**JAVA NOTES**

**Java =** Is a high level object oriented programming language.

Current version of java is1.18 and the stable version is java 1.8

Program - Decision, plan for something, set of activities are combined together to

perform a specific task.

Program in a static state, when we have to execute the program it will in the process state and processor will do it.

Main method is not required at the time of compilation but it is required at the time

execution.

**JVM helps to execute the byte code**

JVM-IBM, AMD, Intel

JVM-Provide runtime environment to execute byte code Tasks of JVM:

1) Loads the byte code

2) Verifies the byte code

3) Execute the byte code

JDK- Java Development Kit (JRE+JVM (SET OF LIBRARIES)) JRE- to execute the byte code or to run the java code JDK- to write (develop) and execute the java code.

JRE-Java Runtime Environment is a combination of JVM and set of libraries(+development tools (compiler, debugger, java docs, etc..)) .JRE is physically exist but JVM is not exist physically

JVM contains the set of libraries to execute the byte code (JVM is available in the back end)

JDK and JRE all are platform dependent only byte (JVM) code is platform independent.

**Features of java are:**

Platform independent Open source

Secure

Object oriented

Simple

**Array** = It is container which stores a group of similar type of data of variables.

To store multiple values which are of similar type in a single variable or single reference is called as array variable.

**Synatx**: datatype variableName[] = {data};

or Datatype[] variableName = {data};

Ex: String kannadMovies[] = {"KGF", "Om", "googly"}; or String [] variableName = {data};

**Variables** = is registered area allocated in the memory.

**Type of Variables** = static variables, local variables, instance/ non static variables, parameter variables.

**Static variable:** The variables are declared outside the main but inside the class is declared as static called static variable.

Ex: static int age = 22;

static boolean is Alive ; // it will give the output value default as false

**Local variable:** Any variable which is declared inside the main or inside constructor or inside block are considered to be as local variable.

Ex: int

age = 22;

In this declaration and initialization are mandatory boolen isAlive = false;

**Non-static or instance:** Variables are declared outside the main and inside the class but cannot be declared as a static.

Declaration is nothing but commanding the variables and in array format the declaration and intilization in a single line or in single variable

ex : String name ;

Intilization is to asine the particular

ex : String name = “Boss”;

|  |  |
| --- | --- |
| Instance variables | Static (class) variables |
| Instance variables are declared in a class, but outside a method, constructor or any block. | Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. |
| Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed. | Static variables are created when the program starts and destroyed when the program stops. |
| Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. ObjectReference*.*VariableName. | Static variables can be accessed by calling with the class name ClassName*.*VariableName. |
| Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class. | There would only be one copy of each class variable per class, regardless of how many objects are created from it. |
| Instance is considered as object member , it gets load into memory while creation of object . | Static is considered as class member , it gets load into memory while creation of class . |
| At the time of object creation instance members gets loaded into the memory | At the time of class creation static members gets loaded into the memory |
| Instance members get loaded into the memory after static members gets stored. | Static members get loaded into the memory first |
| Instance members cannot be shared among different objects | Static members can be shared among different |
| Instance member can be stored multipal times | Static member cannot be stored multipal times |

**For loop:**

for (declaration and intilization of a variable; condition with relation; increment /decrement) {

System.out.println( meats[g] + " ");

}

for (int z = 0 ; z < names.length ; z++){

}

relation = < , > , <= , >= , == , != .

output of relation is true or false

In order to find the size of array - use length property

Length = (“ the size of an array is “ + variable.length);

length can only be used in array veriable only.

**if statements synatx:**

if(condition) //it always check the condition with relational operators only // if true, execute current block

}

**Data types :**

Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| **boolean** | **false** | **1 bit** |
| **char** | **' ' / space** | **2 byte** |
| **byte** | **0** | **1 byte** |
| **short** | **0** | **2 byte** |
| **int** | **0** | **4 byte** |
| **long** | **0L** | **8 byte** |
| **float** | **0.0f** | **4 byte** |
| **double** | **0.0d** | **8 byte** |

**non-premitive data types are used for declaring class and objects.**

implicit and explicit in array

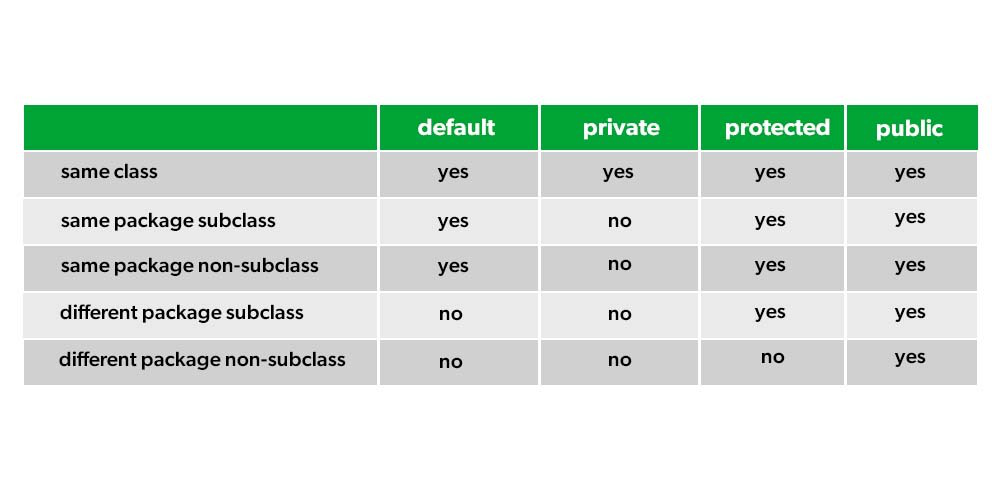
Public = access modifier /acces spicifer.

static = non access modifier.

return type = void, primitive and non primitive data type.

**There are four types of Java access modifiers:**

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.
3. **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.
4. **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.



**method** = it is a collection of statement that are grouped together to perform operation / functionality

any name associated with parasintasis () is called method.

method is written in such type = access-modfier return type methodName (parameter){

body of the method // block of code// block

|  |  |
| --- | --- |
| Compile-time | Runtime |
| The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler. | The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time. |
| In this case, the compiler prevents the code from execution if it detects an error in the program. | In this case, the compiler does not detect the error, so it cannot prevent the code from the execution. |
| It contains the syntax and semantic errors such as missing semicolon at the end of the statement. | It contains the errors such as division by zero, determining the square root of a negative number. |
| It runs class files. | It convert the class file into object and run. |

parameter are anything written in (0), only declaration

.

IDE = Integrated Development Environment/ Equipment . Ex eclips and Intellij Idea

method header

name of the method and parameter is known as method signature. Ex add (int a, int b).

Method Syntax

method name

parameters (formal parameter)

public static void add (int a , int b)

{ methodheader

Body of the method

}

To achieve any condition we should use any relational operation

if (condition){

// if true, execute

}

components in simple class

class object

variables methods states behavoiurs

static instance

java can understand only two thing that is class at compilation and object at runtime in jvm.

Type of methods is static and instatc.

Return value should always match with the return type of the method

**Method Overloading** = method declaring more than once within a same class with a same name but different in the parameter is considered to be as method overloading .

Advantages of method – code reusability.

3 reason – change in the type of parameter

change in the number of parameter

interchange the type of parameter

**Method Overriding**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

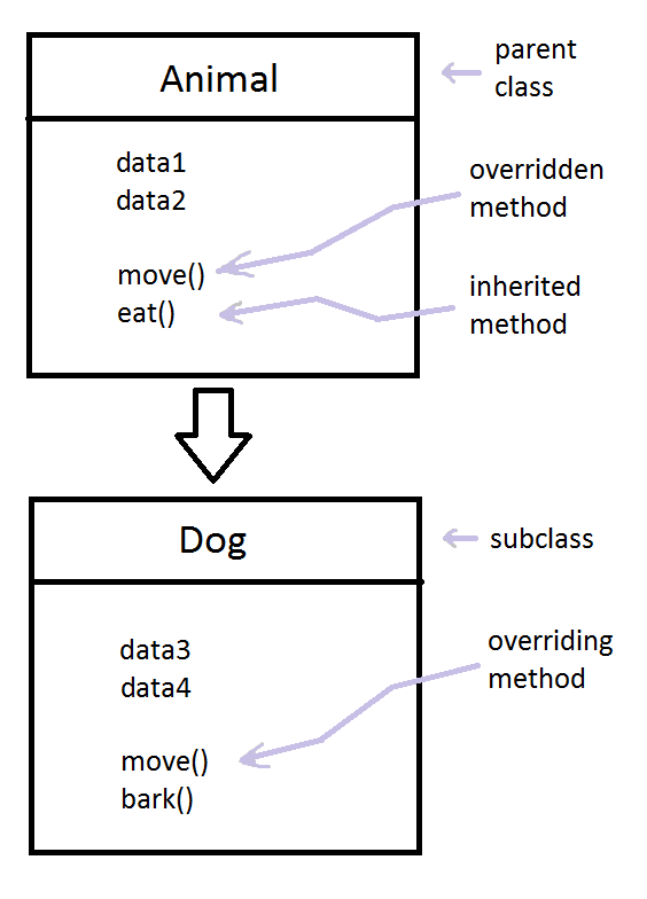
In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass/parentclass.
* Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).



**Setter and Getter :**

A getter is a method that reads the value of a variable.

A setter is a method that updates the value of a variable.

**class** = Is a design or blueprint which contains states and behaviors.

**States** are represented as variables.

**Behaviors** are represented as methods.

**Object** = is an instance of class, purpose of object is to create a multiple copy of a class.

It is also called as root class.

There are 3 ways to initialize object in Java.

1. By reference variable (Using new key world)
2. By method (Class.newInstance() method , Object.clone() method)
3. By constructor (newInstance() method of constructor )

### Methods declared in class java.lang.[Object](https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html)

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

**Object class**

The **Object class** is the parent class of all the classes in java by default. In other words, it is the

top most class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice

that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type

like Employee,Student etc, we can use Object class reference to refer that object.

For example:

Object obj=getObject();//we don't know what object will be returned from this method

|  |  |
| --- | --- |
| **Object** | **Class** |
| Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. |
| Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair. | Class is a **group of similar objects**. |
| Object is a **physical** entity. | Class is a **logical** entity. |
| Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| Object is created **many times** as per requirement. | Class is declared **once**. |
| Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |
| It takes more space in the memory compared to class. | It takes less space in the memory |

**Constructer** = it a special type of method which will be same name as class name with parasintasis and with no returntype , it use when creating object.

use of constructer is to initialization instance variable of class.

**Type of constructer are:**

**Default Constructor** - If we do not create any constructor, the Java compiler automatically a constructor during the execution of the program. This constructor is called default constructor, if there is no parameter passed in the constructer is called default constructer or if there no constructer typed in the code the java compiler will automatically create constructer. The use of it is to declare default values.

**Ex** . ClassName(){

} = Default Constructer

**Parameterized Constructor** - A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors.

Ex. ClassName (int a, int b) = Parameter Constructer

**this()** – it can only be used in a constructor and not any wear else, its is know as constructor chaining, u can communicate for one constructor to other.

**this**. – is key world can be used any wear inside the class.

//use of constructor

//constructor overloading

//constructor overloading this () method

ex:

calss Soap {

Soap() {

}

}

static members (variables, methods) are called as Class members, where as instance/non-static members (variables, methods) are called as object members.

**Eclips :**

Firstly you have to creat project and then creat package

Objet and constructer

class ClassName {

String variableName;

// default Constructer public classname (){

}

default constructer and parameter constructer

what is block

A block is a set of coad. This is done by enclosing the statements between opening and closing curly braces.

**Package :**

Package is a folder.

Package is a collection of classes and interfaces, they are nothing but a folder which help us to reusability for better use of project.

There are two types of package in java one is use define and other is built in packages

* built in package always starts with java
* user defined always starts with com/org

java.lang package is called default package

import package+classname = FQCN / qualified name.

import statement is used when two class are different package.

There are total 14 package in java default package.

**Oops :**

Object Oriented Programming Structure/System.

Pillars or Principals of Oops are

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation

**Inheritance** is a mechanism in which one object acquires all the properties and behaviors of a parent object.

Inheritance is known as code reusability, extends. Inheritance in java is also called as is a relation.

Inheritance is acquiring properties from one class to another class that is from parent to child class/ super to sub class/ Base to derived class.

We achieve inheritance using key world extends.

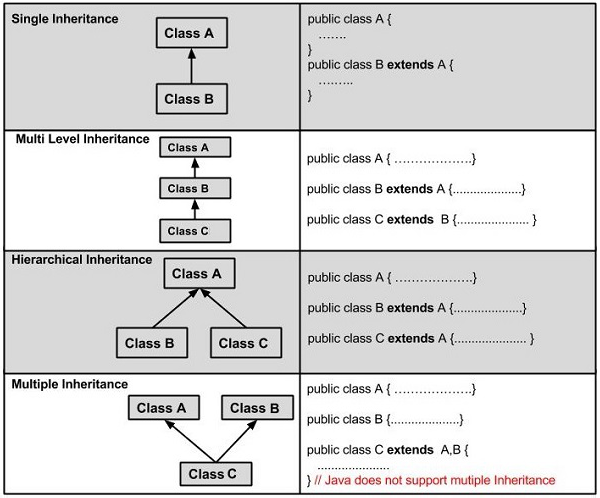
**Types of inheritance in java**

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.







Multiple inheritance is not supported in Java through class.

In multiple inheritance we have one child class and two parent class the child class will not able to recognize the parent class and this process is called **ambiguity .**

super() is called only when we extends the parent class

this() is called only when we use constructor within the same class

Difference Between this and super keyword

|  |  |
| --- | --- |
| this | super |
| The current instance of the class is represented by this keyword. | The current instance of the parent class is represented by the super keyword. |
| In order to call the default constructor of the current class, we can use this keyword. | In order to call the default constructor of the parent class, we can use the super keyword. |
| It can be referred to from a static context. It means it can be invoked from the static context. | It can't be referred to from a static context. It means it cannot be invoked from a static context. |
| We can use it to access only the current class data members and member functions. | We can use it to access the data members and member functions of the parent class. |

Difference Between this() and super() method constructor

|  |  |
| --- | --- |
| this() | super() |
| The this() constructor refers to the current class object. | The super() constructor refers immediate parent class object. |
| It is used for invoking the current class method. | It is used for invoking parent class methods. |
| It can be used anywhere in the parameterized constructor. | It is always the first line in the child class constructor. |
| It is used for invoking a super-class version of an overridden method. | It is used for invoking a super-class version of an overridden method. |

**is a** relationship occurs only if two class which are of same type.

ex = mango is a fruit , earth is a planate.

**has a** relationship occurs only if two class have different type.

ex = programme has Bugs , Exam has HallTicket , Hospital has Patients

**Enum**

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma.

ex

public enum gender {

male , female , others

}

output : gender.male

**Scanner**

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.

**Wrapper classes**

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

**Use of Wrapper classes in Java**

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.
* **AutoBoxing :** Convert primitives into objects and objects into primitives automatically.

//Java program to convert primitive into objects

//Autoboxing example of int to Integer

**public** **class** WrapperExample1{

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

//Converting int into Integer

**int** a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}}

Output:

20 20 20

* **Unboxing:** Converting object into primitives automatically.

//Java program to convert object into primitives

//Unboxing example of Integer to int

**public** **class** WrapperExample2{

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

//Converting Integer to int

Integer a=**new** Integer(3);

**int** i=a.intValue();//converting Integer to int explicitly

**int** j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);

}}

Output:

3 3 3

|  |  |
| --- | --- |
| Primitive Type | Wrapper class |
| boolean | [**Boolean**](https://www.javatpoint.com/java-boolean) |
| char | [**Character**](https://www.javatpoint.com/post/java-character) |
| byte | [**Byte**](https://www.javatpoint.com/java-byte) |
| short | [**Short**](https://www.javatpoint.com/java-short) |
| int | [**Integer**](https://www.javatpoint.com/java-integer) |
| long | [**Long**](https://www.javatpoint.com/java-long) |
| float | [**Float**](https://www.javatpoint.com/java-float) |
| double | [**Double**](https://www.javatpoint.com/java-double) |

**Parent :** Object class is the parent class in Java. All classes in Java directly or indirectly inherit the Object class. Inheritance is an object-oriented concept in which one class uses the properties and behavior of another class.

**Child Class :**

**The class which inherits the properties of other** is known as child class (derived class, sub class) and the class whose properties are inherited is known as parent class (base class, superclass class).

**Polymorphism** – an object behaving differently at a give instance of time.

Two type of polymorphism

1. **Compile time polymorphism** - It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

**Method Overloading** = method declaring more than once within a same class with a same name but different in the parameter is considered to be as method overloading .

Advantages of method – code reusability.

3 reason – change in the type of parameter

change in the number of parameter

interchange the type of parameter

**Example –**

**class** OverloadingCalculation1{

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

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

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

  OverloadingCalculation1 obj=**new** OverloadingCalculation1();

  obj.sum(20,20);//now second int literal will be promoted to long

  obj.sum(20,20,20);

  }

}

1. **Run-time polymorphism** - It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Method Overriding**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass/parentclass.
* Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

Example –

//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

**class** Bank{

**int** getRateOfInterest(){**return** 0;}

}

//Creating child classes.

**class** SBI **extends** Bank{

**int** getRateOfInterest(){**return** 8;}

}

**class** ICICI **extends** Bank{

**int** getRateOfInterest(){**return** 7;}

}

**class** AXIS **extends** Bank{

**int** getRateOfInterest(){**return** 9;}

}

//Test class to create objects and call the methods

**class** Test2{

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

SBI s=**new** SBI();

ICICI i=**new** ICICI();

AXIS a=**new** AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

Static are called as class member so those class cannot be shared to the child class.

converting from parent to child ---do it explicitly

ex: Engineer engineer = new Engineer()

**Polymorphism/ implicit casting**

ex: Engineet engineer2 = new CivilEngineer();

**Explicit casting**

ex : MechEngineer mechEng = (MechEngineer ) eng;

There are two types of casting,

* Primitive Type Casting
* Reference Type Casting

**Primitive Type Casting**

Casting between primitive types enables you to convert the value of one type to another primitive type is called Primitive Type Casting. This is most commonly occurs with the **numeric data types** . But boolean primitive type can never be used in a cast. Its values must be either true or false and cannot be used in a casting operation.

**Encapsulation –**

Group of data enclosing into a single Block is called Encapsulation.

Is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

**Setter and Getter :**

A getter is a method that reads the value of a variable.

A setter is a method that updates the value of a variable.

**Example:**

**class 1:**

//A Account class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.

**class** Account {

//private data members

**private** **long** acc\_no;

**private** String name,email;

**private** **float** amount;

//public getter and setter methods

**public** **long** getAcc\_no() {

**return** acc\_no;

}

**public** **void** setAcc\_no(**long** acc\_no) {

**this**.acc\_no = acc\_no;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **float** getAmount() {

**return** amount;

}

**public** **void** setAmount(**float** amount) {

**this**.amount = amount;

}

}

**class 2:**

//A Java class to test the encapsulated class Account.

**public** **class** TestEncapsulation {

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

//creating instance of Account class

Account acc=**new** Account();

//setting values through setter methods

acc.setAcc\_no(7560504000L);

acc.setName("Sonoo Jaiswal");

acc.setEmail("sonoojaiswal@javatpoint.com");

acc.setAmount(500000f);

//getting values through getter methods

System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());

}

}

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0

**Abstraction method –** are the method which do not have body

**Interface we cannot create object.**

**Implementation class**

**Abstraction -**

Hiding the implementation and allowing the user to access only the functionality with the help of interface.

There are two ways to achieve abstraction in java

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

**Abstract class in Java**

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

Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Interface in Java**

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

The interface in Java is *a mechanism to achieve*[*abstraction*](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).

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

There are mainly three 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.

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

Since Java 9, we can have **private methods** in an interface



// Interface

interface Animal {

public void animalSound(); // interface method (does not have a body)

public void sleep(); // interface method (does not have a body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

public void sleep() {

// The body of sleep() is provided here

System.out.println("Zzz");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Two type of coupling**

loos coupling – change in implantation which does not affect the user. It can be achived only with the help of interface.

tight coupling – change in implantation which affects the user.

**Java Memory –**

Statck – (First in last out (FILO), Last in first out (LIFO)).

Heap – are classified into

* 1. Young Generation - is subdivided in three areas
     1. Eden.
     2. Survivor memory 1.
     3. Survivor memory 2.

All objects newly created will be stored in young generation in young generation Eden memory.

When eden memory is filled full without using the object it calls garbage collection in garbage minor GC from eden memory the objects move to Survivor memory 1 and 2.

* 1. Old Generation.

When all the

When old generation is full filled with object JVM calls garbage collector in garbage collector it calls major GC.

**Major GC** is harmfull it make application slow.

In old Generation is fully filled with objects

* 1. Permanent Generation-

is memory which stores class information , static , methods , blocks , enum

Old Generation

**Heap**

**Young Generation**

**Permanent Generation-**

**Class area / method area**

**Old Generation**

**Stack – FILO , LIFO**

Eden

**Survivor memory 1**

**SCP-String Constant Poll.**

Store String Data.

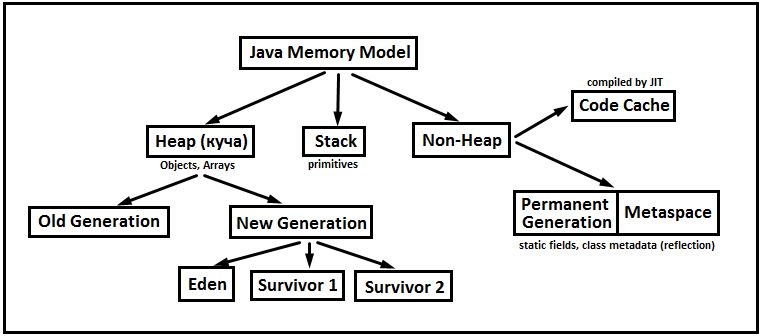
Only string literals will be stored.

* Name of Method
* Return Type
* Parameter value
* Current line info
* Next line into

After all the process it gets pop out one by one and all the frames gets exited

**Survivor memory 2**

**Stack** – The main purpose of stack is to execute methods.It execute methods one after the other by creating fram. In Stack the first method is going to be stored in bottom fram. It stores name of the method , return type , parameters, current line information.



String and Wrapper class have special options in java, they can be create object in two different ways such as

Creating string objects using string literals

String var = “ ”;

Creating string object using new keyword

String var1 = new String (“ “);

In Wrapper class:

Integer i2 = 90 ; // no error would be found because it is as wrapper class.

Reference variable does not address object directly instead it pointing to memory address of Heap from Stack

Reference variable object stored in Stack, but its object stored in heap.

String created using new Keyword the reference will be stored in heap but outside the SCP.

After changing the value of String the new object will be created and the reference will point towards to new object

Immutability:- Cannot change the contents created in memory;

|  |  |
| --- | --- |
| **Double equals(==)** | **.equals( )** |
| Whenever we use double equals it will compare memory address of the object  Used for address comparison | When we use **.equals( )** it will compare the content of an object  Used for value comparison |

**String built-in methods:**

|  |
| --- |
| **1.charAt( ) :** The charAt() method returns the character at the specified index in a string. |
| Syntax: public char charAt (int *index*) |
| Return: char |

|  |
| --- |
| **2.compareTo ( ):** The compareTo() method compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. |
| The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters). |
| Syntax: public int compareTo(String string2) |
| Return: int |

|  |
| --- |
| **3.contains ( ) :** The contains() method checks whether a string contains a sequence of characters. |
| Returns true if the characters exist and false if not. |
| Syntax: public boolean contains(CharSequence *chars*) |
| Return: boolean |

|  |
| --- |
| **4.equals( ) :** The equals() method compares two strings, and returns true if the strings are equal, and false if not. |
| Syntax**:** public boolean equals(String anotherString) |
| Return:Boolean |

|  |
| --- |
| **5.isEmpty():** The isEmpty() method checks whether a string is empty or not.This method returns true if the string is empty ([length()](https://www.w3schools.com/java/ref_string_length.asp) is 0), and false if not. |
| Syntax : public boolean isEmpty() |
| Return: boolean |

|  |
| --- |
| **6.replaceAll():** Replaces each substring of this string that matches the given regular expression with the given replacement. |
| Return: String |

|  |
| --- |
| **7. split():** Splits a string into an array of substrings |
| Return: String[] |

|  |
| --- |
| **8. substring():** Returns a new string which is the substring of a specified string |
| Return: String |

|  |
| --- |
| **9. toCharArray() :** Converts this string to a new character array. |
| Return : char[] |

|  |
| --- |
| **10.toLowerCase ():** The toLowerCase() method converts a string to lower case letters. |

|  |
| --- |
| **11. toUpperCase():** The toUpperCase() method converts a string to upper case letters. |

|  |
| --- |
| **12. valueOf():** Returns the string representation of the specified value. |
| Return: String. |

|  |
| --- |
| **13.concat():** The concat() method appends (concatenate) a string to the end of another string. |
| Syntax: public String concat(String string2) |

**String Buffer** they can be created only using new key world, they are mutable nature.

Stored in Heap outside SCP, Buffer is slower than Builder. Buffer method works one after the other.

ex :

StringBuffer buffer = new StringBuffer( “ ” )

**Important Constructors of String Buffer Class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| 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. |

**Important methods of StringBuffer class**

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | It is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | It is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | It is used to return the character at the specified position. |
| public int | length() | It is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

**String Building**- they can be created only using new key world, they are muitable in nature. Builder method works at a time,

**Important Constructors of StringBuilder class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| 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 Builder with the specified capacity as length. |

**Important methods of StringBuilder class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public StringBuilder replace(int startIndex, int endIndex, String str) | It is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | It is used to reverse the string. |
| public int capacity() | It is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | It is used to return the character at the specified position. |
| public int length() | It is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

**Collections**

That provides an architecture to store and manipulate the group of objects.

Defaluts of arrays :

1. The size of the array is fixed.
2. In array, methods cannot be called.
3. Array can only be called using index.

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.

Methods of Collection interface

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

Hierarchy of Collection

Iterable (I)

Collection(I)

Queue

ArrayList

SortedSet

Set

List

HashSet

LinkedList

TreeSet

LinkedList

LinkedHashSet

toString() Method :

If you want to represent any object as a string, **toString() method** comes into existence.

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

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

Advantage of Java toString() method

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

example using defaut toString() :

**class** Student{

**int** rollno;

String name;

String city;

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

**this**.rollno=rollno;

**this**.name=name;

**this**.city=city;

}

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

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

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

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

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

}

}

**Output:**

Student@1fee6fc

Student@1eed786

example user overriding toString():

**class** Student{

**int** rollno;

String name;

String city;

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

**this**.rollno=rollno;

**this**.name=name;

**this**.city=city;

}

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

**return** rollno+" "+name+" "+city;

}

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

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

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

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

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

}

}

**Output:**

101 Raj lucknow

102 Vijay ghaziabad

**forEach:**

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

public class MetroCities {

public static void main(String[] args) {

String name1 = "Bangalore";

String name2 = "Mumbai";

String name3 = "Kolkata";

String name4 = "Hyderabad";

String name5 = "Chennai";

Collection<String> citie = new ArrayList<String>();

citie.add(name5);

citie.add(name1);

citie.add(name3);

citie.add(name2);

citie.add(name4);

System.out.println("Using foreach loop");

for (String city : citie) {

System.out.println(city);

}

**Iterator**:

An Iterator is an object that can be used to loop through collections, like [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) and [HashSet](https://www.w3schools.com/java/java_hashset.asp). It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

It use at the time of modifying the data we use Iterator method, like removing or adding the data.

The iterator() method can be used to get an Iterator for any collection:

// Import the ArrayList class and the Iterator class

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator <String> it = cars.iterator();

while(it.hasNext()){

String string = it.next();

System.out.println(string);

}

}

}

Removing Items from a Collection

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(12);

numbers.add(8);

numbers.add(2);

numbers.add(23);

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

while(it.hasNext()) {

Integer i = it.next();

if(i < 10) {

it.remove();

}

}

System.out.println(numbers);

}

}

Collections can be iterated easily using two approaches.

* **Using for-Each loop** − Use a foreach loop and access the array using object.
* **Using Iterator** − Use a Iterator method access the array using object , U can modify using Iterator method like removing or adding the data.

Differences

* **ConcurrentModificationException** − Using for-Each loop, if an object is modified, then ConcurrentModificationException can occur. Using iterator, this problem is elliminated.
* **Size Check** − Using for-Each, size check is not required. Using iterator if hasNext() is not used properly, NoSuchElementException can occur.
* **Performance** − Performance is similar for both cases.

|  |  |  |
| --- | --- | --- |
| **S.No.** | **== Operator** | **Equals() Method** |
| 1. | == is considered an operator in Java. | Equals() is considered as a method in Java. |
| 2. | It is majorly used to compare the reference values and objects. | It is used to compare the actual content of the object. |
| 3. | We can use the == operator with objects and primitives. | We cannot use the equals method with primitives. |
| 4. | The == operator can’t compare conflicting objects, so at that time the compiler surrenders the compile-time error. | The equals() method can compare conflicting objects utilizing the equals() method and returns “false”. |
| 5. | == operator cannot be overridden. | Equals() method and can be overridden. |

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(E e) | This method inserts the specified element into the list. |
| boolean hasNext() | This method returns true if the list iterator has more elements while traversing the list in the forward direction. |
| E next() | This method returns the next element in the list and advances the cursor position. |
| int nextIndex() | This method returns the index of the element that would be returned by a subsequent call to next() |
| boolean hasPrevious() | This method returns true if this list iterator has more elements while traversing the list in the reverse direction. |
| E previous() | This method returns the previous element in the list and moves the cursor position backward. |
| E previousIndex() | This method returns the index of the element that would be returned by a subsequent call to previous(). |
| void remove() | This method removes the last element from the list that was returned by next() or previous() methods |
| void set(E e) | This method replaces the last element returned by next() or previous() methods with the specified element. |

How to create List

//Creating a List of type String using ArrayList

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

//Creating a List of type Integer using ArrayList

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

//Creating a List of type Book using ArrayList

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

//Creating a List of type String using LinkedList

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

**Example of List in Java**

**import** java.util.\*;

**public** **class** ListExample1{

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

 //Creating a List

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

 //Adding elements in the List

 list.add("Mango");

 list.add("Apple");

 list.add("Banana");

 list.add("Grapes");

 //Iterating the List element using for-each loop

**for**(String fruit:list)

  System.out.println(fruit);

}

}

**How to convert Array to List**

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

**import** java.util.\*;

**public** **class** ArrayToListExample{

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

//Creating Array

String[] array={"Java","Python","PHP","C++"};

System.out.println("Printing Array: "+Arrays.toString(array));

//Converting Array to List

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

**for**(String lang:array){

list.add(lang);

}

System.out.println("Printing List: "+list);

}

}

**How to convert List to Array**

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

**import** java.util.\*;

**public** **class** ListToArrayExample{

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

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

 fruitList.add("Mango");

 fruitList.add("Banana");

 fruitList.add("Apple");

 fruitList.add("Strawberry");

 //Converting ArrayList to Array

 String[] array = fruitList.toArray(**new** String[fruitList.size()]);

 System.out.println("Printing Array: "+Arrays.toString(array));

 System.out.println("Printing List: "+fruitList);

}

}

**Get and Set Element in List**

The *get() method* returns the element at the given index, whereas the *set() method* changes or replaces the element.

**import** java.util.\*;

**public** **class** ListExample2{

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

 //Creating a List

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

 //Adding elements in the List

 list.add("Mango");

 list.add("Apple");

 list.add("Banana");

 list.add("Grapes");

 //accessing the element

 System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because index starts from 0

 //changing the element

 list.set(1,"Dates");

 //Iterating the List element using for-each loop

**for**(String fruit:list)

  System.out.println(fruit);

 }

}

**How to Sort List**

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

**import** java.util.\*;

**class** SortArrayList{

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

  //Creating a list of fruits

  List<String> list1=**new** ArrayList<String>();

  list1.add("Mango");

  list1.add("Apple");

  list1.add("Banana");

  list1.add("Grapes");

  //Sorting the list

  Collections.sort(list1);

   //Traversing list through the for-each loop

**for**(String fruit:list1)

    System.out.println(fruit);

 System.out.println("Sorting numbers...");

  //Creating a list of numbers

  List<Integer> list2=**new** ArrayList<Integer>();

  list2.add(21);

  list2.add(11);

  list2.add(51);

  list2.add(1);

  //Sorting the list

  Collections.sort(list2);

   //Traversing list through the for-each loop

**for**(Integer number:list2)

    System.out.println(number);

 }

}

**Output:**

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

## Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

### ListIterator Interface declaration

**public** **interface** ListIterator<E> **extends** Iterator<E>

**Example of ListIterator Interface**

**import** java.util.\*;

**public** **class** ListIteratorExample1{

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

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

        al.add("Amit");

        al.add("Vijay");

        al.add("Kumar");

        al.add(1,"Sachin");

        ListIterator<String> itr=al.listIterator();

        System.out.println("Traversing elements in forward direction");

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

        System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());

        }

        System.out.println("Traversing elements in backward direction");

**while**(itr.hasPrevious()){

        System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());

        }

}

}

output

Traversing elements in forward direction

index:0 value:Amit

index:1 value:Sachin

index:2 value:Vijay

index:3 value:Kumar

Traversing elements in backward direction

index:3 value:Kumar

index:2 value:Vijay

index:1 value:Sachin

index:0 value:Amit

**Set in Java**

The **set** is an interface available in the **java.util** package. The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a **collection interface**. The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.

**import** java.util.\*;

**public** **class** setExample{

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

    {

        // creating LinkedHashSet using the Set

        Set<String> data = **new** LinkedHashSet<String>();

        data.add("JavaTpoint");

        data.add("Set");

        data.add("Example");

        data.add("Set");

        System.out.println(data);

    }

}

## Operations on the Set Interface

**import** java.util.\*;

**public** **class** SetOperations

{

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

    {

        Integer[] A = {22, 45,33, 66, 55, 34, 77};

        Integer[] B = {33, 2, 83, 45, 3, 12, 55};

        Set<Integer> set1 = **new** HashSet<Integer>();

        set1.addAll(Arrays.asList(A));

        Set<Integer> set2 = **new** HashSet<Integer>();

        set2.addAll(Arrays.asList(B));

        // Finding Union of set1 and set2

        Set<Integer> union\_data = **new** HashSet<Integer>(set1);

        union\_data.addAll(set2);

        System.out.print("Union of set1 and set2 is:");

        System.out.println(union\_data);

        // Finding Intersection of set1 and set2

        Set<Integer> intersection\_data = **new** HashSet<Integer>(set1);

        intersection\_data.retainAll(set2);

        System.out.print("Intersection of set1 and set2 is:");

        System.out.println(intersection\_data);

        // Finding Difference of set1 and set2

        Set<Integer> difference\_data = **new** HashSet<Integer>(set1);

        difference\_data.removeAll(set2);

        System.out.print("Difference of set1 and set2 is:");

        System.out.println(difference\_data);

    }

}

## List Vs Set Interface

|  |  |
| --- | --- |
| **List** | **Set** |
| The list implementation allows us to add the same or duplicate elements. | The set implementation doesn't allow us to add the same or duplicate elements. |
| The insertion order is maintained by the List. | It doesn't maintain the insertion order of elements. |
| List allows us to add any number of null values. | Set allows us to add at least one null value in it. |
| The List implementation classes are LinkedList and ArrayList. | The Set implementation classes are TreeSet, HashSet and LinkedHashSet. |
| We can get the element of a specified index from the list using the get() method. | We cannot find the element from the Set based on the index because it doesn't provide any get method(). |
| It is used when we want to frequently access the elements by using the index. | It is used when we want to design a collection of distinct elements. |
| The method of List interface listiterator() is used to iterate the List elements. | The iterator is used when we need to iterate the Set elements. |

# Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

**Lambda expression provides implementation of functional interface*.***

**Lambda Expression Syntax**

**(argument-list) -> {body}**

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Examples:

* 1. **No Parameter Syntax**

() -> {

//Body of no parameter lambda

}

* 1. **One Parameter Syntax**

(p1) -> {

//Body of single parameter lambda

}

* 1. **Two Parameter Syntax**

(p1,p2) -> {

//Body of multiple parameter lambda

}

## Example Without Lambda Expression

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample {

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

**int** width=10;

//without lambda, Drawable implementation using anonymous class

Drawable d=**new** Drawable(){

**public** **void** draw(){System.out.println("Drawing "+width);}

};

d.draw();

}

}

Output:

Drawing 10

## Lambda Expression Example

@FunctionalInterface  //It is optional

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample2 {

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

**int** width=10;

//with lambda

Drawable d2=()->{

System.out.println("Drawing "+width);

};

d2.draw();

}

}

Output:

Drawing 10

## Java Lambda Expression Example: Single Parameter

**interface** Sayable{

**public** String say(String name);

}

**public** **class** LambdaExpressionExample4{

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

// Lambda expression with single parameter.

Sayable s1=(name)->{

**return** "Hello, "+name;

};

System.out.println(s1.say("Sonoo"));

// You can omit function parentheses

Sayable s2= name ->{

**return** "Hello, "+name;

};

System.out.println(s2.say("Sonoo"));

}

}

Output:

Hello, Sonoo

Hello, Sonoo

## Java Lambda Expression Example: Multiple Parameters

**interface** Addable{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaExpressionExample5{

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

// Multiple parameters in lambda expression

Addable ad1=(a,b)->(a+b);

System.out.println(ad1.add(10,20));

// Multiple parameters with data type in lambda expression

Addable ad2=(**int** a,**int** b)->(a+b);

System.out.println(ad2.add(100,200));

}

}

Output:

30

300

## Java Lambda Expression Example: Foreach Loop

**import** java.util.\*;

**public** **class** LambdaExpressionExample7{

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

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

list.add("ankit");

list.add("mayank");

list.add("irfan");

list.add("jai");

list.forEach(

(n)->System.out.println(n)

);

}

}

Output:

ankit

mayank

irfan

jai

# Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example:

@FunctionalInterface

**interface** sayable{

**void** say(String msg);   // abstract method

// It can contain any number of Object class methods.

**int** hashCode();

String toString();

**boolean** equals(Object obj);

}

**public** **class** FunctionalInterfaceExample2 **implements** sayable{

**public** **void** say(String msg){

System.out.println(msg);

}

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

FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();

fie.say("Hello there");

}

}

Output :

Hello there

Invalid Functional Interface

A functional interface can extends another interface only when it does not have any abstract method.

**interface** sayable{

**void** say(String msg);   // abstract method

}

@FunctionalInterface

**interface** Doable **extends** sayable{

// Invalid '@FunctionalInterface' annotation; Doable is not a functional interface

**void** doIt();

}

output:

compile-time error

# Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

This section covers the following topics:

**Stream**

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

Examp :

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

}

**public** **class** JavaStreamExample {

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

List<Product> productsList = **new** ArrayList<Product>();

//Adding Products

productsList.add(**new** Product(1,"HP Laptop",25000f));

productsList.add(**new** Product(2,"Dell Laptop",30000f));

productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

productsList.add(**new** Product(4,"Sony Laptop",28000f));

productsList.add(**new** Product(5,"Apple Laptop",90000f));

List<Float> productPriceList2 =productsList.stream()

.filter(p -> p.price > 30000)// filtering data

.map(p->p.price)        // fetching price

.collect(Collectors.toList()); // collecting as list

.forEach(p-> System.out.println(p)); //

}

}

**Output:**

[90000.0]

**Map**

A Map is useful if you have to search, update or delete elements on the basis of a key.

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.

## Stream sort() Method

The [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) interface provides two methods for sorting the Stream elements.

* **sorted()** – Provides the default sorting
* **sorted(Comparator)** – Sorting based on the provided comparator.

### Stream sorted()

Syntax

Stream<T> sorted()

* sorted() is a **stateful intermediate operation** that returns a new Stream.
* It returns a stream consisting of the elements of this stream, sorted according to the **natural order**.
* If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
* For ordered streams, the sort is stable.
* For unordered streams, no stability guarantees are made.

### Stream sorted(comparator)

Syntax

Stream<T> sorted(Comparator<? **super** T> comparator)

* This is a **stateful intermediate operation** that returns a new stream.
* It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator..
* For ordered streams, the sort is stable.
* For unordered streams, no stability guarantees are made.

## Stream sorted() Examples

### Example 1: Sorting in Natural Order

In the given Java example, we are **sorting a List of integers in the natural order** and printing them into the standard output.

List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

List<Integer> sortedList = list.stream()

.sorted()

.collect(Collectors.toList());

System.out.println(sortedList);

Program output.

[1, 2, 3, 4, 5, 6, 7, 8, 9]

### Example 2: Reverse Ordering

In the given Java example, we are **sorting a stream of integers in reverse order using a Comparator.reverseOrder()** and printing them into the standard output.

List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

List<Integer> sortedList = list.stream()

.sorted(Comparator.reverseOrder())

.collect(Collectors.toList());

System.out.println(sortedList);

Program output.

[9, 8, 7, 6, 5, 4, 3, 2, 1]

### Example 3: Custom Ordering using Comparator

In the given Java example, we are **sorting a stream of integers using a custom Comparator**.

List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

Comparator<Integer> reverseComparator = **new** Comparator<Integer>() {

@Override

**public** **int** compare(Integer i1, Integer i2) {

**return** i2.compareTo(i1);

}

};

List<Integer> sortedList = list.stream()

.sorted(reverseComparator)

.collect(Collectors.toList());

System.out.println(sortedList);

Program output.

[9, 8, 7, 6, 5, 4, 3, 2, 1]

### Example 4: Sorting using Lambda Expressions

Java example to sort a stream of integers in reverse order using [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) to specify the comparison logic.

We are rewriting the previous Comparator logic with an inline lambda expression.

List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

List<Integer> sortedList = list.stream()

.sorted( (i1, i2) -> i2.compareTo(i1) )

.collect(Collectors.toList());

System.out.println(sortedList);

Program output.

[9, 8, 7, 6, 5, 4, 3, 2, 1]

**Spring**:

Framework: used to get solution for the problems.

Spring : Spring is a framework Used to manage the objects.

Managing Object: creat, Initial, destroy

MetaInfo : data of external system passed to Spring.

Step1. Passing Metainfo to Spring

Step2. Creating Bean using spring

Bean is a Object created by Spring

Step3. Getting reference from spring using getBean Method() and @ Autowired.

Virsion:-<properties >

<springversion>5.3.24</spring.version>

</properties >

<dependwncies >

<dependency >

<GroupId >org.Springframework<GroupId >

<vresion > {spring.version}<version>

</dependency >

Congratulation Class : helps to provide meta info @configuration

**Map**:

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:

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. |

### Useful methods of Map interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>> entrySet() | It returns the Set view containing all the keys and values. |
| void clear() | It is used to reset the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| int hashCode() | It returns the hash code value for the Map |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

## Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

### Methods of Map.Entry interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

### Java Map Example: Non-Generic (Old Style)

//Non-generic

**import** java.util.\*;

**public** **class** MapExample1 {

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

Map map=**new** HashMap();

//Adding elements to map

map.put(1,"Amit");

map.put(5,"Rahul");

map.put(2,"Jai");

map.put(6,"Amit");

//Traversing Map

Set set=map.entrySet();//Converting to Set so that we can traverse

Iterator itr=set.iterator();

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

//Converting to Map.Entry so that we can get key and value separately

Map.Entry entry=(Map.Entry)itr.next();

System.out.println(entry.getKey()+" "+entry.getValue());

}

}

}

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

Java Map Example: Generic (New Style)

**import** java.util.\*;

**class** MapExample2{

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

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

map.put(100,"Amit");

map.put(101,"Vijay");

map.put(102,"Rahul");

//Elements can traverse in any order

**for**(Map.Entry m:map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

Output:

102 Rahul

100 Amit

101 Vijay

Java Map Example: comparingByKey()

**import** java.util.\*;

**class** MapExample3{

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

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

map.put(100,"Amit");

map.put(101,"Vijay");

map.put(102,"Rahul");

//Returns a Set view of the mappings contained in this map

map.entrySet()

//Returns a sequential Stream with this collection as its source

.stream()

//Sorted according to the provided Comparator

.sorted(Map.Entry.comparingByKey())

//Performs an action for each element of this stream

.forEach(System.out::println);

}

}

Output:

100=Amit

101=Vijay

102=Rahul

Java Map Example: comparingByKey() in Descending Order

**import** java.util.\*;

**class** MapExample4{

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

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

map.put(100,"Amit");

map.put(101,"Vijay");

map.put(102,"Rahul");

//Returns a Set view of the mappings contained in this map

map.entrySet()

//Returns a sequential Stream with this collection as its source

.stream()

//Sorted according to the provided Comparator

.sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))

//Performs an action for each element of this stream

.forEach(System.out::println);

}

}

Output:

102=Rahul

101=Vijay

100=Amit

Java Map Example: comparingByValue()

**import** java.util.\*;

**class** MapExample5{

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

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

map.put(100,"Amit");

map.put(101,"Vijay");

map.put(102,"Rahul");

//Returns a Set view of the mappings contained in this map

map.entrySet()

//Returns a sequential Stream with this collection as its source

.stream()

//Sorted according to the provided Comparator

.sorted(Map.Entry.comparingByValue())

//Performs an action for each element of this stream

.forEach(System.out::println);

}

}

Output:

100=Amit

102=Rahul

101=Vijay

Java Map Example: comparingByValue() in Descending Order

**import** java.util.\*;

**class** MapExample6{

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

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

map.put(100,"Amit");

map.put(101,"Vijay");

map.put(102,"Rahul");

//Returns a Set view of the mappings contained in this map

map.entrySet()

//Returns a sequential Stream with this collection as its source

.stream()

//Sorted according to the provided Comparator

.sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))

//Performs an action for each element of this stream

.forEach(System.out::println);

}

}

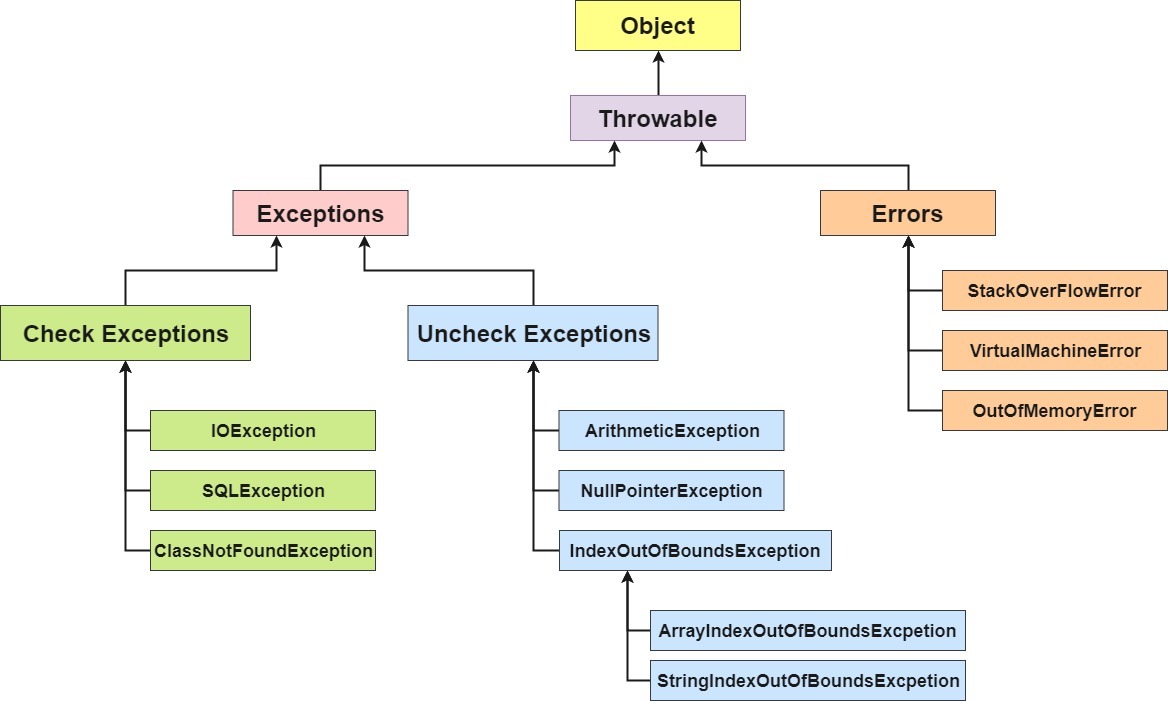
Output:

**Exception :**

Exception is an abnormal event which will trigger in the jvm at the time of execution, it is also class and object and run time is completely handled by jvm and compile time is completely handled by jdk.Abnormal event during the exiquation of the program by jvm.

nullPointer Exception, Arithmetic Exception, ArrayIndexOutOfBoundsException

Exception Hierarchy



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

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 example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

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

example

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

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

**try**{

//code that may raise exception

**int** data=100/0;

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

//rest code of the program

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

}

}

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

**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.

**Points to remember**

* 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 ArithmeticException must come before catch for Exception.

example:

**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");

}

output

Arithmetic Exception occurs

rest of the code

**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.

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.

example:

**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 occured or not

**finally** {

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

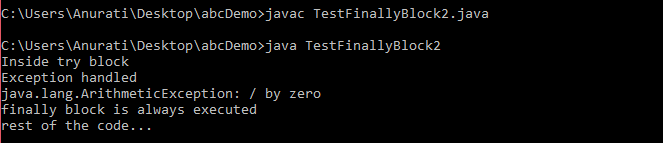
      }

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

      }

    }

**Output:**



Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

**Java throw Exception**

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

**Java throw keyword**

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. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

**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

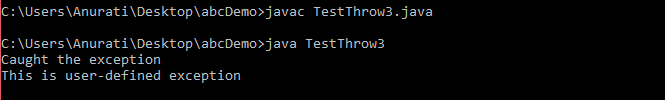
            System.out.println(ude.getMessage());

        }

    }

}

**Output:**



Java throws keyword

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

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

Syntax of Java throws

1. return\_type method\_name() **throws** exception\_class\_name{
2. //method code
3. }

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).

**import** java.io.\*;

**class** M{

**void** method()**throws** IOException{

**throw** **new** IOException("device error");

 }

}

**class** Testthrows4{

**public** **static** **void** main(String args[])**throws** IOException{//declare exception

     M m=**new** M();

     m.method();

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

  }

}

**Output:**

Java throw keyword

|  |  |  |
| --- | --- | --- |
| **Final** | **Finally** | **Finalize** |
| final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

**JDBC**

import java.sql.\*;

public class JDBCDemo {

public static void main(String args[])

throws SQLException, ClassNotFoundException

{

String driverClassName

= "sun.jdbc.odbc.JdbcOdbcDriver";

String url = "jdbc:odbc:XE";

String username = "scott";

String password = "tiger";

String query

= "insert into students values(109, 'bhatt')";

// Load driver class

Class.forName(driverClassName);

// Obtain a connection

Connection con = DriverManager.getConnection(

url, username, password);

// Obtain a statement

Statement st = con.createStatement();

// Execute the query

int count = st.executeUpdate(query);

System.out.println(

"number of rows affected by this query= "

+ count);

// Closing the connection as per the requirement with connection is completed

con.close();

}

} // class

**JDBC**

JDBC (Java Database Connectivity) is a standard API that provides Java programs with the ability to access databases. Here are the seven steps involved in using JDBC:

1. Load the JDBC driver: The first step is to load the JDBC driver using the **Class.forName()** method. This method loads the driver class into memory so that it can be used to connect to the database.
2. Open a connection: The next step is to open a connection to the database using the **DriverManager.getConnection()** method. This method takes a database URL, username, and password as parameters and returns a **Connection** object.
3. Create a statement: Once a connection is established, you can create a **Statement** object using the **Connection.createStatement()** method. A **Statement** object is used to execute SQL statements against the database.
4. Execute the query: You can execute a SQL query using the **Statement.executeQuery()** method. This method returns a **ResultSet** object that contains the results of the query.
5. Process the results: The **ResultSet** object can be used to iterate through the results of the query using the **ResultSet.next()** method. You can also retrieve values from the **ResultSet** using methods like **getInt()**, **getString()**, etc.
6. Close the statement: Once you are finished with the **ResultSet**, you should close the **Statement** object using the **Statement.close()** method. This releases any resources held by the **Statement** object.
7. Close the connection: Finally, you should close the **Connection** object using the **Connection.close()** method. This releases any resources held by the **Connection** object and closes the connection to the database.

**Spring**

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as [Struts](https://www.javatpoint.com/struts-2-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), Tapestry, [EJB](https://www.javatpoint.com/ejb-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

**Spring Annotations**

Spring Boot is a popular framework that is built on top of the Spring framework, and it provides many features to make it easier to build and run Spring-based applications. Some of the main annotations used in Spring Boot and their uses are:

1. **@SpringBootApplication**: This annotation is used to mark the main class of a Spring Boot application. It is a combination of three other annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**. It indicates that the class should be treated as a Spring configuration class, and that it should enable auto-configuration of the Spring application context, and scan for Spring-managed components.
2. **@RestController**: This annotation is used to mark a class that defines RESTful web services. It combines the **@Controller** and **@ResponseBody** annotations, and indicates that the class should be treated as a controller that returns data directly in the response body.
3. **@RequestMapping**: This annotation is used to map HTTP requests to specific methods in a controller class. It can be used to specify the URL path, HTTP method, request headers, and query parameters that a method should handle.
4. **@Autowired**: This annotation is used to automatically wire dependencies into a Spring bean. It can be used to inject dependencies into a constructor, a setter method, or directly into a field.
5. **@Component**: This annotation is used to mark a class as a Spring-managed component. It is a general-purpose annotation that can be used to mark any class as a Spring bean.
6. **@Service**: This annotation is used to mark a class as a service bean. It is a specialization of the **@Component** annotation, and is used to indicate that the class provides a specific service or functionality.
7. **@Repository**: This annotation is used to mark a class as a repository bean. It is also a specialization of the **@Component** annotation, and is used to indicate that the class provides data access functionality.

These are some of the main annotations used in Spring Boot, and they can be used to configure and customize a Spring-based application. By using these annotations, you can simplify the configuration of your application and reduce the amount of boilerplate code you need to write.

**Difference between spring MVC and spring boot**

Spring MVC and Spring Boot are both Java-based frameworks developed by the Spring Framework community, but they serve different purposes.

Spring MVC is a web application framework that provides a model-view-controller (MVC) architecture for building web applications. It provides a set of core components and patterns for handling HTTP requests and responses, including controllers, views, and data models. Spring MVC is built on top of the core Spring Framework and is typically used to develop complex, enterprise-grade web applications.

Spring Boot, on the other hand, is a framework that makes it easier to create standalone, production-ready Spring-based applications with minimal configuration. It provides a set of opinionated defaults and auto-configuration features that simplify the process of setting up and deploying Spring-based applications. Spring Boot is designed to help developers create microservices and other lightweight applications quickly and easily.

In summary, the main differences between Spring MVC and Spring Boot are:

* Spring MVC is a web application framework that provides an MVC architecture for building complex web applications, while Spring Boot is a framework for creating standalone, production-ready applications with minimal configuration.
* Spring MVC requires more configuration and setup to get started, while Spring Boot provides a set of opinionated defaults and auto-configuration features that simplify the development process.
* Spring MVC is typically used for developing large-scale, enterprise-grade web applications, while Spring Boot is used for creating lightweight applications and microservices.

Both frameworks have their own strengths and are widely used in the Java community, depending on the specific requirements of the application being developed.

**Question and answer on spring**

1. What is Spring Framework?

* Spring Framework is an open-source Java-based framework that provides a set of libraries and tools for building enterprise-grade applications. It is widely used to develop Java applications of various sizes and complexities.

1. What is Inversion of Control (IoC) in Spring?

* Inversion of Control (IoC) is a design pattern in which the control flow of a program is inverted. Instead of the programmer controlling the flow of the program, the framework or container takes control and manages the program's execution. Spring uses IoC to provide loose coupling between objects and to improve the maintainability of applications.

1. What is Dependency Injection (DI) in Spring?

* Dependency Injection (DI) is a design pattern in which objects are provided with their dependencies rather than creating them internally. Spring uses DI to provide loose coupling between objects and to make it easier to replace dependencies during testing or future development.

1. What is the difference between constructor injection and setter injection in Spring?

* Constructor injection is a type of dependency injection in which dependencies are passed to the object's constructor. Setter injection is another type of dependency injection in which dependencies are set using setter methods. Constructor injection is preferred when dependencies are required and should be immutable, while setter injection is preferred when dependencies are optional and may be changed.

1. What is the Spring MVC framework?

* The Spring MVC framework is a web application framework that provides a model-view-controller (MVC) architecture for building web applications in Java. It provides a set of core components and patterns for handling HTTP requests and responses, including controllers, views, and data models.

1. What are the components of the Spring MVC framework?

* The components of the Spring MVC framework include the dispatcher servlet, controllers, views, and model objects. The dispatcher servlet receives HTTP requests and forwards them to the appropriate controller. The controllers handle requests and generate responses. The views generate the HTML, CSS, and JavaScript that are sent back to the client. The model objects hold the data that is used to generate the view.

1. What is Spring Boot and how is it different from the Spring Framework?

* Spring Boot is a framework for creating standalone, production-ready applications with minimal configuration. It provides a set of opinionated defaults and auto-configuration features that simplify the process of setting up and deploying Spring-based applications. Spring Boot is built on top of the Spring Framework and is designed to help developers create microservices and other lightweight applications quickly and easily.

1. What is the role of the @Controller annotation in Spring MVC?

* The @Controller annotation is used to indicate that a class is a controller in Spring MVC. Controllers are responsible for handling HTTP requests and generating responses. They are typically annotated with @RequestMapping annotations to specify the URLs that they handle.

1. What is the difference between @Component, @Service, and @Repository annotations in Spring?

* The @Component annotation is a generic stereotype annotation that can be used to annotate any class. The @Service annotation is used to annotate classes that provide business logic services. The @Repository annotation is used to annotate classes that provide data access services. All three annotations are used to provide metadata to the Spring container, which uses the metadata to manage objects and dependencies.

1. What is a bean in Spring?

* A bean is an object that is managed by the Spring container. Beans are created, initialized, and wired together by the Spring container based on the metadata provided by the developer.

1. What is the purpose of the @Autowired annotation in Spring?

* The @Autowired annotation is used to inject dependencies into a Spring-managed object. It can be used to inject dependencies by type or by name.

1. How does Spring provide support for transaction management?

* Spring provides support for transaction

**Java Interview Questions**

Even or Odd programe?

class EvenOrOdd {

public static void main (String [] args){

int [] a = {1,2,3,4,5,6,7,8,9};

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

if (a[i]%2 == 0){

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

}

}

}

}

Prime or not ?

class PrimeOrNot {

public static void main (String [] args){

int n = 5;

boolean prime = true;

for (int i = 2; i < n ; i++){

if (n%i == 0){

prime = false;

}

}

System.out.println (prime);

}

}

Adding Array elements?

class ArrayAdd {

public static void main (String [] args){

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

int sum = 0;

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

sum = sum+ a[i];

}

System.out.println (sum);

}

}

Add the number using Scannere/User input?

class Add {

public static void main (String [] args){

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

int sum = 0;

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

sum = sum+ a[i];

}

System.out.println (sum);

}

}

Find the duplicate elements in the array ?

class DuplicateElements {

public static void main (String [] args){

int a [] = {1,2,2,3,4,5,6,7,7};

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

for (int j =i+1; j<a.length ;j++){

if (a[i]==a[j]){

System.out.print (a[j]);

}

}

}

}

}

Find the Factoriall of the given number?

class Factoriall

{

public static int fact(int n)

{

int factorialValue = 1;

for(int f=1; f<=n; f++){

factorialValue = factorialValue\*f;

}

return factorialValue;

}

}

Find the higesrt number in the array?

class HigestNoinArray {

public static void main (String [] args){

int a[] = {34,45,67,98,99,105,358};

int higest = a [0];

//int lowest = a[0];

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

if (a[i] > higest);

higest = a[i];

/\* if (a[i]<lowest); \*/

//lowest = a[i];

}

System.out.print (higest);

}

}

Find the second higesrt number in the array?

class SecondLargestNo {

public static void main (String [] args){

int a [] = {1,2,3,5,6,7,8,9,566};

int firstNo = 0;

int secondNo = 0;

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

if (firstNo < a[i]){

secondNo = firstNo;

firstNo = a[i];

}

else if (secondNo< a[i]){

secondNo= a[i];

}

}

System.out.print (secondNo);

}

}

Revers the given number?

class Revers {

static int c;

public static void main (String [] args){

int a = 846;

int b = 0;

while (a>0){

c = a%10;

b = b\*10+c;

a = a/10;

}

System.out.println (b);

}

}

**Write a Java program to create and throw custom exceptions?**

***// class that uses custom exception InvalidAgeException***

***public class TestCustomException1***

***{***

***// method to check the age***

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

***if(age < 18){***

***// throw an object of user defined exception***

***throw new InvalidAgeException("age is not valid to vote");***

***}***

***else {***

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

***}***

***}***

***// main method***

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

***{***

***try***

***{***

***// calling the method***

***validate(13);***

***}***

***catch (InvalidAgeException ex)***

***{***

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

***// printing the message from InvalidAgeException object***

***System.out.println("Exception occured: " + ex);***

***}***

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

***}***

***}***

Write a code to check ***DuplicateCharacters  in a string?***

***public class DuplicateCharacters {***

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

***String string1 = "Great responsibility";***

***int count;***

***//Converts given string into character array***

***char string[] = string1.toCharArray();***

***System.out.println("Duplicate characters in a given string: ");***

***//Counts each character present in the string***

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

***count = 1;***

***for(int j = i+1; j <string.length; j++) {***

***if(string[i] == string[j] && string[i] != ' ') {***

***count++;***

***//Set string[j] to 0 to avoid printing visited character***

***string[j] = '0';***

***}***

***}***

***//A character is considered as duplicate if count is greater than 1***

***if(count > 1 && string[i] != '0')***

***System.out.println(string[i]);***

***}***

***}***

***}***

Remove all the white space in a string?

public class BlankSpace {

public static void main(String[] args)

{

String str = " Geeks for Geeks ";

// Call the replaceAll() method

str = str.replaceAll("\\s", "");

System.out.println(str);

}

}

Connect to database using jdbc and fetch a data in text formate?

+----+----------+-----------+--------+

| id | first\_name | last\_name | salary |

+----+------------+-----------+--------+

| 1 | John | Doe | 50000 |

| 2 | Jane | Smith | 60000 |

| 3 | Bob | Johnson | 55000 |

+----+------------+-----------+--------+

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

public class JDBCTextDataFetch {

public static void main(String[] args) {

// Database connection parameters

String jdbcUrl = "jdbc:mysql://localhost:3306/your\_database";

String username = "your\_username";

String password = "your\_password";

try {

// Step 1: Register JDBC driver

Class.forName("com.mysql.jdbc.Driver");

// Step 2: Establish a connection to the database

Connection connection = DriverManager.getConnection(jdbcUrl, username, password);

// Step 3: Create a statement

Statement statement = connection.createStatement();

// Step 4: Execute a SQL query

String sqlQuery = "SELECT \* FROM your\_table";

ResultSet resultSet = statement.executeQuery(sqlQuery);

// Step 5: Process the ResultSet and fetch data in text format

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

String description = resultSet.getString("description");

// Process and print the data in text format

System.out.println("ID: " + id);

System.out.println("Name: " + name);

System.out.println("Description: " + description);

System.out.println(); // Add a newline for separation

}

// Step 6: Close the resources

resultSet.close();

statement.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}