

}

}

class Child1 extends Parent {

void Child1\_Details() {

System.out.println("Child1 Detail...");

}

}

class Child2 extends Parent {

void Child2\_Details() {

System.out.println("Child2 Detail...");

}

}

public class HierarichialInheritance

{

public static void main(String args[])

{

Child1 c = new Child1();

Child2 c1 = new Child2();

c.Parent\_Details();

c.Child1\_Details();

c1.Parent\_Details();

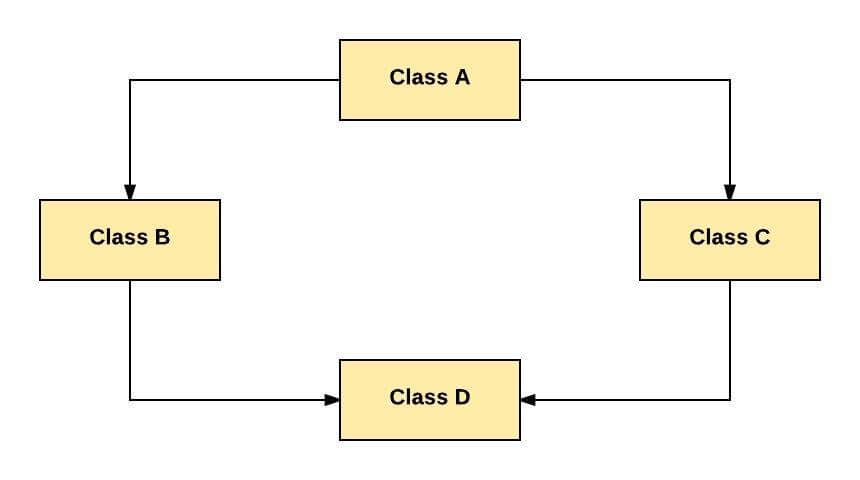
c1.Child2\_Details();

}

}

### Hybrid Inheritance:

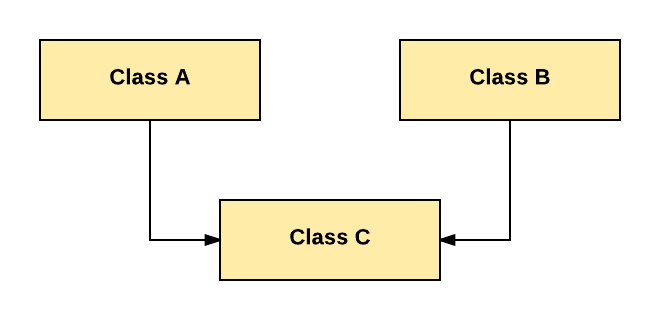
Hybrid inheritance is one of the inheritance types in Java which is a combination of Hierarchical and Multiple inheritance.



As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

### Multiple Inheritance:

Multiple Inheritance is one of the inheritances in Java types where one class extending more than one class. Java does not support multiple inheritance.



As per above diagram, Class C extends Class A and Class B both.

**Note:** Java doesn’t support hybrid/Multiple inheritance because it results in data ambiguity problem.

### Abstraction

* **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.

### Ways to achieve Abstraction

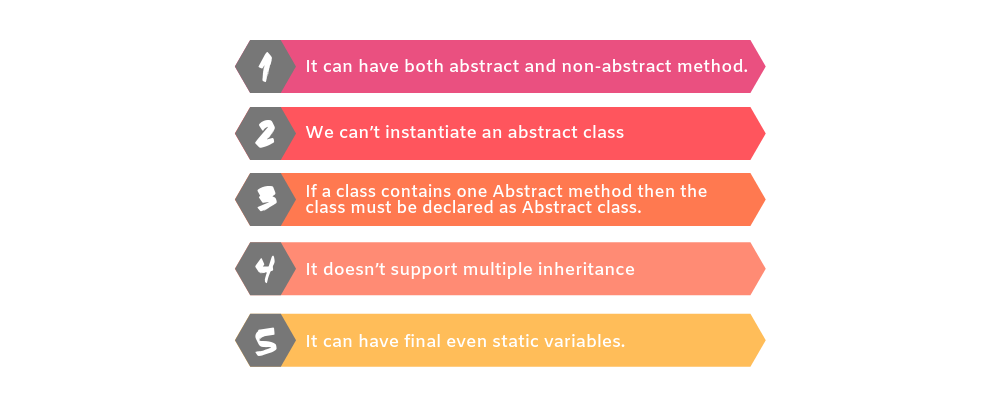
There are two ways to achieve abstraction in java

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.

**Rules for abstract class**



**Example**

**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

**abstract** **class** Birds

{

**abstract** **void** fly();

}

**class** Eagles **extends** Birds

{

**void** fly()

{

System.out.println("Eagles are high flyers");

}

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

{

  Birds b = **new** Eagles();  //upcasting in Java, a subclass object can be referred to by a reference variable of its superclass. This is known as "upcasting."

  b.fly();

}

}

### Abstract class having constructor, data member and methods

### Example

**abstract** **class** Birds

{

Birds() **// Default Constructor**

{

System.out.println(“Birds is created”);

}

**abstract** **void** fly(); **//Abstract Method**

**void** birdquality() **//Non-abstract Method**

**{**

System.out.println(“The eagles are symbol of beauty, bravery, courage, honour, pride, determination, and grace. “);

**}**

}

**class** Eagles **extends** Birds

{

**void** fly()

{

System.out.println("Eagles are high flyers");

}

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

{

  Birds b = **new** Eagles();

  b.fly();

b.birdquality();

}

}

# Interface

* An **interface** is a blueprint of a class. It has static constants and abstract methods.
* The interface in Java is a mechanism to achieve [abstraction](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).
* Java Interface also **represents the IS-A relationship.**
* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have **default and static methods** in an interface.
* Since Java 9, we can have **private methods** in an interface.

**Reasons to use interface**

* 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.

}

**Note:**

* Interface fields are public, static and final by default, and the methods are public and abstract.



#### Relationship between classes and interfaces

**Example**

**interface** Shape

{

**void** draw();

}

**class** Square **implements** Shape{

**public** **void** draw()

{

System.out.println("Draw Square");

}

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

{

Square obj = **new** Square();

obj.draw();

 }

}

**Multiple inheritance in Java by interface**

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



**Example**

**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();

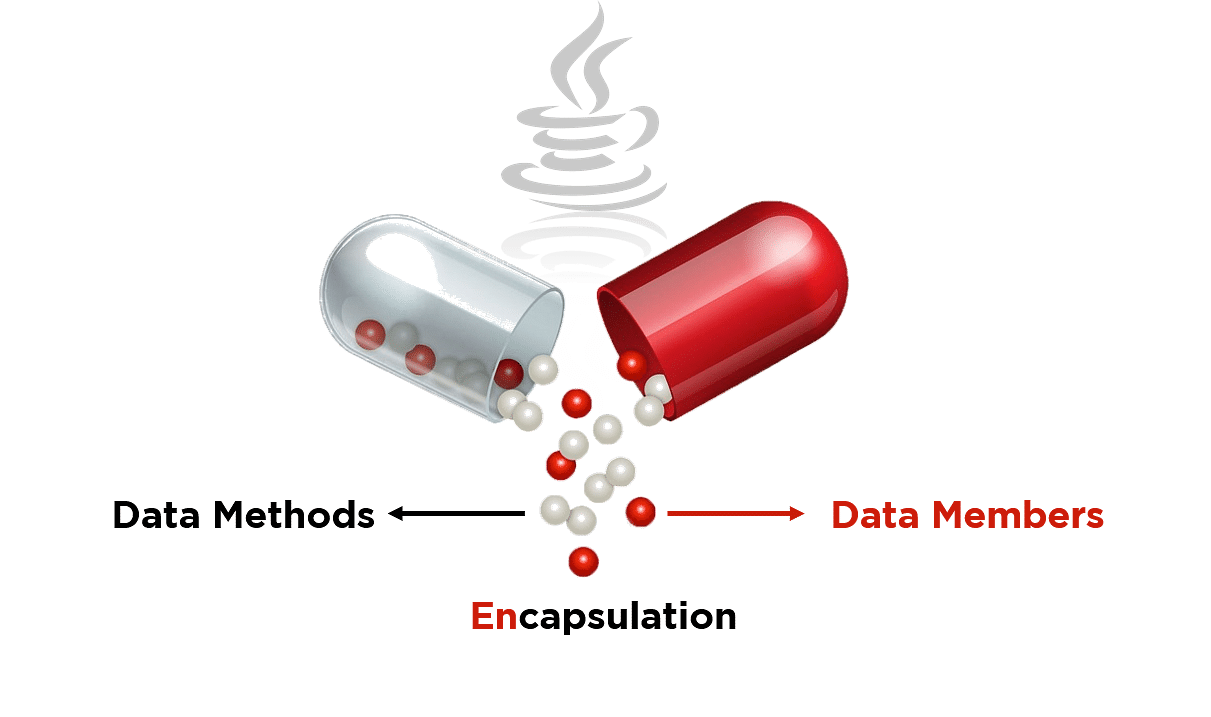
obj.show();

 }

}

**Encapsulation**

* **Encapsulation** is a process of wrapping code and data together into a single unit*,* for example, a capsule which is mixed of several medicines.
* In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found
* We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
* The **Java Bean** class is the example of a fully encapsulated class.



**Syntax:**

<Access\_Modifier> class <Class\_Name>

{

 private <Data\_Members>;

 private <Data\_Methods>;

}

**Need for Encapsulation**

* Better Control
* Setter and Getter
* Security
* Flexibility

**Example**

**Student.java**

**public** **class** Student

{

**//private data member**

**private** String name;

**//getter method for name**

**public** String getName()

{

**return** name;

}

**//setter method for name**

**public** **void** setName(String name1)

{

Name1=name

}

}

**Test.java**

**class** Test

{

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

{

**//creating instance of the encapsulated class**

Student s=**new** Student();

**//setting value in the name member**

s.setName("vijay");

**//getting value of the name member**

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

}

}

# Polymorphism

**Polymorphism** is a concept by which we can perform a **single action in different ways**. Polymorphism is derived from 2 Greek words: **poly and morphs**. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms.

**Two types of polymorphism**

* Compile-time polymorphism
* Runtime polymorphism.

We can perform polymorphism in java by method overloading and method overriding.

**Runtime Polymorphism (Dynamic Polymorphism)**

* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
* In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

### Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



**Syntax**

**class** A{}

**class** B **extends** A{}

A a=**new** B ();//upcasting

**Runtime Polymorphism with Method**

**Example**

**class** Fruits

{

**void** flavour()

{

System.out.println("Sweet/Sour/Bitter");}

}

**class** Apple **extends** Fruits{

**void** flavour()

{

System.out.println("Flavour is sweet");

}

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

{

    Fruits f = **new** Apple(); //upcasting

    f.flavour();

  }

}

**Compile time Polymorphism(or Static Polymorphism)**

* Polymorphism that is resolved during compiler time is known as static polymorphism.
* Method overloading is an example of compile time polymorphism.

**Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

**Example**

class SampleAdd

{

int add(int a, int b)

{

return a+b;

}

int add(int a, int b,int c)

{

return a+b+c;

}

}

class Demo

{

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

{

  SampleAdd obj=**new** SampleAdd();

  System.out.println(obj.add(10,20));

System.out.println(obj.add(10,20,30));

}

}

**Keywords**

# this keyword

* this is a **reference variable** that refers to the current object.



**Uses of ‘this’ keyword**

* It can be used to refer instance variable of current class
* It can be used to invoke or initiate current class constructor
* It can be passed as an argument in the method call
* It can be used to return the current class instance

**Example**

class Student

{

int rollno;

String name;

Student(int rollno,String name)

{

this.rollno=rollno;

this.name=name;

}

void display()

{

System.out.println(rollno+" "+name+" ");

}

}

class Test

{

public static void main(String args[])

{

Student s1=new Student(1,”James”);

Student s2=new Student(2,”Alex”);

s1.display();

s2.display();

}

}

# Super Keyword in Java

* The **super** keyword 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

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

**Example to invoke super class variables,methods and constructor**

**class** Animal

{

String color="white";

Animal()

{

System.out.println("animal is created");

}

**void** eat()

{

System.out.println("eating...");

}

}

**class** Dog **extends** Animal

{

String color="black";

Dog()

{

**super**();

System.out.println("dog is created");

}

**void** printColor()

{

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

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

}

**void** eat()

{

System.out.println("Dog eating bread...");

**super.**eat();

}

}

**class** TestSuper1

{

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

{

Dog d=**new** Dog();

d.printColor();

d.eat();

}

}

# 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
2. method
3. class

**Final variable**

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

### Example

**class** Car{

**final** **int** speedlimit=100;//final variable

**void** run(){

  speedlimit=400;

 }

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

 Car obj=**new**  Car();

 obj.run();

 }

}

**Final method**

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

### Example

**class** Car

{

**final** **void** run()

{

System.out.println("running");

}

}

**class** Hundai **extends** Car

{

**void** run()

{

System.out.println("running safely with 100kmph");

}

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

{

    Hundai h= **new** Hundai();

    h.run();

   }

}

**Final class**

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

### Example

**final** **class** Car{}

**class** Hundai **extends** Car

{

**void** run()

{

System.out.println("running safely with 100kmph");

}

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

{

   Hundai h= **new** Hundai();

   h.run();

  }

}

**Typecasting**

* In Java, type casting, also known as type conversion, is the process of changing the data type of a value or an object from one type to another.
* Type casting is essential when you need to perform operations on values or objects of different data types or when you want to assign a value of one data type to a variable of another data type.
* Two main types of type casting in Java:
  1. **Primitive Type Casting:**
  + This involves converting primitive data types from one type to another.
  + Primitive data types include **int**, **double**, **char**, **boolean**, etc.
  + Casting can be implicit (automatic) or explicit (manual).

**Example of implicit casting (widening):**

int myInt = 5;

double myDouble = myInt; // Implicit casting (widening)

**Example of explicit casting (narrowing):**

double myDouble = 3.14;

int myInt = (int) myDouble; // Explicit casting (narrowing)

**Reference Type Casting:**

* This involves converting references to objects from one class type to another.
* It is used in situations where there's an inheritance hierarchy (subclasses and superclasses) or when working with interfaces.
* Casting can be upcasting (implicit) or downcasting (explicit).

**Example of upcasting (implicit):**

class Vehicle { }

class Car extends Vehicle { }

Car car = new Car();

Vehicle vehicle = car; // Upcasting (implicit)

**Example of downcasting (explicit):**

class Vehicle { }

class Car extends Vehicle { }

Vehicle vehicle = new Car();

Car car = (Car) vehicle; // Downcasting (explicit)

# String

* String is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. String class implements Serializable, Comparable, CharSequence.

**Example:**

**char**[] ch={'j','a','v','a','p','o','i','n','t'};

String s=**new** String(ch);

String s=**new** String(“Javapoint”);

is same as:

**String** s="javapoint";

**CharSequence Interface**

The CharSequence interface is used to represent the sequence of characters.



* The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.
* For mutable strings, you can use StringBuffer and StringBuilder classes.
* String is an object that represents a sequence of characters. The java. lang. String class is used to create a string object.

### How to create a string object?

There are two ways to create String object:

* By string literal
* By new keyword

### 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 the string doesn't exist in the pool, a new string instance is created and placed in the pool.
* For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance



**Example Explanation**

Only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object.

After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

**Uses**

* To make more memory efficiently

### new keyword

**Example**

String s=**new** String("Welcome”);

**Note:**

While using new keyword [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) 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 a heap (non-pool).

**Example**

**public** **class** StringExample

{

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

{

String s1="java";// Java string literal

**char** ch[]={'s','t','r','i','n','g','s'};

String s2=**new** String(ch);//converting char array to string String s3 String s3=**new** String("example");// Java string by new keyword.

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}

}

* **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

**Example String Methods**

**public** **class** Stringmethods {

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

{ //concatination

String s1="hello";

String s2=" its beautiful";

System.**out**.println(s1.concat(" world"));

System.**out**.println(s1+s2);

System.**out**.println(s1);

//substring

String s4="java is simple";

System.**out**.println(s4.substring(2));

System.**out**.println(s4.substring(0, 1));

System.**out**.println(s4.substring(0, 2));

System.**out**.println(s4.substring(0, 3));

System.**out**.println(s4.substring(1, 7));

//upper and lower case

String s5="hello";

String s6="Hello";

System.**out**.println(s5.toUpperCase());

System.**out**.println(s6.toLowerCase());

//trim()

String s7=" hello ";

System.**out**.println(s7.trim());

System.**out**.println(s7);

//starts and ends with

String s8=" hello ";

String s9="hello";

String s10="Hello";

System.**out**.println(s8.startsWith("he"));

System.**out**.println(s9.startsWith("he"));

System.**out**.println(s10.endsWith("io"));

//char at

String s11=" morning ";

String s12="Hello";

System.**out**.println(s11.charAt(4));

System.**out**.println(s12.charAt(1));

//length()

String s13=" hello ";

String s14="hello";

System.**out**.println(s13.length());

System.**out**.println(s14.length());

//value of

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

System.**out**.println(a+b);

String s=String.valueOf(a);

System.**out**.println(s+10);

//replace of

String s15="Java is a programming language." + "Java is a platform. Java is an island.";

String rep=s15.replace("Java", "c");

System.**out**.println(rep);

//replace of

String s16="c";

String r=s16.replace("c", "h");

System.**out**.println(r);

//System.out.println(s1); string is immutable

}

}

# String compare

* Compare String in Java on the basis of content and reference.
* It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

Three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

**equals() Method**

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

* **public boolean equals(Object another)** compares this string to the specified object.
* **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

**Example**

**class** Teststringcomparison1{

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

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   String s4="Saurav";

String s5="Java";

  String s6="JAVA";

   System.out.println(s1.equals(s2));//true

   System.out.println(s1.equals(s3));//true

   System.out.println(s1.equals(s4));//false

System.out.println(s5.equals(s6));//false

   System.out.println(s5.equalsIgnoreCase(s6));//true

 }

}

**Using == operator**

* The == operator compares references not values.

**Example**

**class** Teststringcomparison3

{

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

{

   String s1="Sachin";

   String s2="Sachin";

   String s3=**new** String("Sachin");

   System.out.println(s1==s2);//true (because both refer to same instance)

   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)

 }

}

**Using compareTo() method**

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

* **s1 == s2** : The method returns 0.
* **s1 > s2** : The method returns a positive value.
* **s1 < s2** : The method returns a negative value.

**Example**

**class** Teststringcomparison4{

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

   String s1="Java";

   String s2="Java";

   String s3="Programming";

   System.out.println(s1.compareTo(s2));//0

   System.out.println(s1.compareTo(s3));//1(because s1>s3)

   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

 }

}

# StringBuffer Class

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

### Constructors of StringBuffer Class

* **StringBuffer()** : It creates an empty String buffer with the initial capacity of 16.
* **StringBuffer(String str):** It creates a String buffer with the specified string.
* **StringBuffer(int capacity):** It creates an empty String buffer with the specified capacity as length.

**Example**

**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);

sb.insert(1,"Java");

System.out.println(sb);

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

System.out.println(sb);

sb.delete(1,3);

System.out.println(sb);

sb.reverse();

System.out.println(sb);

}

}

# 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.

### Constructors of StringBuilder Class

* **StringBuilder ()**: It creates an empty String builder with the initial capacity of 16.
* **StringBuilder (String str):** It creates a String builder with the specified string.
* **StringBuilder (int length):** It creates an empty String buffer with the specified capacity as length.

**Example**

**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);

sb.insert(1,"Java");

System.out.println(sb);

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

System.out.println(sb);

sb.delete(1,3);

System.out.println(sb);

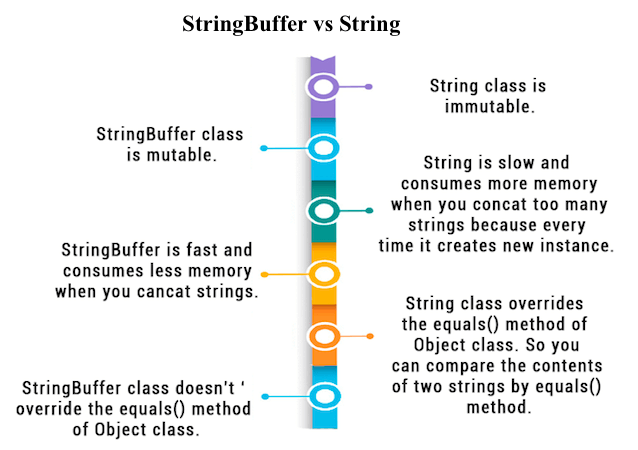
sb.reverse();

System.out.println(sb);

}

}

# Difference between String and StringBuffer



# Difference between StringBuffer and StringBuilder

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously | StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |
| StringBuffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |

# Arrays

**Array** is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



### Advantages

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### Disadvantages

* **Size Limit:** We can store only the 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 which grows automatically.

### Types of Array

* Single Dimensional Array
* Multidimensional Array

**Single Dimensional Array in Java**

**Syntax**

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

**Instantiation of an Array**

arrayRefVar=**new** datatype[size];

### Example

**class** Testarray{

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

**int** a[]=**new** **int**[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//traversing array

**for**(**int** i=0;i<a.length;i++) //length is the property of array

{

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

}

}

}

## For-each Loop for Java Array

We can also print the array using [for-each loop](https://www.javatpoint.com/for-each-loop). The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop:

**for**(data\_type variable:array){

//body of the loop  }

**Example**

**class** Testarray1

{

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

{

**int** arr[]={33,3,4,5};

//printing array using for-each loop

**for**(**int** i:arr)

System.out.println(i);

}

}

**Passing Array to a Method**

We can pass the java array to method so that we can reuse the same logic on any array.

**Example**

**class** Testarray2

{

//creating a method which receives an array as a parameter

**static** **void** minValue(**int** arr[]){

**int** min=arr[0];

**for**(**int** i=1;i<arr.length;i++)

**{**

**if**(min>arr[i])

   min=arr[i];

 }

System.out.println(min);

}

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

{

**int** a[]={33,3,4,5};//declaring and initializing an array

minValue(a);//passing array to method

}

}

**Returning Array from the Method**

We can also return an array from the method in Java.

**class** TestReturnArray

{

//creating method which returns an array

**static** **int**[] getValue()

{

Int c[]={1,2,3,4,5}

Int c[]=new int[]{10,20,30,40};

Return c;

**return** **new** **int**[]{10,30,50,90,60};

}

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

{

//calling method which returns an array

**int** arr[]=getValue();

//printing the values of an array

**for**(**int** i=0;i<arr.length;i++)

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

}

}

**Multidimensional Array**

The data is stored in row and column based index (also known as matrix form).

**Syntax**

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

**Instantiate Multidimensional Array**

**int**[][] arr=**new** **int**[3][3];

### Example

**class** Testarray3{

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

{

//declaring and initializing 2D array

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

//printing 2D array

**for**(**int** i=0;i<3;i++){

**for**(**int** j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

**Jagged Array**

 Jagged Array is an array of arrays with different number of columns.

**Example**

**class** TestJaggedArray

{

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

{

        //declaring a 2D array with odd columns

**int** arr[][] = **new** **int**[3][];

         arr[0] = **new** **int**[3];

         arr[1] = **new** **int**[4];

         arr[2] = **new** **int**[2];

        //initializing a jagged array

**int** count = 0;

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

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

                 arr[i][j] = count++;

        //printing the data of a jagged array

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

{

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

{

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

       }

            System.out.println();//new line

        }

    }

}

# Wrapper classes in Java

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Wrapper classes are also used to provide a variety of utility functions for primitives data types like converting primitive types to string objects and vice-versa, converting to various bases like binary, octal or [hexadecimal](https://en.wikipedia.org/wiki/Hexadecimal), or comparing various objects.

We cannot provide null values to Primitive types but wrapper classes can be null. So wrapper classes can be used in such cases we want to assign a null value to primitive data types.

The different scenarios, where we need to use the wrapper classes:

* **Serialization**
* **Synchronization**
* **java.util package**
* **Collection Framework**

# 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?**

### By making a reference null

Employee e=new Employee();

e=null;

### By assigning a reference to another:

Employee e1=new Employee();

Employee e2=new Employee();

e1=e2;

### By anonymous object:

new Employee();

Anonymous Object that don’t have any reference So gc their memory once they have serve their purpose.

**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() throws Throwable{}

**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(){}

The garbage collector is a part of Java Virtual Machine(JVM). Garbage collector checks the heap memory, where all the objects are stored by JVM, looking for unreferenced objects that are no more needed. And automatically destroys those objects. Garbage collector calls finalize() method for clean up activity before destroying the object. Java does garbage collection automatically; there is no need to do it explicitly, unlike other programming languages.

The garbage collector in Java can be called explicitly using the following method:

System.gc()

**Exception**

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

### Advantage

The core advantage of exception handling is **to maintain the normal flow of the application**.

**Let's consider a scenario:**

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

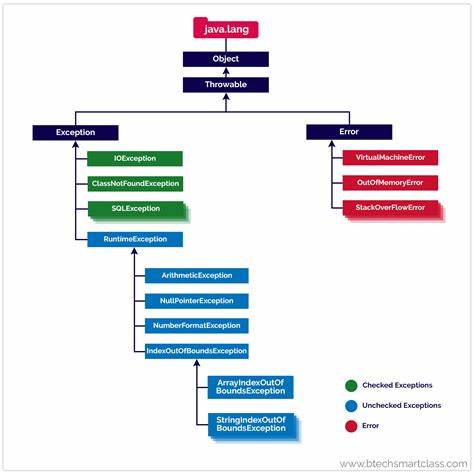
statement 9;

statement 10;

Explanation of scenario: Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

**Hierarchy of Java Exception classes**

The **java.lang.Throwable** class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error.



### Types of Java Exceptions

1. Checked Exception
2. Unchecked Exception
3. Error

## Difference between Checked and Unchecked Exceptions

### 1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

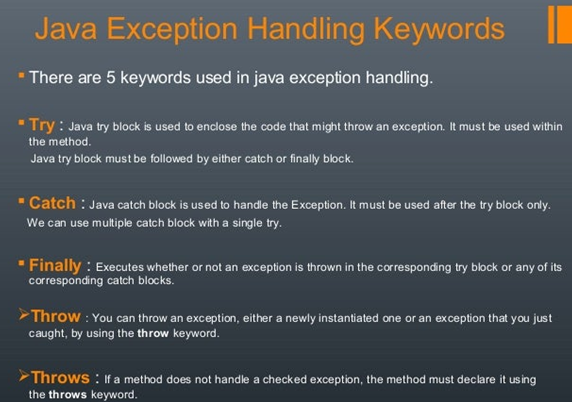
### 2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

Error is irrecoverable. Some examples of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

**Java Exception Keywords**



### Syntax of Java try-catch

**try**{

//code that may throw an exception

}**catch**(Exception\_class\_Name ref){}

### Syntax of try-finally block

**try**{

//code that may throw an exception

}**finally**{}

**Problem without exception handling**

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

**Example**

public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

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

    }

}

**Example using try-catch statements**

**ArithmeticException**

**public** **class** TryCatchExample

{

**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...");

  }

### NullPointerException

**public** **class** TryCatchExample1

{

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

{

**try**

{

       //code that may raise exception

      String s=null;

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

    }

**catch**(NullPointerException e)

{

System.out.println(e);

}

   //rest code of the program

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

  }

### NumberFormatException

**public** **class** TryCatchExample1

{

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

{

**try**

{

       //code that may raise exception

      String s="abc";

**int** i=Integer.parseInt(s);

    }

**catch**(NumberFormatException e)

{

System.out.println(e);

}

   //rest code of the program

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

  }

### ArrayIndexOutOfBoundsException

**public** **class** TryCatchExample1

{

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

{

**try**

{

       //code that may raise exception

      int a[]=new int[5];

a[10]=50;

    }

**catch**(ArrayIndexOutOfBoundsException e)

{

System.out.println(e);

}

   //rest code of the program

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

  }

# Java Catch Multiple Exceptions

**Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Notes:

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e., catch for Arithmetic Exception must come before catch for Exception.

### Flowchart of Multi-catch Block



### Example

**public** **class** MultipleCatchBlock1 {

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

**try**{

**int** a[]=**new** **int**[5];

                a[5]=30/0;

               }

**catch**(ArithmeticException e)

                  {

                   System.out.println("Arithmetic Exception occurs");

                  }

**catch**(ArrayIndexOutOfBoundsException e)

                  {

                   System.out.println("ArrayIndexOutOfBounds Exception occurs");

                  }

**catch**(Exception e)

                  {

                   System.out.println("Parent Exception occurs");

                  }

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

    }

}

# Java Nested try block

* In Java, using a try block inside another try block is permitted. It is called as nested try block.
* Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

**Situations where we use nested try block**

* 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 catch block within another try block

**try**

    {

        statement 3;

        statement 4;

//try catch block within nested try block

**try**

        {

            statement 5;

            statement 6;

     }

**catch**(Exception e2)

        {

//exception message

        }

    }

**catch**(Exception e1)

    {

//exception message

    }

}

//catch block of parent (outer) try block

**catch**(Exception e3)

{

//exception message

}

....

### Example

**public** **class** NestedTryBlock{

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

 //outer try block

**try**{

   //inner try block 1

**try**{

      System.out.println("going to divide by 0");

**int** b =39/0;

    }

     //catch block of inner try block 1

**catch**(ArithmeticException e)

    {

      System.out.println(e);

    }

    //inner try block 2

**try**{

**int** a[]=**new** **int**[5];

     //assigning the value out of array bounds

     a[5]=4;

     }

    //catch block of inner try block 2

**catch**(ArrayIndexOutOfBoundsException e)

    {

       System.out.println(e);

    }

   System.out.println("other statement");

  }

  //catch block of outer try block

**catch**(Exception e)

  {

    System.out.println("handled the exception (outer catch)");

  }

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

 }

}

# Java finally block

* **Java finally block** is a block used to execute important code such as closing the connection, etc.
* Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

### Flowchart of finally block



**Why use Java finally block?**

* Finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

**Usage of Java finally**

### Case 1: When an exception does not occur

### If the Java program does not throw any exception, and the finally block is executed after the try block.

**class** TestFinallyBlock {

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

**try**{

//below code do not throw any exception

**int** data=25/5;

   System.out.println(data);

  }

//catch won't be executed

**catch**(NullPointerException e){

System.out.println(e);

}

//executed regardless of exception occurred or not

**finally** {

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

}

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

  }

}

### Case 2: When an exception occur but not handled by the catch block

**public** **class** TestFinallyBlock1{

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

**try** {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //cannot handle Arithmetic type exception

      //can only accept Null Pointer type exception

**catch**(NullPointerException e){

        System.out.println(e);

      }

      //executes regardless of exception occured or not

**finally** {

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

      }

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

      }

    }

### Case 3: When an exception occurs and is handled by the catch block

**public** **class** TestFinallyBlock2{

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

**try** {

        System.out.println("Inside try block");

        //below code throws divide by zero exception

**int** data=25/0;

       System.out.println(data);

      }

      //handles the Arithmetic Exception / Divide by zero exception

**catch**(ArithmeticException e){

        System.out.println("Exception handled");

        System.out.println(e);

      }

      //executes regardless of exception occurred or not

**finally** {

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

      }

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

      }

    }

# throw Exception

* The Java throw keyword is used to throw an exception explicitly.
* We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description.
* We can throw either checked or unchecked exceptions in Java by throw keyword.
* It is mainly used to throw a custom exception.
* We can also define our own set of conditions and throw an exception explicitly using throw keyword.

**Syntax**

**throw** **new** exception\_class("error message");

**Example**

**public** **class** TestThrow1 {

    //function to check if person is eligible to vote or not

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

**if**(age<18) {

            //throw Arithmetic exception if not eligible to vote

**throw** **new** ArithmeticException("Person is not eligible to vote");

        }

**else** {

            System.out.println("Person is eligible to vote!!");

        }

    }

    //main method

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

        //calling the function

        validate(13);

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

  }

}

### Example 3: Throwing User-defined Exception

**class** UserDefinedException **extends** Exception

{

**public** UserDefinedException(String str)

    {

        // Calling constructor of parent Exception

**super**(str);

    }

}

// Class that uses above MyException

**public** **class** TestThrow3

{

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

    {

**try**

        {

            // throw an object of user defined exception

**throw** **new** UserDefinedException("This is user-defined exception");

        }

**catch** (UserDefinedException ude)

        {

            System.out.println("Caught the exception");

            // Print the message from MyException object

            System.out.println(ude.getMessage());

        }

    }

}

# throws keyword

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

### Syntax

return\_type method\_name() **throws** exception\_class\_name{

//method code

}

**Note:**

### Which exception should be declared?

**Ans:** Checked exception only, because:

* **unchecked exception:** under our control so we can correct our code.
* **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### Advantage of Java throws keyword

* Now Checked Exception can be propagated (forwarded in call stack).
* It provides information to the caller of the method about the exception.

**Example**

**Exception Chaining/Exception Propagation using throws keywords**

**import** java.io.IOException;

**class** Testthrows1{

**void** m()**throws** IOException{

**throw** **new** IOException("device error");//checked exception

  }

**void** n()**throws** IOException{

    m();

  }

**void** p(){

**try**{

    n();

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

  }

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

   Testthrows1 obj=**new** Testthrows1();

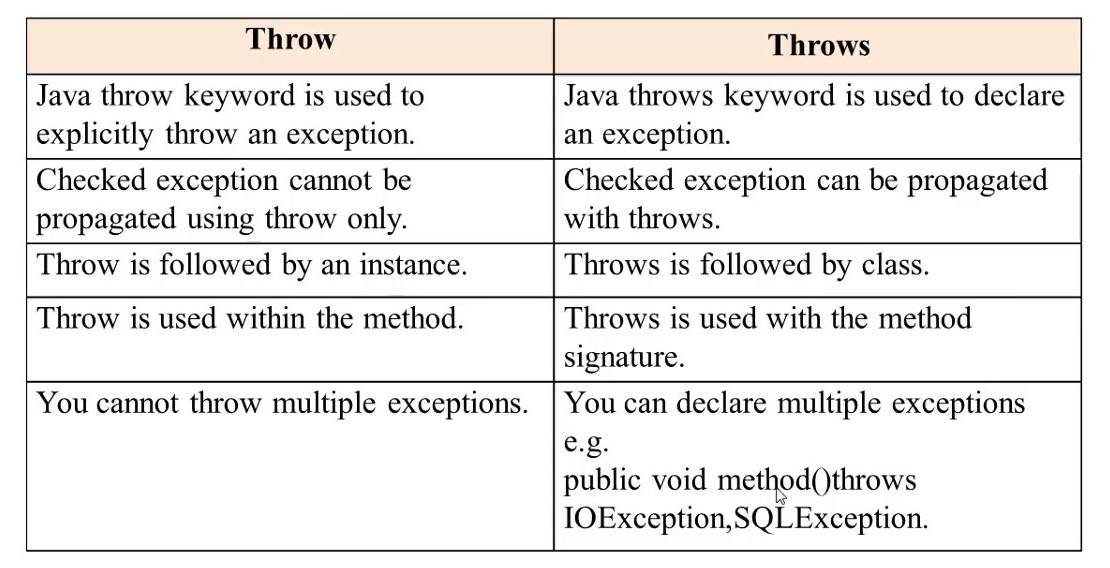
   obj.p();

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

  }

}

# Difference between throw and throws in Java



**Multithreading**

**Multithreading**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.

**Advantages**

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](https://www.javatpoint.com/register-memory), 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.

**Thread**

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](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

# Life cycle of a Thread

states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

**New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

**Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

* **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.  
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
* **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

**Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

**Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

**Terminated:** A thread reaches the termination state because of the following reasons:

* When a thread has finished its job, then it exists or terminates normally.
* **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



# How to create a thread in Java

There are two ways to create a thread:

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

### Thread class:

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

**Constructors of Thread class:**

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

**Methods of Thread class:**

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

### Runnable interface:

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

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

### Starting a thread:

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

### Java Thread Example by extending Thread class

**class** Multi **extends** Thread{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi t1=**new** Multi();  //new state

t1.start();  //active state

 }

}

### Java Thread Example by implementing Runnable interface

**class** Multi3 **implements** Runnable{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi3 m1=**new** Multi3();

Thread t1 =**new** Thread(m1);   // Using the constructor Thread(Runnable r)

t1.start();

 }

}

# Thread.sleep()

### The sleep() Method Syntax:

**public** **static** **void** sleep(**long** mls) **throws** InterruptedException

**public** **static** **void** sleep(**long** mls, **int** n) **throws** InterruptedException

**Example**

**class** TestSleepMethod1 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<5;i++){

  // the thread will sleep for the 500 milli seconds

**try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }

 }

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

  TestSleepMethod1 t1=**new** TestSleepMethod1();

  TestSleepMethod1 t2=**new** TestSleepMethod1();

  t1.start();

  t2.start();

 }

}

# Synchronization in Java

* Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
* Java Synchronization is better option where we want to allow only one thread to access the shared resource.

**The synchronization is mainly used to**

* To prevent thread interference.
* To prevent consistency problem.

**Thread Synchronization**

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. **Mutual Exclusive**
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. **Cooperation (Inter-thread communication in java)**

### Concept of Lock in Java

Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them

### Thread Non-Synchronization

**class** Table{

**void** printTable(**int** n){//method not synchronized

**for**(**int** i=1;i<=5;i++){

     System.out.println(n\*i);

**try**{

      Thread.sleep(400);

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

   }

  }

}

**class** MyThread1 **extends** Thread{

Table t;

MyThread1(Table t){

**this**.t=t;

}

**public** **void** run(){

t.printTable(5);

}

}

**class** MyThread2 **extends** Thread{

Table t;

MyThread2(Table t){

**this**.t=t;

}

**public** **void** run(){

t.printTable(100);

}

}

**class** TestSynchronization1{

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

Table obj = **new** Table();//only one object

MyThread1 t1=**new** MyThread1(obj);

MyThread2 t2=**new** MyThread2(obj);

t1.start();

t2.start();

}

}

**Thread Synchronization**

### Java Synchronized Method

* If you declare any method as synchronized, it is known as synchronized method.
* Synchronized method is used to lock an object for any shared resource.
* When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

**class** Table{

**synchronized** **void** printTable(**int** n){//synchronized method

**for**(**int** i=1;i<=5;i++){

     System.out.println(n\*i);

**try**{

      Thread.sleep(400);

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

   }

  }

}

**class** MyThread1 **extends** Thread{

Table t;

MyThread1(Table t){

**this**.t=t;

}

**public** **void** run(){

t.printTable(5);

}

 }

**class** MyThread2 **extends** Thread{

Table t;

MyThread2(Table t){

**this**.t=t;

}

**public** **void** run(){

t.printTable(100);

}

}

**public** **class** TestSynchronization2{

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

Table obj = **new** Table();//only one object

MyThread1 t1=**new** MyThread1(obj);

MyThread2 t2=**new** MyThread2(obj);

t1.start();

t2.start();

}

}

# Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

### notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax:**

public final void notify()

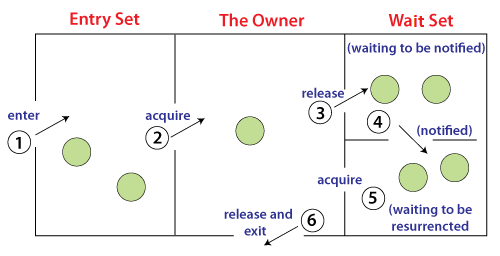
### notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

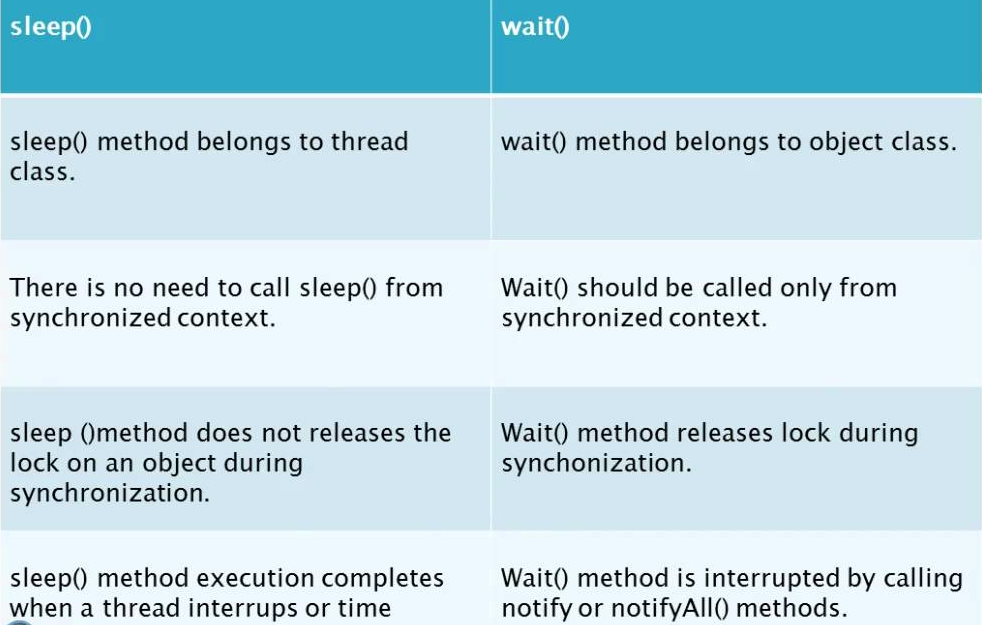
public final void notifyAll()

## Understanding the process of inter-thread communication



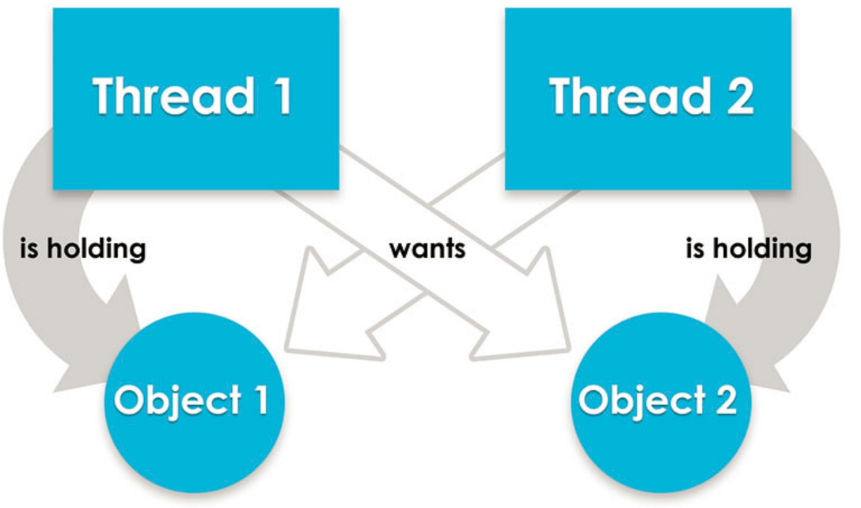
1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### Difference between wait and sleep?



**Deadlock in Java**

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



**Collections**

* Collections are used to group objects into single entity. Once the objects are stored in a collection you can then perform operations on single object or multiple objects (bulk operations).
* You can insert, update, delete, and retrieve objects into entity. Also, we can query the entity for size and other things.

Let us first understand what Java Collections are.

**Java collections are made up of following:**

**Interfaces**

* **Definition:** Interfaces in Java collections are abstract data types that represent different types of collections, such as lists, sets, queues, and maps.
* **Purpose:** These interfaces define a contract or set of methods that a collection class must implement. They provide a standardized way of interacting with various types of collections.
* **Example:** The **List** interface represents an ordered collection of elements, and classes like **ArrayList** and **LinkedList** implement this interface.

**Implementations**

* **Definition:** Implementations are classes that provide concrete implementations of the collection interfaces. They can be abstract or concrete classes.
* **Purpose:** These classes adhere to the contract specified by the interfaces and provide ready-to-use, efficient, and reusable data structures. Developers can choose the appropriate implementation based on their specific needs.
* **Example:** The **ArrayList** class is an implementation of the **List** interface, providing a dynamic array that can grow or shrink as needed.

**Algorithms**

* **Definition:** Algorithms in Java collections refer to the core functionalities provided by the framework for manipulating and processing elements within collections.
* **Purpose:** These algorithms include operations like searching, sorting, shuffling, frequency counting, and more. They are designed to work seamlessly with different collection types and are highly optimized for performance.
* **Example:** The **Collections** class in Java provides static methods for various algorithms, such as **sort()** for sorting a list and **binarySearch()** for searching a sorted list.

**Benefits of Java Collections:**

1. **Standardization:**
   * Provides a standardized set of interfaces and classes for common data structures.
2. **Reusability:**
   * Offers reusable data structures and algorithms, reducing development time and effort.
3. **Interoperability:**
   * Allows interchangeability of different implementations through common interfaces.
4. **Flexibility:**
   * Provides a diverse range of interfaces and implementations to suit various programming needs.
5. **Concurrency Support:**
   * Offers concurrent collection classes for safe use in multithreaded environments.
6. **Algorithms:**
   * Includes well-tuned algorithms for common operations like sorting and searching.
7. **Integration with Other APIs:**
   * Seamlessly integrates with other parts of the Java API, enhancing overall code readability.

**Hierarchy of Collection Framework**

Let us see the hierarchy of Collection framework. The **java.util** package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.

Top of Form

## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

**It contains only one abstract method. i.e.,**

Iterator<T> iterator()

It returns the iterator over the elements of type T.

## Collection Interface

* The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have.
* In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
* Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

* List <data-type> list1= **new** ArrayList();
* List <data-type> list2 = **new** LinkedList();
* List <data-type> list3 = **new** Vector();
* List <data-type> list4 = **new** Stack();

##### **ArrayList**

ArrayList is the implementation class of List Interface which is used to store a group of individual objects where duplicate values are allowed. ArrayList internally follows array structure, which means in ArrayList all the elements are stored in contiguous memory locations same as an array, but ArrayList size is not fixed.

ArrayList is not a synchronized class. If any object is synchronized we can access only one thread at a time but if an object is not. synchronized then we can access multiple threads.

**package** Collections;

**import** java.util.\*;

**public** **class** ArrayList1 {

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

// **TODO** Auto-generated method stub

List<Integer> li=**new** ArrayList<>();

li.add(10);

li.add(20);

li.add(30);

li.add(40);

li.add(50);

**for**(**int** i=0;i<li.size();i++)

{

System.***out***.println("List elements are:"+li.get(i));

}

li.add(2, 1000);

li.set(0, 500);

li.remove(0);

**for**(Integer i:li)

{

System.***out***.println("List elements using foreach are:"+i);

}

li.remove(Integer.*valueOf*(30));

Iterator<Integer> it=li.iterator();

**while**(it.hasNext())

{

System.***out***.println("List elements using Iterator are:"+it.next());

}

List<String> list=**new** ArrayList<>();

list.add("Anu");

list.add("Priya");

list.add("Tarun");

list.add("Kiran");

list.add("Hari");

list.remove("Tarun");

System.***out***.println(list);

System.***out***.println(list.contains("Sindhu"));

}

}

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

**import** java.util.\*;

**public** **class** TestJavaCollection2{

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

LinkedList<String> al=**new** LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

## Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

**package** Collections;

**import** java.util.\*;

**public** **class** Stack1 {

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

// **TODO** Auto-generated method stub

Stack<String> fruits=**new** Stack<>();

fruits.push("Apple");

fruits.push("Banana");

fruits.push("Mango");

fruits.push("Kiwi");

fruits.push("Grapes");

fruits.push("Pomogranate");

System.***out***.println(fruits);

System.***out***.println(fruits.peek());

System.***out***.println(fruits.pop());

System.***out***.println(fruits.peek());

}

}

## Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

**Queue interface can be instantiated as:**

* Queue<String> q1 = **new** PriorityQueue();
* Queue<String> q2 = **new** ArrayDeque();
* Queue<String> q2 = **new** LinkedList();

## PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

**package** Data;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** PriorityQueue1 {

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

// **TODO** Auto-generated method stub

PriorityQueue<Integer>pq=**new** PriorityQueue<>(Comparator.*reverseOrder*());

pq.offer(100);

pq.offer(20);

pq.offer(140);

pq.offer(60);

pq.offer(180);

pq.offer(10);

System.***out***.println(pq);

System.***out***.println(pq.peek());

System.***out***.println(pq.poll());

System.***out***.println(pq);

}

}

**LinkedList**

package Data;

import java.util.LinkedList;

import java.util.Queue;

public class Queue1 {

public static void main(String[] args) {

// TODO Auto-generated method stub

Queue<Integer> queue=new LinkedList<>();

queue.offer(100);

queue.offer(120);

queue.offer(140);

queue.offer(160);

queue.offer(180);

queue.offer(200);

System.out.println(queue);

System.out.println(queue.peek());

System.out.println(queue);

System.out.println(queue.poll());

System.out.println(queue);

// we can also use add(),element(),remove() but this methods will throw exception if the element is performed is specified task/operations.

}

}

## ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends. ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

**package** Data;

**import** java.util.ArrayDeque;

**public** **class** ArrayDeque1 {

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

// **TODO** Auto-generated method stub

ArrayDeque<Integer> adq=**new** ArrayDeque<>();

adq.offer(50);

adq.offerFirst(20);

adq.offerLast(5);

adq.offer(30);

System.***out***.println(adq);

System.***out***.println(adq.peek());

System.***out***.println(adq.peekFirst());

System.***out***.println(adq.peekLast());

System.***out***.println(adq);

System.***out***.println(adq.poll());

System.***out***.println(adq.pollFirst());

System.***out***.println(adq.pollLast());

System.***out***.println(adq);

}

}

## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

* Set<data-type> s1 = **new** HashSet<data-type>();
* Set<data-type> s2 = **new** LinkedHashSet<data-type>();
* Set<data-type> s3 = **new** TreeSet<data-type>();

**package** Data;

**import** java.util.\*;

**public** **class** Set1 {

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

// **TODO** Auto-generated method stub

Set<Integer> s1=**new** HashSet<Integer>();

s1.add(20);

s1.add(30);

s1.add(80);

s1.add(10);

s1.add(50);

System.***out***.println(s1);

s1.add(50);

s1.remove(10);

System.***out***.println(s1);

System.***out***.println(s1.isEmpty());

System.***out***.println(s1.size());

System.***out***.println(s1.contains(80));

s1.clear();

System.***out***.println(s1);

}

}

**package** Data;

**import** java.util.HashSet;

**import** java.util.LinkedHashSet;

**import** java.util.Set;

**public** **class** LinkedHashSet1 {

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

// **TODO** Auto-generated method stub

Set<Integer> s1=**new** LinkedHashSet<Integer>();

s1.add(20);

s1.add(30);

s1.add(80);

s1.add(10);

s1.add(50);

System.***out***.println(s1);

s1.add(50);

s1.remove(10);

System.***out***.println(s1);

System.***out***.println(s1.isEmpty());

System.***out***.println(s1.size());

System.***out***.println(s1.contains(80));

s1.clear();

System.***out***.println(s1);

}

}

**package** Data;

**import** java.util.Comparator;

**import** java.util.HashSet;

**import** java.util.Set;

**import** java.util.TreeSet;

**public** **class** TreeSet1 {

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

// **TODO** Auto-generated method stub

Set<Integer>s1=**new** TreeSet<Integer>(Comparator.*reverseOrder*());

s1.add(20);

s1.add(30);

s1.add(80);

s1.add(10);

s1.add(50);

System.***out***.println(s1);

s1.add(50);

s1.remove(10);

System.***out***.println(s1);

System.***out***.println(s1.isEmpty());

System.***out***.println(s1.size());

System.***out***.println(s1.contains(80));

s1.clear();

System.***out***.println(s1);

}}

# Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys. A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below: A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

Java Map Hierarchy

* A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

|  |  |
| --- | --- |
| **Class** | **Description** |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

**package** Data;

**import** java.util.\*;

**public** **class** Map1 {

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

// **TODO** Auto-generated method stub

Map<String,Integer> map1=**new** HashMap<>();

map1.put("Kripa", 1);

map1.put("Diya", 2);

map1.put("Alex", 3);

map1.put("Gleena", 4);

map1.put("Tony", 5);

System.***out***.println(map1);

**if**(!map1.containsKey("Diya"))

{

map1.put("Diya", 5);

}

System.***out***.println(map1);

System.***out***.println(map1.putIfAbsent("jiya", 4));

}

}

**LinkedHashMap1**

**import** java.util.\*;

**public** **class** LinkedHashMap1 {

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

// **TODO** Auto-generated method stub

Map<Integer,String> map1=**new** LinkedHashMap<>();

map1.put(1, "Sithara");

map1.put(2,"Keerthi");

map1.put(6, "Vimal");

map1.put(10, "Hari");

map1.put(3, "Leena");

map1.put(4, "Arun");

map1.put(5, "Erin");

map1.put(**null**, "Arun");

map1.put(11, **null**);

System.***out***.println(map1);

System.***out***.println(map1.containsKey(3));

}

}

**TreeMap**

**package** Data;

**import** java.util.\*;

**public** **class** LinkedHashMap1 {

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

// **TODO** Auto-generated method stub

Map<Integer,String> map1=**new** TreeMap<>();

map1.put(1, "Sithara");

map1.put(2,"Keerthi");

map1.put(6, "Vimal");

map1.put(10, "Hari");

map1.put(3, "Leena");

map1.put(4, "Arun");

map1.put(5, "Erin");

System.***out***.println(map1);

System.***out***.println(map1.containsKey(3));

System.***out***.println(map1.containsValue("Suriya"));

**for**(Map.Entry<Integer, String> e:map1.entrySet())

{

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

System.***out***.println(e.getKey());

System.***out***.println(e.getValue());

}

**for**(Integer key:map1.keySet())

{

System.***out***.println(key);

}

**for**(String value:map1.values())

{

System.***out***.println(value);

}

System.***out***.println(map1.containsValue("Leena"));

}

}

**3/11/2022**

**JDBC(Java Database Connectivity)**

* JDBC’s full form is Java Database Connectivity. It is basically a Java API that is used to execute and connect query along with the database.
* It is considered to be part of Java SE, that is, Java Standard Edition. API of JDBC makes uses of drivers of JDBC in order to get connected along with the database.
* We can consider four types of JDBC drivers mentioned as below :
* JDBC-ODBC Bridge Driver
* Thin Driver
* Native Driver and
* Network Protocol Driver

## What is API ?

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

JDBC API can be used in order to access the tabular data stored in any of the relational databases. By using JDBC API, you can save, update, delete, and fetch the data from the database. Therefore, it is considered to be Open Database Connectivity, that is, ODBC provided by Microsoft.

**Architecture Of JDBC**

### Lightbox

### The ****java.sql**** package contains classes and interfaces for JDBC API. A list of popular interfaces of JDBC API are given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface

A list of popular **classes**of JDBC API are given below:

* DriverManager class
* Blob class (Binary Large Object)
* Clob class (Character Large Object)
* Types class

# Java Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection

### 1.Register the driver class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.  **Syntax**  public static void forName(String className)throws ClassNotFoundException 2. Create the connection object  |  | | --- | | The **getConnection()** method of DriverManager class is used to establish connection with the database. |  Syntax **public** **static** Connection getConnection(String url)**throws** SQLException  **public** **static** Connection getConnection(String url,String name,  String password )**throws** SQLException 3.Create the Statement object  |  | | --- | | The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database. |  Syntax **public** Statement createStatement()**throws** SQLException 4.Execute the query  |  | | --- | | The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table. |  Syntax **public** ResultSet executeQuery(String sql)**throws** SQLException 5.Close the connection object  |  | | --- | | By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection. |  Syntax **public** **void** close()**throws** SQLException |

Here is a hint: Calculating the following **measures** could help to define proper KPIs:

* # of men
* # of women
* # of leavers
* % employees promoted (FY21)
* % of women promoted
* % of hires men
* % of hires women
* % turnover
* Average performance rating: men
* Average Performance rating: women