1) Why multiple inheritance not possible in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Ex:

\*\*\*

class A

{

void msg()

{

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

}

class B

{

void msg()

{

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

}

class C extends A,B

{//suppose if it were

Public Static void main(String args[]){

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}

o/p:

\*\*\*

Compile time error

2) Why Method overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

EX:

\*\*\*

class Adder{

static int add(int a,int b){

return a+b;

}

static double add(int a,int b){

return a+b;

}

}

class TestOverloading3{

public static void main(String[] args){

System.out.println(Adder.add(11,11));//ambiguity

}

}

o/p:

\*\*\*

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note:

\*\*\*\*\*

Compile time error is better than run time error. So, java compiler renders compiler time error if you declare the same method having some parameters.

3) Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

class TestOverloading4{

public static void main(String[] args){

System.out.println("main with String[]");

}

public static void main(String args){

System.out.println("main with String");

}

public static void main(){

System.out.println("main without args");

}

}

o/p:

\*\*\*

Main with String[]

Real time example of method overriding

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Bank{

int getRateOfInterest(){

return 0;

}

}

class SBI extends Bank{

int getRateOfInterest(){

return 8;

}

}

class ICICI extends Bank{

int getRateOfInterest(){

return 7;

}

}

class AXIS extends Bank{

int getRateOfInterest(){

return 9;

}

}

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());

}

}

o/p:

\*\*\*\*

SBI Rate of Interest: 8

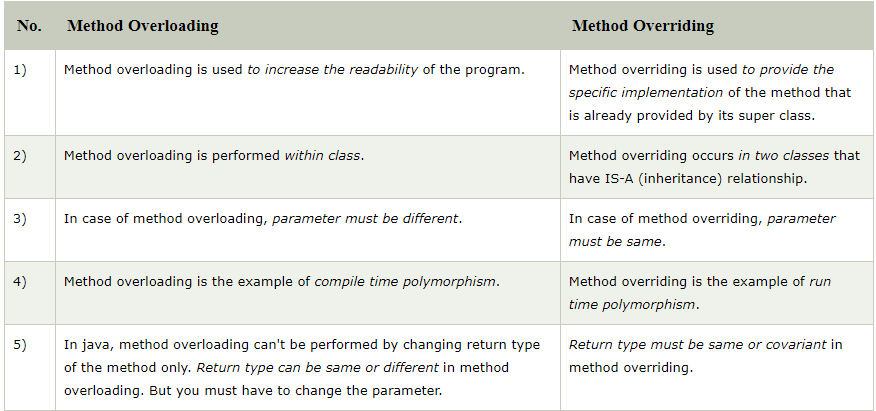
ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

We cannot override static method because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

We cannot override java main method

4) Different between method overloading and method overriding?



5)Java Access Modifiers with Method Overriding?

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

EX:

\*\*\*

class A{

protected void msg(){System.out.println("Hello java");}

}

public class Simple extends A{

void msg(){System.out.println("Hello java");}//C.T.Error

 public static void main(String args[]){

   Simple obj=new Simple();

   obj.msg();

}

}

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

6)Exception Handling with Method overloading in java?

There are many rules if we talk about method overriding with exception handling. The Rules are as follows:

🡺If the superclass method does not declare an exception

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

🡺If the superclass method declares an exception

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Rule no-1:

\*\*\*\*\*\*\*\*\*

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

EX:

\*\*\*

import java.io.\*;

class Parent{

void msg(){

System.out.println("parent");

}

}

class TestExceptionChild extends Parent{

void msg()throws IOException{

System.out.println("TestExceptionChild");

}

public static void main(String args[]){

Parent p=new TestExceptionChild();

p.msg();

}

}

o/p:

\*\*\*

Compile Time error

Rule No-2:

\*\*\*\*\*\*\*\*\*

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Ex:

\*\*\*

import java.io.\*;

class Parent{

void msg(){System.out.println("parent");}

}

class TestExceptionChild1 extends Parent{

void msg()throws ArithmeticException{

System.out.println("child");

}

public static void main(String args[]){

Parent p=new TestExceptionChild1();

p.msg();

}

}

o/p:

\*\*\*

Child

7)Covariant Return Type

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

EX:

\*\*\*

class A{

A get(){return this;}

}

class B1 extends A{

B1 get(){return this;}

void message(){System.out.println("welcome to covariant return type");

}

public static void main(String args[]){

new B1().get().message();

}

}

o/p: welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

8)How is Covariant types implemented?

Java doesn't allow the return type based overloading but JVM always allows return type based overloading. JVM uses full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. Java c uses this fact to implement covariant return types.

What are the rules for instance initializer block?

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

9)Instance initializer block

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| Instance initializer block is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

Example of instance initializer block:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class A{

A(){

System.out.println("parent class constructor invoked");

}

}

class B3 extends A{

B3(){

super();

System.out.println("child class constructor invoked");

}

B3(int a){

super();

System.out.println("child class constructor invoked "+a);

}

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

public static void main(String args[]){

B3 b1=new B3();

B3 b2=new B3(10);

}

}

o/p:

\*\*\*\*

Parent class constructor invoked

Instance initializer block is invoked

Child class constructor invoked

Parent class constructor invoked

Instance initializer block is invoked

Child class constructor invoked 10

10)Final class

\*\*\*\*\*\*\*\*\*

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

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

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

11)Is final method inherited?

Yes, final method is inherited but you cannot override it.

EX:

\*\*\*

class Bike{

  final void run(){

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

}

}

class Honda2 extends Bike{

   public static void main(String args[]){

    new Honda2().run();

   }

}

o/p:

\*\*\*

Running

12)What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

EX:

\*\*\*

Class student{

Int id;

String name;

Final String PAN\_CARD\_NUMBER;

…..

}

13)Can we initialize blank final variable?

Yes, only in constructor

EX:

\*\*\*

Class A{

Final int a; //blank final variable

A(){

a=10;

System.out.println(a);

}

Public static void main(String []args){

New A();

}

}

o/p:10

14)Static blank final variable

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

EX:

\*\*\*

Class A{

Static final int data; //static blank final variable

Static

{

Data=50;

}

Public static void main(String []args){

System.out.println(A.data);

}

}

15)What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

16)Can we declare a constructor final?

No, because constructor is never inherited.

17)What is interface?

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

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

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

🡪Java Interface also represents the IS-A relationship.

🡪It cannot be instantiated just like the abstract class.

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

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

18)Multiple inheritance with multiple interface

EX:

\*\*\*

interface Printable{

void print();

}

interface Showable{

void show();

}

class A7 implements Printable,Showable{

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

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

public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

}

}

o/p:

\*\*\*

Hello

Welcome

19)Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class

EX:

\*\*\*

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestInterface3 implements Printable, Showable{

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

public static void main(String args[]){

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

o/p:

\*\*\*\*

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

20) Interface inheritance

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

EX:

\*\*\*

interface Printable{

void print();

}

interface Showable extends Printable{

void show();

}

class TestInterface4 implements Showable{

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

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

public static void main(String args[]){

TestInterface4 obj = new TestInterface4();

obj.print();

obj.show();

 }

}

o/p:

\*\*\*

Hello

Welcome

21)Java 8 Default Method in interface

Since Java 8, we can have method body in interface. But we need to make it default method.

EX

\*\*\*

interface Drawable{

void draw();

default void msg(){System.out.println("default method");}

}

class Rectangle implements Drawable{

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

}

class TestInterfaceDefault{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

d.msg();

}

}

o/p:

\*\*\*

Drawing rectangle

Default method

22)java 8 static method in interface

Since java 8, we can have static method in interface.

EX:

\*\*\*

interface Drawable{

void draw();

static int cube(int x){return x\*x\*x;}

}

class Rectangle implements Drawable{

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

}

class TestInterfaceStatic{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

System.out.println(Drawable.cube(3));

}

}

o/p:

\*\*\*

Drawing rectangle

27

23) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

EX:

\*\*\*

// how serializable interface is written?

Public interface serializable{

}

Nested interface will be later

**ARRAY**

Array

\*\*\*\*\*

🡺Normally, an array is a collection of similar type of elements which have a contiguous memory location.

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

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

🡺Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

🡺In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

🡺Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

Advantages:

\*\*\*\*\*\*\*\*\*\*

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

Dis Advantages

\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Single dimensional Array Syntax

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dataType[] arr; (or) dataType []arr; (or) dataType arr[];

initiation or declaration of array

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Datatype var[]=new datatype[size];

EX: int var[]=new int[5];

Example of single dimension array program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Testarray{

public static void main(String args[]){

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

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//printing array

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

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

}

}

o/p:

\*\*\*\*

10

20

70

40

50

We can declare it is also correct

Int a[]={33,4,5};

For(int i=0;i<=a.length-1;i++){

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

}

Using for each loop

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If we don’t know a starting point and ending point then we go for For-Each loop

Syntax

\*\*\*\*\*\*

For(datatype var:collection/arraytype){

//body

}

Ex:

\*\*\*

Class Test{

Public static void main(String[] args){

Int []a={33,6,98,87};

For(int i:a){

System.out.println(i);

}

}

}

o/p:

\*\*\*

33

6

98

87

Passing Array to a Method in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Ex:

\*\*\*

class test{

static void min(int arr[]) {

int min=arr[0];

for(int i=1;i<arr.length-1;i++) {

if(min>arr[i]) {

min=arr[i];

System.*out*.println(min);

}

}

}

public static void main(String[] args) {

int a[]= {30,20,65,70};

*min*(a);

}

}

o/p:

\*\*\*

20

Anonymous Array in java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java supports the feature of an anonymous array, so you don’t need to declare the array while passing an array to the method.

Java program to demonstrate the way of passing an anonymous array

Class test{

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

Static void print array(int arr[]){

For(int i=0;i<arr.length;i++){

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

}

Public static void main(String args[]){

printAraay(new int[]{10,22,44,66});//passing anonymous array to method

}

}

o/p:

\*\*\*\*

10

22

44

66

Returning array from the method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Java Program to return an array from the method

class TestReturnArray{

//creating method which returns an array

static int[] get(){

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

}

public static void main(String args[]){

//calling method which returns an array

int arr[]=get();

//printing the values of array

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

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

}

}

}

o/p

\*\*\*

10

30

50

90

60

ArrayIndexOutOfBoundsException

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

EX:

\*\*\*

//Java Program to demonstrate the case of

//ArrayIndexOutOfBoundsException in a Java Array.

public class TestArrayException{

public static void main(String args[]){

int arr[]={50,60,70,80};

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

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

}

}

}

o/p:

\*\*\*\*

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

Multidimensional Array in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In such case, data is stored in row and column based index(also known as matrix form).

Multidimensional Array syntax

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

Ex: int[][] arr=new int[3][3]; //3 row and 3 coloumn

Ex:

\*\*\*

arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

Example of Multidimensional java array

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **class** Test1{

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

//declaring and initializing 2D array

**int** arr[][]={{10,20,30,40,50},{60,70,80,90,100},{110,120,130,140,150}};

//printing 2D array

**for**(**int** i=0;i<5;i++) {

**for**(**int** j=0;j<5;j++) {

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

}

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

}

}

}

o/p:

\*\*\*\*

10 20 30 40 50

60 70 80 90 100

110 120 130 140 150

Jagged Array in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

EX:

\*\*\*

//Java Program to illustrate the jagged array

class TestJaggedArray{

public static void main(String[] args){

//declaring a 2D array with odd columns

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

arr[0] = new int[3];

arr[1] = new int[4];

arr[2] = new int[2];

//initializing a jagged array

int count = 0;

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

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

arr[i][j] = count++;

//printing the data of a jagged array

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

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

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

}

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

}

}

}

o/p:

\*\*\*\*

0 1 2

3 4 5 6

7 8

What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

Ex:

\*\*\*

class Testarray4{

public static void main(String args[]){

//declaration and initialization of array

int arr[]={4,4,5};

//getting the class name of java array

Class c=arr.getClass();

String name=c.getName();

//printing the class name of java array

System.out.println(name);

}

}

o/p:

\*\*\*

[I

Copying a Java Array

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We can copy an array to another by the arraycopy() method of System class.

Syntax of arraycopy method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**public** **static** **void** arraycopy(

Object src, **int** srcPos,Object dest, **int** destPos, **int** length

)

Ex:

\*\*\*

/Java Program to copy a source array into a destination array in Java

class TestArrayCopyDemo {

    public static void main(String[] args) {

        //declaring a source array

        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };

        //declaring a destination array

        char[] copyTo = new char[7];

        //copying array using System.arraycopy() method

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);

        //printing the destination array

        System.out.println(String.valueOf(copyTo));

    }

}

o/p:

\*\*\*\*

Caffeine

Cloning an array in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

EX:

\*\*\*

//Java Program to clone the array

**class** Testarray1{

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

**int** arr[]={33,3,4,5};

System.out.println("Printing original array:");

**for**(**int** i:arr)

System.out.println(i);

System.out.println("Printing clone of the array:");

**int** carr[]=arr.clone();

**for**(**int** i:carr)

System.out.println(i);

System.out.println("Are both equal?");

System.out.println(arr==carr);

}

}

o/p:

\*\*\*

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

False

Addition of 2 Matrices in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Testarray5{

public static void main(String args[]){

//creating two matrices

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

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

//creating another matrix to store the sum of two matrices

int c[][]=new int[2][3];

//adding and printing addition of 2 matrices

for(int i=0;i<2;i++){

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

c[i][j]=a[i][j]+b[i][j];

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

}

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

}

}

}

o/p:

\*\*\*\*

2 6 8

6 8 10

Multipication of 2 Matrices in java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



public class MatrixMultiplicationExample{

public static void main(String args[]){

//creating two matrices

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

int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two matrices

int c[][]=new int[3][3]; //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices

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

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

c[i][j]=0;

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

{

c[i][j]+=a[i][k]\*b[k][j];

}//end of k loop

System.out.print(c[i][j]+" "); //printing matrix element

}//end of j loop

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

}

}

}

o/p:

\*\*\*\*

6 6 6

12 12 12

18 18 18

Java OOPS And Misc

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Object class in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

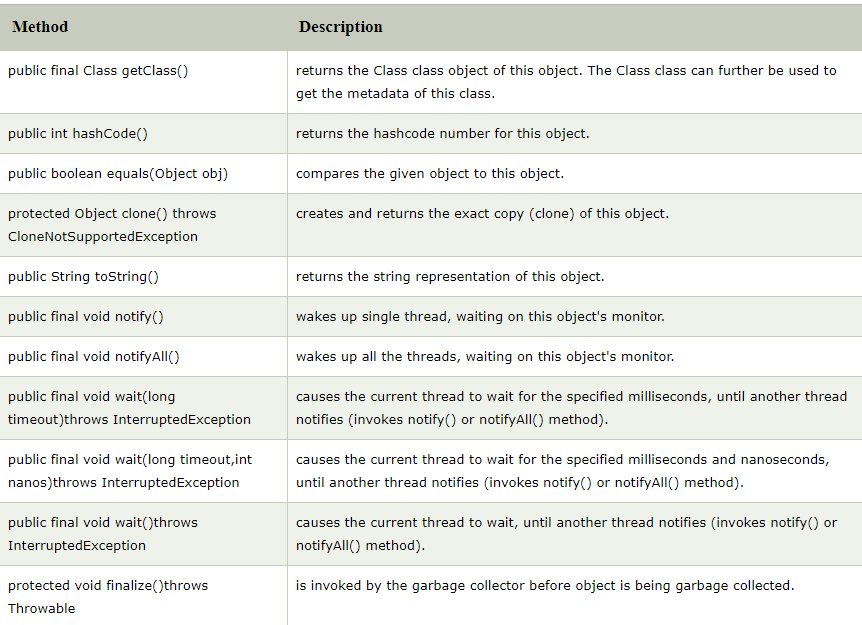
The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

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.

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

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Objecting cloning in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

The clone() method is defined in the Object class.

Syntax of the clone() method is

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Protected Object clone() throws CloneNotSupportedException

Why use clone() method?

The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

* You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
* It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
* Clone() is the fastest way to copy array.

Disadvantage of Object cloning

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Following is a list of some disadvantages of clone() method:

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Ex:

\*\*\*

class Student18 implements Cloneable{

int rollno;

String name;

Student18(int rollno,String name){

this.rollno=rollno;

this.name=name;

}

public Object clone()throws CloneNotSupportedException{

return super.clone();

}

public static void main(String args[]){

try{

Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);

System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}

}

Java Math class

\*\*\*\*\*\*\*\*\*\*\*\*\*

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Ex:

\*\*\*

public class JavaMathExample1

{

public static void main(String[] args)

{

double x = 28;

double y = 4;

// return the maximum of two numbers

System.out.println("Maximum number of x and y is: " +Math.max(x, y)); //28.0

// return the square root of y

System.out.println("Square root of y is: " + Math.sqrt(y)); //2.0

//returns 28 power of 4 i.e. 28\*28\*28\*28

System.out.println("Power of x and y is: " + Math.pow(x, y)); //614656.0

// return the logarithm of given value

System.out.println("Logarithm of x is: " + Math.log(x)); //3.3322

System.out.println("Logarithm of y is: " + Math.log(y)); //1.3862

// return the logarithm of given value when base is 10

System.out.println("log10 of x is: " + Math.log10(x)); //1.4471

System.out.println("log10 of y is: " + Math.log10(y)); //0.6020

// return the log of x + 1

System.out.println("log1p of x is: " +Math.log1p(x)); //3.3672

// return a power of 2

System.out.println("exp of a is: " +Math.exp(x)); //1.4462

// return (a power of 2)-1

System.out.println("expm1 of a is: " +Math.expm1(x)); //1.44625

}

}

Ex-2:

\*\*\*\*\*

public class JavaMathExample2

{

public static void main(String[] args)

{

double a = 30;

// converting values to radian

double b = Math.toRadians(a);

// return the trigonometric sine of a

System.out.println("Sine value of a is: " +Math.sin(a)); //-0.98803

// return the trigonometric cosine value of a

System.out.println("Cosine value of a is: " +Math.cos(a)); //0.1542

// return the trigonometric tangent value of a

System.out.println("Tangent value of a is: " +Math.tan(a)); //-6.4053

// return the trigonometric arc sine of a

System.out.println("Sine value of a is: " +Math.asin(a)); //Nan

// return the trigonometric arc cosine value of a

System.out.println("Cosine value of a is: " +Math.acos(a)); //Nan

// return the trigonometric arc tangent value of a

System.out.println("Tangent value of a is: " +Math.atan(a)); //1.5374

// return the hyperbolic sine of a

System.out.println("Sine value of a is: " +Math.sinh(a)); //5.3432

// return the hyperbolic cosine value of a

System.out.println("Cosine value of a is: " +Math.cosh(a)); //5.3432

// return the hyperbolic tangent value of a

System.out.println("Tangent value of a is: " +Math.tanh(a)); //1.0

}

}

Wrapper classes in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

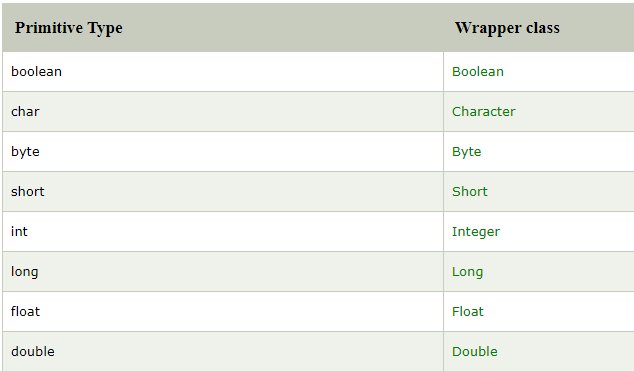
Use of Wrapper classes in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:



Autoboxing

\*\*\*\*\*\*\*\*\*\*

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

EX:

\*\*\*

//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);  //20 20 20

}

}

Unboxing

\*\*\*\*\*\*\*\*

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

EX:

\*\*\*

//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);  //3 3 3

}

}

Java Wrapper classes Example

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Java Program to convert all primitives into its corresponding

//wrapper objects and vice-versa

public class WrapperExample3{

public static void main(String args[]){

byte b=10;

short s=20;

int i=30;

long l=40;

float f=50.0F;

double d=60.0D;

char c='a';

boolean b2=true;

//Autoboxing: Converting primitives into objects

Byte byteobj=b;

Short shortobj=s;

Integer intobj=i;

Long longobj=l;

Float floatobj=f;

Double doubleobj=d;

Character charobj=c;

Boolean boolobj=b2;

//Printing objects

System.out.println("---Printing object values---");

System.out.println("Byte object: "+byteobj);

System.out.println("Short object: "+shortobj);

System.out.println("Integer object: "+intobj);

System.out.println("Long object: "+longobj);

System.out.println("Float object: "+floatobj);

System.out.println("Double object: "+doubleobj);

System.out.println("Character object: "+charobj);

System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives

byte bytevalue=byteobj;

short shortvalue=shortobj;

int intvalue=intobj;

long longvalue=longobj;

float floatvalue=floatobj;

double doublevalue=doubleobj;

char charvalue=charobj;

boolean boolvalue=boolobj;

//Printing primitives

System.out.println("---Printing primitive values---");

System.out.println("byte value: "+bytevalue);

System.out.println("short value: "+shortvalue);

System.out.println("int value: "+intvalue);

System.out.println("long value: "+longvalue);

System.out.println("float value: "+floatvalue);

System.out.println("double value: "+doublevalue);

System.out.println("char value: "+charvalue);

System.out.println("boolean value: "+boolvalue);

}

}

o/p:

\*\*\*\*

---Printing object values---

Byte object: 10

Short object: 20

Integer object: 30

Long object: 40

Float object: 50.0

Double object: 60.0

Character object: a

Boolean object: true

---Printing primitive values---

byte value: 10

short value: 20

int value: 30

long value: 40

float value: 50.0

double value: 60.0

char value: a

boolean value: true

Custom Wrapper class in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

Ex:

\*\*\*

//Creating the custom wrapper class

class Javatpoint{

private int i;

Javatpoint(){}

Javatpoint(int i){

this.i=i;

}

public int getValue(){

return i;

}

public void setValue(int i){

this.i=i;

}

@Override

public String toString() {

  return Integer.toString(i);

}

}

//Testing the custom wrapper class

public class TestJavatpoint{

public static void main(String[] args){

Javatpoint j=new Javatpoint(10);

System.out.println(j);  //10

}

}

Call by Value and Call by Reference in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Ex:

\*\*\*

In case of call by value original value is not changed. Let's take a simple example:

class Operation{

 int data=50;

 void change(int data){

 data=data+100;//changes will be in the local variable only

 }

 public static void main(String args[]){

   Operation op=new Operation();

   System.out.println("before change "+op.data);  //50

   op.change(500);

   System.out.println("after change "+op.data);  //50

 }

}

Ex-2:

\*\*\*\*\*

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

class Operation2{

 int data=50;

 void change(Operation2 op){

 op.data=op.data+100;//changes will be in the instance variable

 }

 public static void main(String args[]){

  Operation2 op=new Operation2();

   System.out.println("before change "+op.data);  //50

   op.change(op);//passing object

   System.out.println("after change "+op.data);  //150

 }

}

Recursion in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax

\*\*\*\*\*\*

Returntype method name{

//code to be executed

Method name();//calling same method

}

Ex-1:

\*\*\*\*\*

Public class RecursionExample1{

Static void p(){

System.out.println(“hello”);

P();

}

Public static void main(String[] args){

P();

}

}

o/p:

\*\*\*\*

Hello

Hello

………

……….

Java.lang

Ex-2:

\*\*\*\*\*

public class RecursionExample2 {

static int count=0;

static void p(){

count++;

if(count<=5){

System.out.println("hello "+count);

p();

}

}

public static void main(String[] args) {

p();

}

}

o/p:

\*\*\*\*

Hello 1

Hello 2

Hello 3

Hello 4

Hello 5

Factorial using Recursion

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Factorial{

Static int factorial(int n){

If (n==1){

return 1;

}

Else{

Return(n\*factorial(n-1));

}

Public static void main(String []args){

System.out.println(“Factorial of 5 is: “+factorial(5));

}

}

o/p:

\*\*\*\*

Factorial of 5 is :120

Fibonacci Series using Recursion

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Recursion{

Static int n1=0, n2=1, n3=0;

Static void printFibo(int count){

If(count>0){

N3=n1+n2;

N1=n2;

N2=n3;

Syatem.out.print(“ “+n3);

printFibo(count-1);

}

}

Public static void main(String[] args){

Int count=15;

System.out.print(n1+” “+n2); //printing 0 and 1

printFibo(count-2); //n-2 bcz 2 numbers are already printed

}

}

o/p:

\*\*\*\*

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Java Strictfp Keyword

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Strictfp class A{} //strictfp applied on class

Strictfp interface M{} //strictfp applied on interface

Class A{

Strictfp void m() {} //strictfp applied on method

}

The strictfp keyword **cannot**be applied on abstract methods, variables or constructors.

Class B{

Strictfp abstract void m(); //illegal combination of modifiers

}

Class B{

Strictfp int data=10; //modifier strictfp not allowed here

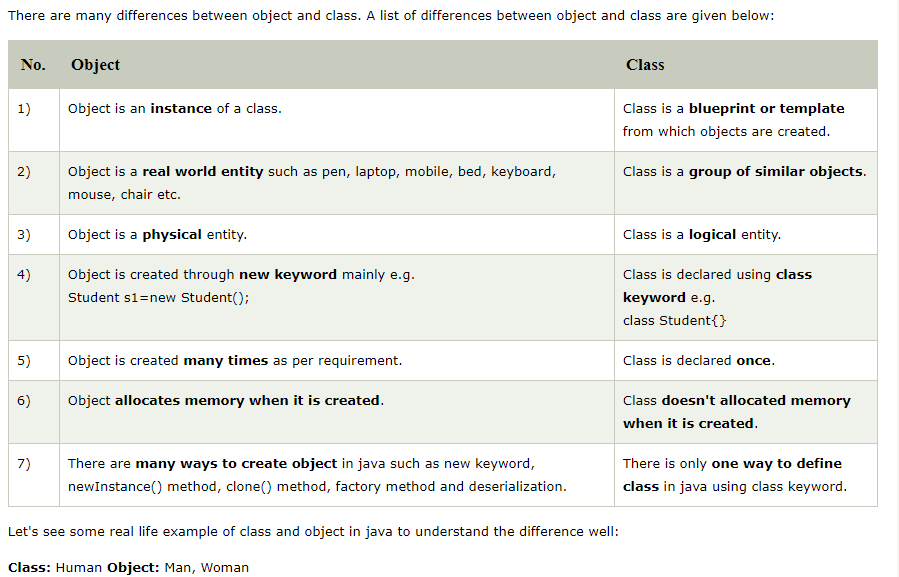
}

Class B{

Strictfp B(){} //modifier strictfp not allowed here

}

Difference b/w object and class?

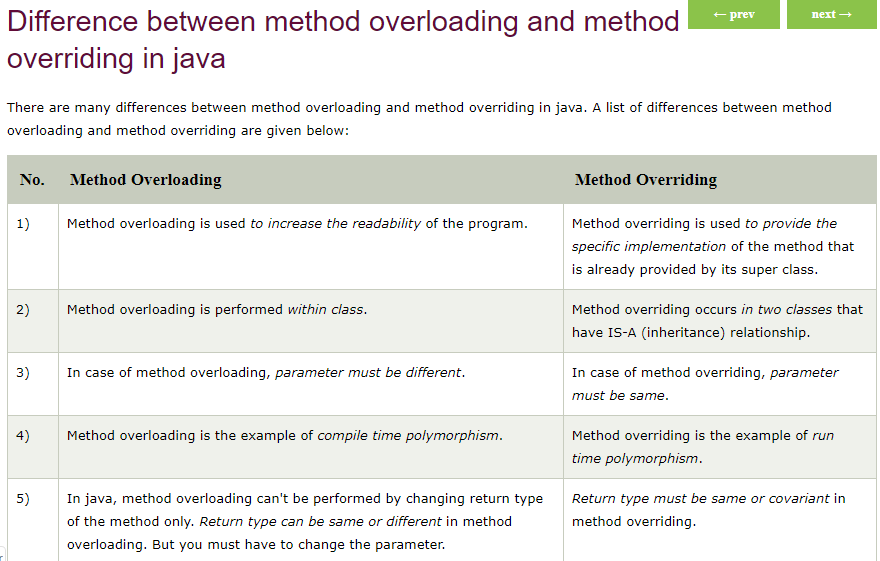


**Class:** Fruit **Object:** Apple, Banana, Mango, Guava wtc.

**Class:** Mobile phone **Object:** iPhone, Samsung, Moto

**Class:** Food **Object:** Pizza, Burger, Samosa

Difference between method overloading and method overriding?



**STRING**

In java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

Or

A String is a combination of multiple characters.

🡪String is final class present in java.lang.package.

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

String s=**new** String(ch);

(or)

String s=”javatpoint”;

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

The java.lang.String class implements Serializable, comparable and charSequence interfaces.



CharSequence Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

How to create a string object?

There are two ways to create String object:

1.By String literal

2.By new keyword

1)String Literal

\*\*\*\*\*\*\*\*\*\*\*\*\*

Java String literal is created by using double quotes.

String s=”welcome”;

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



String s1=”welcome”;

String s2=”welcome”; // It doesn’t create a new instance

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

\*\*🡪String objects are stored in a special memory area known as the “String constant pool”.

Why Java uses the concept of String literal?

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

2)By new Keyword

String s=new String(“welcome”); // create two objects and one reference variable.

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

Immutable String in java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Ex:

\*\*\*

String s=”sai”;

s.concat(“kiran”); //concat() method appends the string at the end

System.out.println(s); //will print sai bcz Strings are immutable objects.

o/p:

\*\*\*

sai

In above program sai cannot be changed but a new object is created with sai kiran. That is why string is known as Immutable objects.

“sai”

“sai Kiran”

s

String constant pool

String s=”Sai”;

S=s.concat(“kiran”);

System.out.println(s);// o/p:🡺Sai Kiran

In this case, s points to the "Sai Kiran". Please notice that still sachin object is not modified.

Why String objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all referes to one object "sai". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

Java String compare

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We can compare string in java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare string in java:

1. By equals() method
2. By = = operator
3. By compareTo() method

1)String compare by equals()method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

Ex-1:

\*\*\*\*\*

String s1=”sai”;

String s2=”sai”;

String s3=new String(“sai”);

String s4=”saikiran”;

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

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

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

Ex-2:

\*\*\*\*

String s1=”saikiran”;

String s2=”SAIKIRAN”;

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

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

2)String compare by == operator

The == operator compares references not values.

String s1=”sai”;

String s2=”sai”;

String s3=new String(“sai”);

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

System.out.println(s1==s3);//false(bcz s3 refers to instance created in non pool)

3)String compare by compareTo() method

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

Suppose s1 and s2 are two string variables. If:

* **s1 == s2**:0
* **s1 > s2** :positive value
* **s1 < s2**  :negative value

String s1=”sai”;

String s2=”sai”;

String s3=”kiran”;

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

System.out.println(s1.compareTo(s3));//8(bcz s1>s3)

System.out.println(s1.compareTo(s2));//-8(bcz s3<s1)

String Concatenation in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

1)String Concentration by +(String concatenation) operator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java string concatenation operator (+) is used to add strings.

Class Test{

Public static void main(String[] args){

String s1=”Sai”+”Kiran”;

System.out.println(s1); //Sai kiran

}

}

The **Java compiler transforms** above code to this:

String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also

Class Test{

Public static void main(String[] args){

String s=20+30+sai+80+90; //50sai8090

System.out.println(S);

}

}

2)String Concation by concat() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The String concat() method concatenates the specified string to the end of current string.

Public String concat(String another)

Ex:

\*\*\*

Class Test{

Psvm(St[] args){

String s1=”sai”;

String s2=”kiran”;

String s3=s1.concat(s2);

System.out.println(s3); //Sai Kiran

}

}

SubString in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

Index start from 0.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):**This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

**Start index**: Inclusive

**End Index:** Exclusive

Java String class methods:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

1)Java String toUpperCase() and toLowerCase() method

2)Java String trim() method

3)Java String startWith() and endsWith() method

4)Java String charAt() method

5)Java String length() method

6)Java String intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

String s=new String(“Saikiran”);

String s2=s.intern();

System.out.println(s2); //Saikiran

7)Java String valueOf() method

The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

Int a=10;

String s=String.valueOf(a);

System.out.println(s+10);//1010

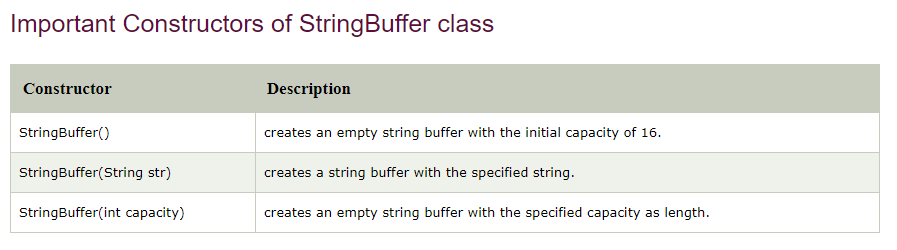
8)Java String replace() method

Java StringBuffer class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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



What is mutable string?

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

1)StringBuffer append() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Class Test{

Psvm(St[] args){

StringBuffer sb=new StringBuffer(“Hello”);

Sb.append(“java”);// now orginal string is changed

Sopl(sb);//Hello java

}

}

2)StringBuffer insert() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Class Test{

Psvm(St[] args){

StringBuffer sb=new StringBuffer(“Hello”);

Sb.insert(1,”java”);// now orginal string is changed

Sopl(sb);//Hjavaello

}

}

3)StringBuffer replace() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Class Test{

Psvm(St[] args){

StringBuffer sb=new StringBuffer(“Hello”);

Sb.replace(1,3,”java”);

Sopl(sb); //HJavalo

}

}

4)StringBuffer delete() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

class StringBufferExample4{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

5)StringBuffer reverse() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

class StringBufferExample5{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

6)StringBuffer capacity() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

class StringBufferExample6{

public static void main(String args[]){

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

}

}

7)StringBuffer ensureCapacity() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

**class** StringBufferExample7{

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

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

System.out.println(sb.capacity());//default 16

sb.append("Hello");

System.out.println(sb.capacity());//now 16

sb.append("java is my favourite language");

System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2

sb.ensureCapacity(10);//now no change

System.out.println(sb.capacity());//now 34

sb.ensureCapacity(50);//now (34\*2)+2

System.out.println(sb.capacity());//now 70

}

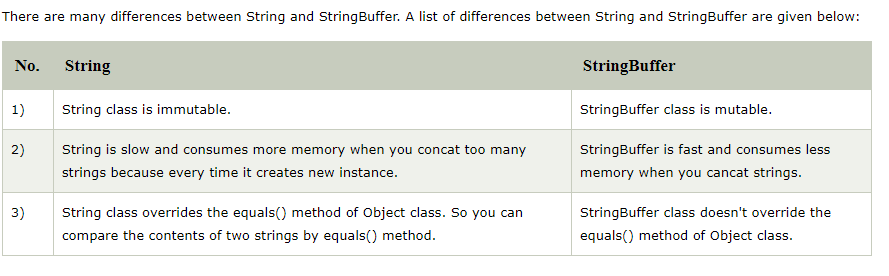
}

Java StringBuilder class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

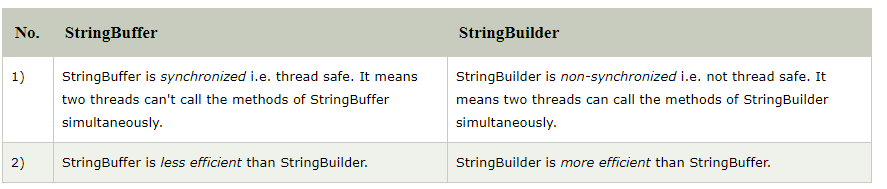
Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Difference b/w String and StringBuffer?



Difference b/w StringBuffer and StringBuilder?

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.



How to create immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

Ex to create Immutable class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

public final class Employee{

final String pancardNumber;

public Employee(String pancardNumber){

this.pancardNumber=pancardNumber;

}

public String getPancardNumber(){

return pancardNumber;

}

}

The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

Java 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. depends on your implementation.

Advantage of java toString() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Ex: without toString() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Student{

 int rollno;

 String name;

 String city;

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

 this.rollno=rollno;

 this.name=name;

 this.city=city;

 }

 public 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()

 }

}

o/p:

\*\*\*\*

Student@1fee6fc

Student@1eed786

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls toString() method, overriding this method will return the specified values.

Ex of java toString() method

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Student{

 int rollno;

 String name;

 String city;

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

 this.rollno=rollno;

 this.name=name;

 this.city=city;

 }

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

  return rollno+" "+name+" "+city;

 }

 public static void main(String args[]){

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

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

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

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

 }

}

o/p:

\*\*\*\*

101 raj lucknow

102 vijay Ghaziabad

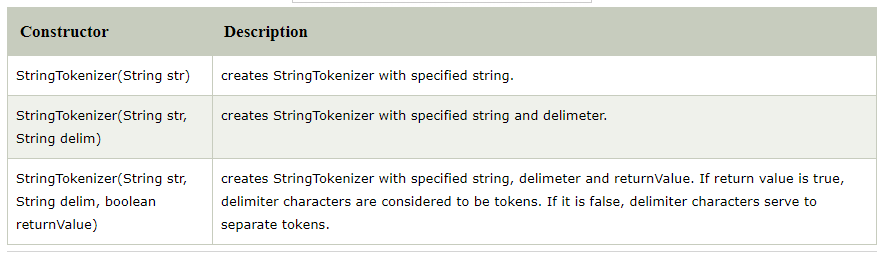
String Tokenizer in java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

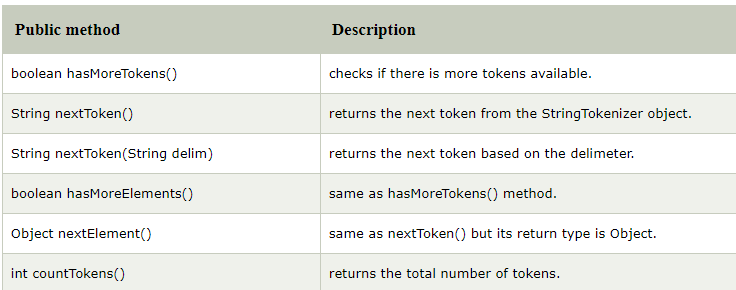
Constructors used in string tokenizer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



Methods in String tokenizer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



Ex of StringTokenizer class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.util.StringTokenizer;

public class Simple{

 public static void main(String args[]){

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

     while (st.hasMoreTokens()) {

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

     }

   }

}

o/p:

\*\*\*

My

Name

Is

Sai

Example of nextToken(String delim) method of StringTokenizer class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.util.\*;

public class Test {

   public static void main(String[] args) {

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

      // printing next token

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

   }

}

o/p:

\*\*\*\*

Next token is : my

Note:

\*\*\*\*\*

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

Reverse String in Java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)By StringBuilder/StringBuffer

Class Reverse{

Public static String reverseString(String str){

StringBuilder sb=new StringBuilder(str);

Sb.reverse();

Return sb.toString();

}

}

Class Test{

Public static void main(String[] args){

System.out.println(reverse.reverseString(“my name is sai”));

System.out.println(reverse.reverseString(“my name is kiran”));

}

}

2)By Reverse Iteration

Class Reverse{

Public static String reverseString(String str){

Char ch[]=str.toCharArray();

String rev=” “;

For(int i=ch.length-1;i>=0;i--){

Rev+=ch[i];

}

Return rev;

}

}

Class Test{

Public static void main(String[] args){

System.out.println(reverse.reverseString(“my name is sai”));

System.out.println(reverse.reverseString(“my name is kiran”));

}

}

**COLLECTIONS**

**\*\*\*\*\*\*\*\*\*\*\***

What is Java Non generic vs. Generic collections?

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type safe so typecasting is not required at runtime.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

ArrayList al=new ArrayList(); 🡺non generic type

ArrayList<String> al=new ArrayList<String>(); 🡺 generic type

Ways to iterate the elements of the collection in java

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.
7. By using Iterator interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Iterator itr=list.iterator(); 🡺al, hm, hs wecan take any thing

While(itr.hasNext()){

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

}

Here also we can take any reference values based on generic or non generic process.

2)For-Each loop

For(String obj:al){

System.out.println(obj);

}

3)Iterating collection through remaining ways

Let's see an example to traverse the ArrayList elements through other ways

Ex:

\*\*\*

import java.util.\*;

class ArrayList4{

 public static void main(String args[]){

    ArrayList<String> list=new ArrayList<String>();//Creating arraylist

           list.add("Ravi");//Adding object in arraylist

           list.add("Vijay");

           list.add("Ravi");

           list.add("Ajay");

           System.out.println("Traversing list through List Iterator:");

 //Here, element iterates in reverse order

              ListIterator<String> list1=list.listIterator(list.size());

              while(list1.hasPrevious())

              {

                  String str=list1.previous();

                  System.out.println(str);

              }

        System.out.println("Traversing list through for loop:");

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

           {

            System.out.println(list.get(i));

           }

        System.out.println("Traversing list through forEach() method:");

  //The forEach() method is a new feature, introduced in Java 8.

            list.forEach(a->{ //Here, we are using lambda expression

                System.out.println(a);

              });

            System.out.println("Traversing list through forEachRemaining() method:");

              Iterator<String> itr=list.iterator();

              itr.forEachRemaining(a-> //Here, we are using lambda expression

              {

            System.out.println(a);

             });

 }

}

o/p:

\*\*\*

**Traversing list through List Iterator:**

Ajay

Ravi

Vijay

Ravi

**Traversing list through for loop:**

Ravi

Vijay

Ravi

Ajay

**Traversing list through forEach() method:**

Ravi

Vijay

Ravi

Ajay

**Traversing list through forEachRemaining() method:**

Ravi

Vijay

Ravi

Ajay

User-defined class objects in Java ArrayList

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let's see an example where we are storing Student class object in an array list.

class Student{

  int rollno;

  String name;

  int age;

  Student(int rollno,String name,int age){

   this.rollno=rollno;

   this.name=name;

   this.age=age;

  }

}

Ex:

\*\*\*

import java.util.\*;

 class ArrayList5{

 public static void main(String args[]){

  //Creating user-defined class objects

  Student s1=new Student(101,"Sonoo",23);

  Student s2=new Student(102,"Ravi",21);

  Student s2=new Student(103,"Hanumat",25);

 //creating arraylist

  ArrayList<Student> al=new ArrayList<Student>();

  al.add(s1);//adding Student class object

  al.add(s2);

  al.add(s3);

  //Getting Iterator

  Iterator itr=al.iterator();

 //traversing elements of ArrayList object

  while(itr.hasNext()){

    Student st=(Student)itr.next();

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

  }

 }

}

o/p:

\*\*\*\*

101 sonoo 23

102 Ravi 21

103 hanumat 25

Java ArrayList Serialization and Deserialization Example:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let's see an example to serialize an ArrayList object and then deserialize it.

Ex:

\*\*\*

import java.io.\*;

import java.util.\*;

 class ArrayList6 {

        public static void main(String [] args)

        {

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

          al.add("Ravi");

          al.add("Vijay");

          al.add("Ajay");

          try

          {

              //Serialization

              FileOutputStream fos=new FileOutputStream("file");

              ObjectOutputStream oos=new ObjectOutputStream(fos);

              oos.writeObject(al);

              fos.close();

              oos.close();

              //Deserialization

              FileInputStream fis=new FileInputStream("file");

              ObjectInputStream ois=new ObjectInputStream(fis);

            ArrayList  list=(ArrayList)ois.readObject();

            System.out.println(list);

          }catch(Exception e)

          {

              System.out.println(e);

          }

       }

    }

o/p:

\*\*\*\*

[ravi, vijay, Ajay]

Example of ArrayList: Book

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.util.\*;

class Book {

int id;

String name, author, publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class ArrayListExample {

public static void main(String[] args) {

 //Creating list of Books

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

//Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc GrawHill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

//Adding Books to list

    list.add(b1);

    list.add(b2);

    list.add(b3);

    //Traversing list

    for(Book b:list){

        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

    }

}

}

o/p:

\*\*\*\*

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

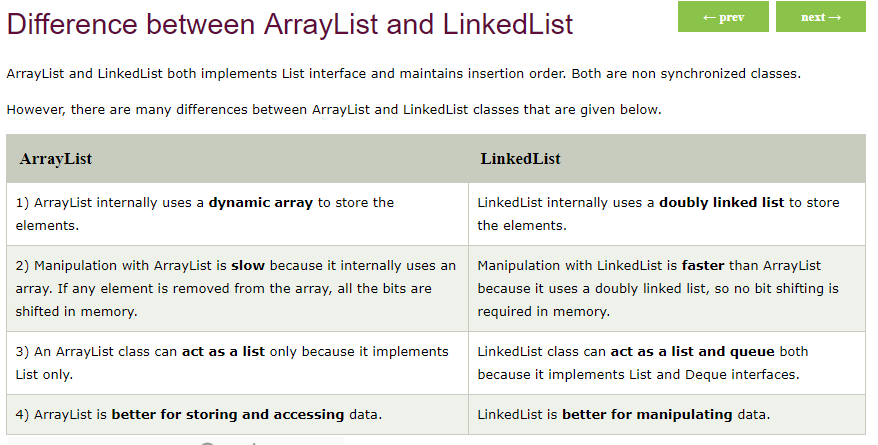
Java LinkedList class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.



Java List Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

List Interface is the subinterface of Collection. It contains index-based methods to insert and delete elements. It is a factory of ListIterator interface.

List interface declaration:

Public interface list<E> extends Collection <E>

Java HashSet

\*\*\*\*\*\*\*\*\*\*\*

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference b/w List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

Java LinkedHashSet class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

Java TreeSet class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

Java Map Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

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

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

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

Map Entry Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Entry is the sub interface 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.

Map Example Non Generic Type

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//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());

    }

}

}

o/p:

\*\*\*\*

1 amit

2 jai

5 rahul

6 amit

Java HashMap class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

* Java HashMap class contains values based on the key.
* Java HashMap class contains only unique keys.
* Java HashMap class may have one null key and multiple null values.
* Java HashMap class is non synchronized.
* Java HashMap class maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Java TreeMap

\*\*\*\*\*\*\*\*\*\*\*

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

Java HashTable class

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Java HashTable Example: getOrDefault()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Test{

Public static void main(String[] args){

HashTable<Integer,String> ht=new HashTable<Integer,String>();

Ht.put(101,”sai”);

Ht.put(102,”kiran”);

Ht.put(103,”tadipatri”);

Ht.put(104,”java”);

//Here, we specify the if and else statements as arguments of the method.

System.out.println(ht.getOrDefault(102,”notfound”));

System.out.println(ht.getOrDefault(105,”notfound”));

}

}

o/p:

\*\*\*

Talanki

Not found

Java HashTable Example:putIfAbsent()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Test{

Public static void main(String[] args){

HashTable<Integer,String> ht=new HashTable<Integer,String>();

Ht.put(101,”sai”);

Ht.put(102,”kiran”);

Ht.put(103,”tadipatri”);

Ht.put(104,”java”);

System.out.println(ht);----------🡪1

//Inserts, as the specified pair is unique

Ht.putIfAbsent(105,”selenium”);

System.out.println(ht);------------🡪2

//Returns the current value, as the specified pair already exist

Ht.putIfAbsent(101,”sai”);

System.out.println(ht);-------------🡪3

}

}

o/p:

\*\*\*\*

{104=java, 103=tadipatri, 102=kiran, 101=sai}------🡪1

{105=selenium, 104=java, 103=tadipatri, 102=kiran, 101=sai}------🡪2

{105=selenium, 104=java, 103=tadipatri, 102=kiran, 101=sai}------🡪3

Difference between HashMap and HashTable?

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

