1**) Java Program to Read The Number From Standard Input?**

class GFG {

public static void main(String[] args)

{

// Declare the variables

int num;

// Input the integer

System.out.println("Enter the integer: ");

// Create Scanner object

Scanner s = new Scanner(System.in);

// Read the next integer from the screen

num = s.nextInt();

// Display the integer

System.out.println("Entered integer is: "+ num);

}

}

**Output**:

Enter the integer: 10

Entered integer is: 10

2) **Enter data using BufferReader**

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Enter the name:");

// Reading data using readLine

String name = reader.readLine();

// Printing the read line

System.out.println(name);

-----

**// Using Scanner for Getting Input from User**

Scanner in = new Scanner(System.in);

String s = in.nextLine();

System.out.println("You entered string " + s);

int a = in.nextInt();

System.out.println("You entered integer " + a);

float b = in.nextFloat();

System.out.println("You entered float " + b);

------------

3) Java Program to Swap Two Numbers

static void swapValuesUsingThirdVariable(int m, int n)

{

int temp = m; m = n; n = temp;

System.out.println("Value of m is " + m+ " and Value of n is " + n);

public static void main(String[] args)

{ // Random integer values

int m = 9, n = 5;

// Calling above function to reverse the numbers

swapValuesUsingThirdVariable(m, n);

swapValuesUsingThirdVariable(10, 20);

swapValuesUsingThirdVariable(20, 30);

}

**Output**:- Value of m is 5 and Value of n is 9

Value of m is 20 and Value of n is 10

Value of m is 30 and Value of n is 20

------------

static void swapValuesWithoutUsingThirdVariable(int m,int n)

{

// is stored in first variable

m = m - n;

// Sum is stored in second variable

n = m + n;

// Difference of 1st from 2nd is replaced in first variable

m = n - m;

// Print numbers

System.out.println("Value of m is " + m + " and Value of n is " + n);

}

main method

swapValuesUsingThirdVariable(m, n);

swapValuesUsingThirdVariable(10, 20);

swapValuesUsingThirdVariable(20, 30);

**Output**: Value of m is 5 and Value of n is 9

Value of m is 20 and Value of n is 10

Value of m is 30 and Value of n is 20

---------------

static void swapValuesWithoutUsingThirdVariable(int m,int n)

{

m = m ^ n;

n = m ^ n;

m = m ^ n;

System.out.println("Value of m is " + m + " and Value of n is " + n);

}

**Output**: Value of m is 5 and Value of n is 9

------

4) Java Program to Check Even or Odd Integers

public static boolean oddreven(int n)

{ if(n % 2 == 0)

{ System.out.println("even Number"); }

else

{ System.out.println("Odd Number"); }

return true;

}

oddreven(99);

**Output**:- Odd Number

---------------------------

public static boolean oddreven1(int n)

{ if((n | 1) > n)

{ System.out.println("even Number"); }

else

{ System.out.println("Odd Number"); }

return true;

}

oddreven1(100);

**Output**:- even Number

------------

public static boolean oddreven1(int n)

{

if ((n & 1) == 1) {

// Print statement

System.out.println("Number is Odd");

}

else {

// Print statement

System.out.println("Number is Even");

}

}

oddreven1(99);

Output:- Number is Odd

----------------

if ((n & 1) == 1) { System.out.println("Number is Odd"); }

else { System.out.println("Number is Even"); }

oddreven1(99);

Output:- Number is Odd

-------------------

5)Java Program to Find the Largest of three Numbers

int x=biggestOfThree(10, 20, 15);

System.out.println("biggest number= "+x );

static int biggestOfThree(int x, int y, int z)

{ return z > (x > y ? x : y) ? z : ((x > y) ? x : y); }

**Output**:-

biggest number= 20

-----------------------------

if (x >= y && x >= z) return x;

else if (y >= x && y >= z) return y;

else return z;

**Output**:-

biggest number= 20

---------------------------

6)

static void prime\_N(int N)

{ // Declaring the variables

int x, y, flg; N=150;

// Printing display message

System.out.println("All the Prime numbers within 1 and " + N + " are:");

// Using for loop for traversing all the numbers from 1 to N

for (x = 1; x <= N; x++) {

// Omit 0 and 1 as they are neither prime nor composite

if (x == 1 || x == 0)

continue;

// Using flag variable to check if x is prime or not

flg = 1;

for (y = 2; y <= x / 2; ++y) {

if (x % y == 0)

{ flg = 0; break; } }

// If flag is 1 then x is prime but if flag is 0 then x is not prime

if (flg == 1) System.out.print(x + " "); } }

**Output**:-

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

------------------------------

7) public static void isLeapYear(int year)

{

// flag to take a non-leap year by default

boolean is\_leap\_year = false;

// If year is divisible by 4

if (year % 4 == 0) {

is\_leap\_year = true;

// To identify whether it is a century year or not

if (year % 100 == 0) {

// Checking if year is divisible by 400 therefore century leap year

if (year % 400 == 0)

is\_leap\_year = true;

else

is\_leap\_year = false;

}

}

// We land here when corresponding if fails If year is not divisible by 4

else

// Flag dealing- Non leap-year

is\_leap\_year = false;

if (!is\_leap\_year)

System.out.println(year + " : Non Leap-year");

else

System.out.println(year + " : Leap-year");

}

**Output**:-

2000 : Leap-year

2008 : Leap-year

-------------------------

8) Java Program to Check Armstrong Number between Two Integers

370 = 3\*3\*3 + 7\*7\*7 + 0

= 27 + 343 + 0

= 370

----------------------------

9) Java Program to Check Whether the Character is Vowel or Consonant

// Function to find whether an input character is vowel or not

static void Vowel\_Or\_Consonant(char y)

{

if (y == 'a' || y == 'e' || y == 'i' || y == 'o' || y == 'u')

System.out.println("It is a Vowel.");

else

System.out.println("It is a Consonant.");

}

Vowel\_Or\_Consonant('b');

Vowel\_Or\_Consonant('u');

Vowel\_Or\_Consonant('E');

Output:-

It is a Consonant.

It is a Vowel.

It is a Consonant.

if (y == 'a' || y == 'e' || y == 'i' || y == 'o'

|| y == 'u' || y == 'A' || y == 'E' || y == 'I'

|| y == 'O' || y == 'U')

------------------------

10) factorial number

static int factorial(int n)

{ if (n == 0)

return 1;

return n \* factorial(n-1);

}

System.out.println(factorial(4));

Output:- 120

-------------------------------

11) Java Program for fibonacci series

int i = 1;

while (i <= n)

{

System.out.print(t1 + " ");

int sum = t1 + t2;

t1 = t2;

t2 = sum;

i++;

}

Output:- First 9 terms of fibonnaci series:

0 1 1 2 3 5 8 13 21

----------------------

12) Java Program for Simple Interest:

P is the principal amount

T is the time and

R is the rate

Simple Interest = (P x T x R)/100

float P = 1, R = 1, T = 1;

/\* Calculate simple interest \*/

float P = 10000, R = 5, T = 5;

System.out.println("Simple interest = " + SI);

Output:- 2500.0

--------------------

13) Java Program for compound interest

P is the principal amount

T is the time and

R is the rate

Compound Interest = P(1 + R/100)r

double principle = 10000, rate = 10.25, time = 5;

double CI = principle \* (Math.pow((1 + rate / 100), time));

System.out.println("Compound Interest is "+ CI);

Output:-

Compound Interest is 16288.9

------------------------

14)Java Program to Find the Perimeter of a Rectangle

int length = 10;

int breadth = 20;

int perimeter = 2 \* (length + breadth);

System.out.println("The perimeter of the given rectangle of length "

+ length + " and breadth " + breadth + " = "+ perimeter);

Output:- The perimeter of the given rectangle of length 10 and breadth 20 = 60

----------------------

Java Program to Print Right Triangle Star Pattern

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

int a, b, n=5;

for (a = 0; a <= n; a++) {

for (b = 0; b <= a; b++) {

// printing stars

System.out.print("\* ");

}

System.out.println();

}

---------------------

15) Java Program to Print Right Triangle Star Pattern

int number = 5;

int i = number, j;

while (i > 0)

{

j = 0;

while (j++ < number - i)

{

System.out.print(" ");

}

j = 0;

// Inner loop Condition check

while (j++ < (i \* 2) - 1)

{

System.out.print("\*");

}

// By now, we reach end of execution for one row so next line

System.out.println();

i--;

}

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

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

\*\*\*\*\*

\*\*\*

\*

----------------------------

16)

Convert String to Double in Java

String str = "2033.12244";

double str1 = Double.parseDouble(str);

**Convert Boolean toString() method in Java with examples**

boolean value = true;

String output = Boolean.toString(value);

Step-2

Boolean b = new Boolean(true);

String output = b.toString();

**Convert Double to String**

1. Using valueOf() method of String class

2. Using toString() method of Double class

int a = 10; int b = 20;

String s1 = String.valueOf(a);

String s2 = String.valueOf(b);

System.out.println(s1+s2);

System.out.println(s1+s2);

double number = 123.456;

String output = String.valueOf(number);

System.out.println(output);

double number = 123.456;

String output = Double.toString(number);

System.out.println(number);

**Convert String to Long**

Using the parseLong() method of the Long class

Using valueOf() method of long class

Using constructor of Long class.

**step-1**

String str = "99999";

long varLong = Long.parseLong(str);

Integer varint=Integer.parseInt(str);

System.out.println("Long - " + varLong);

System.out.println("int - " + varint);

**step-2**

String str = "999999999999";

long varLong = Long.valueOf(str);

System.out.println("Long - " + varLong);

**step-3**

String str = "999999999999";

long num = new Long(str);

System.out.println("Long - " + num);

**Convert Long to String**

**A. Using + operator**

Long varLong = 999999999999L;

String str = varLong+" ";

System.out.println(str);

**B. Using String.valueOf()**

Long varLong = 999999999999L;

String str = String.valueOf(varLong);

System.out.println(str);

**C.Using Long.toString()**

Long varLong = 999999999999L;

String str = Long.toString(varLong);

System.out.println(str);

**D. new Long(long l)**

Long varLong = 999999999999L;

String str = new Long(varLong).toString();

System.out.println(str);

**E. Using String.format()**

Long varLong = 999999999999L;

String str = String.format("%d", varLong);

System.out.println(str);

**F. Using StringBuilder, StringBuffer object**

long varLong = 9999999L;

String str = new StringBuilder().append(varLong).toString();

System.out.println(str);

**Convert Char to Int**

**Way 1: Using ASCII values**

char ch='a';

int i3 = ch - '0'; //49

**Way 2: Using valueOf() method of String class**

char ch = '3';

int a = Integer.parseInt(String.valueOf(ch)); //3

**Way 3: Using getNumericValue() method of Character Class**

char ch = '3';

int a = Character.getNumericValue(ch); //3

**17) Java Program For Int to Char Conversion**

* Using switch case (naive Approach)
* Using concept of typecasting (Naive Approach)
* Using modulo operator(Optimal approach)

String s=12345 + " ";

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

// Switch case Reading digits one by one using charAt() method

switch (s.charAt(i)) {

// Case 1

case '1':

System.out.print("one ");

// Break statement to hault

// normal execution of the program

break;

// Case 2

case '2':

System.out.print("two ");

break;

---------------

case '0':

System.out.print("zero ");

break;

// Default case

default:

System.out.print("InValid ");

break;

}

**output**:- one two three four five

Method 2: Using concept of Type-casting

int i = 97;

char ch = (char)i;

System.out.println(ch);

Output:- a

-------------

Classes and Objects in Java

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components

1. Modifiers: A class can be public or has default access (Refer this for details).

2. Class keyword: class keyword is used to create a class.

3. Class name: The name should begin with an initial letter (capitalized by convention).

4. Superclass(if any): The name of the class’s parent (superclass), if any, preceded by the 5. keyword extends. A class can only extend (subclass) one parent.

5. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

6. Body: The class body is surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

Object

It is a basic unit of Object-Oriented Programming and represents real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

State: It is represented by attributes of an object. It also reflects the properties of an object.

Behavior: It is represented by methods of an object. It also reflects the response of an object with other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

Ways to create an object of a class

There are four ways to create objects in java. Strictly speaking there is only one way(by using new keyword), and the rest internally use new keyword.

Using new keyword: It is the most common and general way to create an object in java. Example:

// creating object of class Test

Test t = new Test();

Using Class.forName(String className) method: There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give a fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.

// creating object of public class Test

// consider class Test present in com.p1 package

Test obj = (Test)Class.forName("com.p1.Test").newInstance();

Using clone() method: clone() method is present in Object class. It creates and returns a copy of the object.

// creating object of class Test

Test t1 = new Test();

// creating clone of above object

Test t2 = (Test)t1.clone();

Deserialization: De-serialization is a technique of reading an object from the saved state in a file. Refer Serialization/De-Serialization in java

FileInputStream file = new FileInputStream(filename);

ObjectInputStream in = new ObjectInputStream(file);

Object obj = in.readObject();

Creating multiple objects by one type only (A good practice)

In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, the wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java. Example:

Test test = new Test();

test = new Test();

In inheritance system, we use parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using same referenced variable. Example:

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

public class Test

{

// using Dog object

Animal obj = new Dog();

// using Cat object

obj = new Cat();

}

Anonymous objects

Anonymous objects are objects that are instantiated but are not stored in a reference variable.

They are used for immediate method calling.

They will be destroyed after method calling.

They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).

In the example below, when a key is button(referred by the btn) is pressed, we are simply creating anonymous object of EventHandler class for just calling handle method.

btn.setOnAction(new EventHandler()

{

public void handle(ActionEvent event)

{

System.out.println("Hello World!");

}

});

This article is contributed by Gaurav Miglani. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

----------------

Singleton Class in Java

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

Remember the key points while defining class as a singleton class that is while designing a singleton class:

Make a constructor private.

Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

Purpose of Singleton Class

The primary purpose of a Singleton class is to restrict the limit of the number of object creation to only one. This often ensures that there is access control to resources, for example, socket or database connection.

The memory space wastage does not occur with the use of the singleton class because it restricts the instance creation. As the object creation will take place only once instead of creating it each time a new request is made.

We can use this single object repeatedly as per the requirements. This is the reason why the multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings, and much more.

For example, there is a license with us, and we have only one database connection or suppose if our JDBC driver does not allow us to do multithreading, then Singleton class comes into the picture and makes sure that at a time, only a single connection or a single thread can access the connection.

How to Design/Create a Singleton Class in Java?

To create a singleton class, we must follow the steps, given below:

1. Ensure that only one instance of the class exists.

2. Provide global access to that instance by

Declaring all constructors of the class to be private.

Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.

The instance is stored as a private static variable.

Example of singleton classes is Runtime class, Action Servlet, Service Locator. Private constructors and factory methods are also an example of the singleton class.

Difference between Normal Class and Singleton Class

We can distinguish a Singleton class from the usual classes with respect to the process of instantiating the object of the class. To instantiate a normal class, we use a java constructor. On the other hand, to instantiate a singleton class, we use the getInstance() method.

The other difference is that a normal class vanishes at the end of the lifecycle of the application while the singleton class does not destroy with the completion of an application.

Forms of Singleton Class Pattern

There are two forms of singleton design pattern, which are:

Early Instantiation: The object creation takes place at the load time.

Lazy Instantiation: The object creation is done according to the requirement.

Implementation: Let us brief how the singleton class varies from the normal class in java. Here the difference is in terms of instantiation as for normal class we use constructor, whereas for singleton class we use getInstance() method which we will be peeking out in example 1 as depicted below. In general, in order to avoid confusion, we may also use the class name as method name while defining this method which will be as depicted in example 2 below as follows.

Singleton class

Singleton x = Singleton.getInstance();

Singleton y = Singleton.getInstance();

System.out.println("Hashcode of x is " + x.hashCode());

System.out.println("Hashcode of y is " + y.hashCode());

// Condition check

if (x == y)

{S.o.p("Three objects point to the same memory location on the heap i.e, to the same object");

} else {

System.out.println("Three objects DO NOT point to the same memory location on the heap"); }

private static Singleton single\_instance = null;

public String s;

private Singleton()

{

s = "Hello I am a string part of sinfleton1 class";

}

public static Singleton getInstance()

{

if (single\_instance == null)

single\_instance = new Singleton();

System.out.println(single\_instance.s);

return single\_instance;

}

Output:-

Hello I am a string part of sinfleton1 class

Hello I am a string part of sinfleton1 class

Hashcode of y is 1510467688

Hashcode of z is 1510467688

Three objects point to the same memory location on the heap i.e, to the same object

Interfaces in Java

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a class. A Java interface contains 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 the 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.

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

Interfaces specify what a class must do and not how. It is the blueprint of the class.

An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use an Interface?

It is used to achieve total abstraction.

Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.

It is also used to achieve loose coupling.

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

Difference Between Class and Interface

The major differences between a class and an interface are:

S. No. Class Interface

1. In class, you can instantiate variables and create an object.

2. In an interface, you can’t instantiate variables and create an object.

1. Class can contain concrete(with implementation) methods

2. The interface cannot contain concrete(with implementation) methods

1. The access specifiers used with classes are private, protected, and public.

2. In Interface only one specifier is used- Public.

Example:-

Real-World Example: Let’s consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

The advantages of using interfaces in Java are as follows:

Without bothering about the implementation part, we can achieve the security of the implementation.

In Java, multiple inheritance is not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

New Features Added in Interfaces in JDK 8

1. Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

// Java program to show that interfaces can

// have methods from JDK 1.8 onwards

interface In1

{

final int a = 10;

default void display()

{

System.out.println("hello");

}

}

TestClass t = new TestClass();

t.display();

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object. Note: these methods are not inherited.

interface In1{ final int a = 10; static void display(){S.o.p("hello"); } }

In1.display();

------------

package com.vinod.search;

public class LinearSearch1

{

static int search(int arr[], int n, int x)

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

if (arr[i] == x) return i; }

return -1;

}

public static void main(String[] args)

{

int[] arr = { 3, 4, 1, 7, 5 };

int n = arr.length;

int x = 4;

int index = search(arr, n, x);

if (index == -1)

System.out.println("Element is not present in the array");

else

System.out.println("Element found at position "); } }

Output:- Element found at position

ALGORITHM :

Step 1: Start

Step 2: Declare an array and search element as key.

Step 3: Traverse the array until the number is found.

Step 4: If the key element is found, return the index position of the array element

Step 5: If the key element is not found, return -1

Step 6: Stop.

BEST CASE COMPLEXITY

In linear search, the best-case occurs when the search element is present at the first location of the array. So the best-case time complexity of the linear search is o(1).

The best-case time complexity of the linear search is o(1).

AVERAGE CASE COMPLEXITY

In linear search average case occurs when the search element is present at the random location of the array.so the average case time complexity of the linear search is o(1).

The average case time complexity of the linear search is o(n).

WORST-CASE COMPLEXITY

In linear search, the worst case occurs when the search element is present at the last location of the array So the worst-case time complexity of the linear search is o(1).In the worst case in if the search element is not present in the given array then we need to traverse the entire array to search element. So the worst-case time complexity of the linear search is o(n).

The worst-case time complexity of the linear search is o(n)

SPACE COMPLEXITY:

The SPACE complexity of the linear search is o(1)

Linear Search Applications

we use linear search in following things:

• for search item in the smaller array.

• For fast searching

The time complexity of the above algorithm is O(n). Please refer complete article on Linear Search for more details!

-------------------------

18) BinarySearch

Binary search is one of the searching techniques applied when the input is sorted as here we are focusing on finding the middle element that acts as a reference frame whether to go left or right to it as the elements are already sorted.

int binarySearch(int arr[], int l, int r, int x)

{

if (r>=l)

{

System.out.print(" l= "+l+" r= "+r+" x= "+x);

int mid = l + (r - l)/2;

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

// If the element is present at the middle itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then it can only be present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

// Else the element can only be present in right subarray

return binarySearch(arr, mid+1, r, x);

}

// We reach here when element is not present in array

return -1;

}

// Driver method to test above

public static void main(String args[])

{

BinarySearch1 ob = new BinarySearch1();

int arr[] = {1,2,3,4,5,6,7,8,9,10,13,15,18,20};

int n = arr.length;

int x = 4;

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

int result = ob.binarySearch(arr,0,n-1,x);

if (result == -1)

System.out.println("Element not present");

else

System.out.println("Element found at index " + result);

}

There are two ways to do a binary search in Java

Arrays.binarysearch

Collections.binarysearch

Type 1: Arrays.binarysearch()

It works for arrays which can be of primitive data type also.

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Declaring an integer array

int arr[] = { 10, 20, 15, 22, 35 };

// Sorting the above array

// using sort() method od Arrays class

Arrays.sort(arr);

int key = 22;//40

int res = Arrays.binarySearch(arr, key);

if (res >= 0)

System.out.println(

key + " found at index = " + res);

else

System.out.println(key + " Not found");

}

}

Output:-

22 found at index = 3

40 Not found

Type 2: Collections.binarysearch()

It works for objects Collections like ArrayList and LinkedList.

// Creating an empty ArrayList of integer type

List<Integer> al = new ArrayList<Integer>();

// Populating the Arraylist

al.add(1);

al.add(2);

al.add(3);

al.add(10);

al.add(20);

// 10 is present at index 3

int key = 10; //15

int res = Collections.binarySearch(al, key);

if (res >= 0)

System.out.println(

key + " found at index = " + res);

else

System.out.println(key + " Not found");

} }

Output:-

10 found at index = 3

15 Not found

------------------String------------------

20) **Java Program to Get a Character From the Given String**

String str = "Geeks Gor Geeks";

StringBuilder string1 = new StringBuilder(str);

string1.setCharAt(6, 'F');

System.out.println(string1);

Output:-

Geeks For Geeks

-------------

22) **Reverse a string in Java**

char ch;

String str = "Geeks Gor Geeks",nstr="";

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

{

ch= str.charAt(i); //extracts each character

nstr= ch+nstr; //adds each character in front of the existing string

}

System.out.println("Reversed word: "+ nstr);

Output: Reversed word: skeeG roG skeeG

method-2

byte[] strAsByteArray = str.getBytes();

byte[] result = new byte[strAsByteArray.length];

// Store result in reverse order into the result byte[]

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

result[i] = strAsByteArray[strAsByteArray.length - i - 1];

System.out.println(new String(result));

}

System.out.println(new String(result));

output:-

skeeG roG skeeG

method-3

String input = "Geeks for Geeks";

StringBuilder input1 = new StringBuilder();

// append a string into StringBuilder input1

input1.append(input);

// reverse StringBuilder input1

input1.reverse();

// print reversed String

System.out.println(input1);

output:-skeeG rof skeeG

-------------

23) **Reverse a String using Stack**

public static String ReverseString(String str)

{

char[] reverseString = new char[str.length()];

// Declare a stack of type Character

Stack<Character> stack = new Stack<Character>();

// Traverse the String and push the character one by one into the Stack

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

// push the character into the Stack

stack.push(str.charAt(i));

}

// Now Pop the Characters from the stack until it becomes empty

int i = 0;

while (!stack.isEmpty()) {

// popping element until stack become empty get the character from the top of the stack

reverseString[i++] = stack.pop();

}//while

// return string object

return new String(reverseString);

}

// Driver code

public static void main(String[] args)

{

String str1 = "GeeksForGeeks";

System.out.println(str1 + " <- Reverse -> " + ReverseString(str1));

String str2 = "Hello World";

System.out.println(str2 + " <- Reverse -> " + ReverseString(str2));

}

output:-

GeeksForGeeks <- Reverse -> skeeGroFskeeG

Hello World <- Reverse -> dlroW olleH

**24) Sort a String in Java (2 different ways)**

// Custom string input

String str = "geeksforgeeks";

char arr[] = str.toCharArray();

char temp;

int i = 0;

while (i < arr.length) {

int j = i + 1;

while (j < arr.length) {

if (arr[j] < arr[i]) {

// Comparing the characters one by one

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}//if

j += 1; //j++; same

}//while j

i ++; //i+=1; same

}//while i

System.out.println(arr);

output:- eeeefggkkorss

Method-2

public static String sortString(String inputString)

{

// Converting input string to character array

char tempArray[] = inputString.toCharArray();

// Sorting temp array using

Arrays.sort(tempArray);

// Returning new sorted string

return new String(tempArray);

}

System.out.println(sortString("hai i am vinod"));

output:-aadhiiimnov

------------------

**25) Print first letter of each word in a string using regex**

String s1 = "Geeks for Geeks";

String s2 = "A Computer Science Portal for Geeks";

Pattern p = Pattern.compile("\\b[a-zA-Z]");

Matcher m1 = p.matcher(s1);

Matcher m2 = p.matcher(s2);

System.out.println("First letter of each word from string \"" + s1 + "\" : ");

while (m1.find())

System.out.print(m1.group());

output:-

First letter of each word from string "Geeks for Geeks" : GfG

------------------

**26) Remove Leading Zeros From String in Java**

String str = "00000123569"; int i = 0;

while (i < str.length() && str.charAt(i) == '0')

i++;

// Converting string into StringBuffer object as strings are immutable

StringBuffer sb = new StringBuffer(str);

// The StringBuffer replace function removes i characters from given index (0 here)

sb.replace(0, i, "");

// Printing leading zeros inside string

System.out.println(sb);

output:- 123569

----------------

**27) Compare two Strings in Java**

String string1 = new String("Geeksforgeeks");

String string2 = new String("Practice");

S.o.p("Comparing " + string1 + " and " + string2+ " : " + string1.equals(string2));

S.o.p("Comparing " + string1 + " and " + string2+ " : " + string1.equalsIgnoreCase(string2));

S.o.p("Comparing " + string1 + " and " + string2+ " : " + Objects.equals(string1, string2));

Using String.compareTo() :

Syntax:

int str1.compareTo(String str2)

Working:

It compares and returns the following values as follows:

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value.

// Comparing for String 1 < String 2

S.o.p("Comparing " + string1 + " and " + string2+ " : " + string1.compareTo(string2));

String s1 = new String("HELLO");

String s2 = new String("HELLO");

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

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

-----------------

**28) Java program to print Even length words in a String**

String s = "i am Geeks for Geeksq and a Geek";

// Splits Str into all possible tokens

for (String word : s.split(" "))

// if length is even

if (word.length() % 2 == 0)

// Print the word

System.out.println(word);

Output:- am Geeksq Geek

----------------

Initializing a List in Java

The Java.util.List is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

List a = new ArrayList();

List b = new LinkedList();

List c = new Vector();

List d = new Stack();

**1. Using List.add() method**

Since list is an interface, one can’t directly instantiate it. However, one can create objects of those classes which have implemented this interface and instantiate them.

Few classes which have implemented the List interface are Stack, ArrayList, LinkedList, Vector etc.

Syntax:

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

List<Integer> llist=new LinkedList<Integer>();

List<Integer> stack=new Stack<Integer>();

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

list.add(1);

list.add(3);

System.out.println("ArrayList : " + list.toString());

// For LinkedList

List<Integer> llist = new LinkedList<Integer>();

llist.add(2);

llist.add(4);

System.out.println("LinkedList : " + llist.toString());

// For Stack

List<Integer> stack = new Stack<Integer>();

stack.add(3);

stack.add(1);

System.out.println("Stack : " + stack.toString());

List<Integer> list=new ArrayList<Integer>(){{add(1);add(2);add(3); }};

System.out.println(list);

ArrayList : [1, 3]

LinkedList : [2, 4]

Stack : [3, 1]

[1,2,3]

**2. Using Arrays.asList()**

Creating Immutable List

Arrays.asList() creates an immutable list from an array. Hence it can be used to instantiate a list with an array.

List<Integer> list=Arrays.asList(1, 2, 3);

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

output:- List : [1, 2, 3]

**Creating Mutable List**

List<Integer> list=new ArrayList<>(Arrays.asList(1, 2, 3));

**3.Using Collections class methods**

There are various methods in Collections class that can be used to instantiate a list. They are:

**3.1 Using Collections.addAll()**

Collections class has a static method addAll() which can be used to initialize a list. Collections.addAll() take in any number of elements after it is specified with the Collection in which the elements are to be inserted.

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

Collections.addAll(list, 1, 2, 3, 4);

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

output:- List : [1, 2, 3, 4]

**3.2 Using Collections.unmodifiableList()**

Collections.unmodifiableList() returns a list which can’t be altered i.e. it can neither add or delete an element. Any attempt to modify the list will result in an UnsupportedOperationExample.

List<Integer> list = Collections.unmodifiableList(Arrays.asList(1, 2, 3));

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

output:- List : [1, 2, 3]

**3.3 Using Collections.singletonList()**

Collections.singletonList() returns an immutable list consisting of one element only.

List<Integer> list = Collections.singletonList(2);

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

output:- List : [2]

**3.4 Using Java 8 Stream**

With the introduction of Stream and functional programming in Java 8, now one can construct any stream of objects and then collect them as a list.

List<Integer> list1 = Stream.of(1, 2, 3).collect(Collectors.toList());

System.out.println("List using Syntax 1: " + list1.toString());

List using Syntax 1: [1, 2, 3]

List<Integer> list2 = Stream.of(3, 2, 1)

.collect(Collectors.toCollection(ArrayList::new));

System.out.println("List using Syntax 2: "+ list2.toString());

output:- List using Syntax 2: [3, 2, 1]

List<Integer> list3 = Stream.of(1, 2, 3, 4)

.collect(Collectors.collectingAndThen(Collectors.toList(),

Collections::unmodifiableList));

System.out.println("List using Syntax 3: "+ list3.toString());

output:- List using Syntax 3: [1, 2, 3, 4]

**3.5 Using Java 9 List.of()**

Java 9 introduced List.of() method which takes in any number of arguments and constructs a compact and unmodifiable list out of them.

List<Integer> unmodifiableList = List.of(1, 2, 3);

System.out.println("List : "+ unmodifiableList.toString());

output:- [1, 2, 3]

----------------

**How to Find a Sublist in a List in Java?**

List in Java contains index-based methods. This enables us to maintain the order collection. So this enables us to search, insert, delete, and even update the elements. This enables us to store multiple copies of the same element already existing on our list. Also, in addition, null elements are allowed to be a part of the List.

We access the list through their index numbers called Interface that we won’t be discussing here.

Types of List

* ArrayList
* LinkedList
* Stack
* Vector

ArrayList is used where elements to be inserted are known because there is no flexibility once we have declared the ArrayList but is frequently used as operation over elements are much faster and the good part of Arraylist is we can directly access the element through ArrayList interface.

**The syntax for ArrayList :**

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

LinkedList is preferred over Arraylist if we want a flexible list with no size constraint and operations over elements are quite slower.

LinkedList<E> extends AbstractList<E> implements List<E>, Deque<E> ;

Vector method is similar to Arraylist just Vector has an edge over ArrayList because all elements in vectors are synchronized and are only useful if making multithreaded applications. So, in practice vector class isn‘t used more frequently anymore.

Vector object= new vector(datatype parameter1, datatype parameter2, ...., datatype parameterN)

**Sublist is a portion of List**

The subList() method of java.util.ArrayList class is used to return a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa. The returned list supports all the optional list operations.

public List subList(int fromIndex, int toIndex)

fromIndex: low endpoint (inclusive) of the subList

toIndex: high endpoint (exclusive) of the subList

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

arrlist.add(1); arrlist.add(4); arrlist.add(9); arrlist.add(25); arrlist.add(36);

System.out.println("Original arrlist: "+ arrlist);

List<Integer> arrlist2 = arrlist.subList(2, 4);

System.out.println("Sublist of arrlist: "+ arrlist2);

output

Original arrlist: [1, 4, 9, 25, 36]

Sublist of arrlist: [9, 25]

-----------------

**Min and Max in a List in Java**

public static Integer findMin(List<Integer> list)

{

if (list == null || list.size() == 0) { return Integer.MAX\_VALUE; }

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

Collections.sort(sortedlist);

return sortedlist.get(0);

}

public static Integer findMax(List<Integer> list)

{

if (list == null || list.size() == 0) { return Integer.MIN\_VALUE; }

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

Collections.sort(sortedlist);

return sortedlist.get(sortedlist.size() - 1);

}

public static void main(String[] args)

{

// create an ArrayList Object list

List<Integer> list = new ArrayList();

list.add(44); list.add(11); list.add(22); list.add(75); list.add(07); list.add(33);

System.out.println("Min: " + findMin(list));

System.out.println("Max: " + findMax(list));

}

output:-Min: 7 Max: 75

**Method-2**

List<Integer> list = new ArrayList();

list.add(44); list.add(11); list.add(22); list.add(75); list.add(07); list.add(33);

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

Collections.sort(sortedlist);

int min= Collections.min(sortedlist);

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

int max= Collections.max(sortedlist);

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

output:- min=7 max=75

--------------------------------------------

**How to remove a SubList from a List in Java**

Given a list in Java, the task is to remove all the elements in the sublist whose index is between fromIndex, inclusive, and toIndex, exclusive. The range of the index is defined by the user.

Method 1: Using subList() and clear() method

List.subList(int fromIndex, int toIndex).clear()

AbstractList<String> list = new LinkedList<String>();

list.add("GFG"); list.add("for");list.add("Geeks");list.add("computer");list.add("portal");

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

list.subList(1, 3).clear();

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

**output**:-Original List: [GFG, for, Geeks, computer, portal]

Final List: [GFG, computer, portal]

**Method 2: Using removeRange() method**

List.removeRange(int fromIndex, int toIndex)

public class GFG extends ArrayList<Integer> {

public static void main(String[] args)

{ GFG arr = new GFG(); arr.add(1); arr.add(2); arr.add(3); arr.add(4); arr.add(5);

arr.add(6); arr.add(7); arr.add(8);

System.out.println("Original List: "+ arr);

arr.removeRange(2, 4); System.out.println("Final List: "+ arr);

Original List: [1, 2, 3, 4, 5, 6, 7, 8]

Final List: [1, 2, 5, 6, 7, 8]

--------------------------------------

**How to Remove Duplicates from ArrayList in Java**

**Method 1:- Using Iterator**

Get the ArrayList with duplicate values.

Create another ArrayList.

Traverse through the first arraylist and store the first appearance of each element into the second arraylist using contains() method.

The second ArrayList contains the elements with duplicates removed.

public class GFG{

ArrayList<Integer>

list = new ArrayList<>(Arrays.asList(1, 10, 1, 2, 2, 3, 3, 10, 3, 4, 5, 5));

System.out.println("ArrayList with duplicates: " + list);

ArrayList<Integer> newList = removeDuplicates(list);

System.out.println("ArrayList with duplicates removed: " + newList);

public static <T> ArrayList<T> removeDuplicates(ArrayList<T> list)

{ ArrayList<T> newList = new ArrayList<T>();

for (T element : list) {

if (!newList.contains(element)) { newList.add(element); } }

return newList; }

**output**:-ArrayList with duplicates: [1, 10, 1, 2, 2, 3, 3, 10, 3, 4, 5, 5]

ArrayList with duplicates removed: [1, 10, 2, 3, 4, 5]

**Method 2:- Using LinkedHashSet**

A better way (both time complexity and ease of implementation wise) is to remove duplicates from an ArrayList is to convert it into a Set that does not allow duplicates. Hence LinkedHashSet is the best option available as this do not allows duplicates as well it preserves the insertion order.

List<Integer> list = new ArrayList(Arrays.asList(1, 10, 1, 2, 2, 3, 3, 10, 3, 4, 5, 5));

Set<Integer> set = new LinkedHashSet();

set.addAll(list);

list.clear();

list.addAll(set);

System.out.println(list);

output:- [1, 10, 2, 3, 4, 5]

**Method 3:- Using Java 8 Stream.distinct()**

You can use the distinct() method from the Stream API. The distinct() method return a new Stream without duplicates elements based on the result returned by equals() method, which can be used for further processing. The actual processing of Stream pipeline starts only after calling terminal methods like forEach() or collect().

List<Integer> list = new ArrayList(Arrays.asList(1, 10, 1, 2, 2, 3, 3, 10, 3, 4, 5, 5));

List<Integer> newList = list.stream().distinct().collect(Collectors.toList());

System.out.println("ArrayList with duplicates removed: "+ newList);

output:-

ArrayList with duplicates removed: [1, 10, 2, 3, 4, 5]

-------------------------

**How to sort an ArrayList in Ascending Order in Java**

Approach: An ArrayList can be Sorted by using the sort() method of the Collections Class in Java. This sort() method takes the collection to be sorted as the parameter and returns a Collection sorted in the Ascending Order by default.

Collections.sort(ArrayList);

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

list.add("Geeks"); list.add("For"); list.add("ForGeeks"); list.add("GeeksForGeeks");

list.add("A computer portal");

System.out.println("Unsorted ArrayList: "+ list);

Collections.sort(list);

System.out.println("Sorted ArrayList "+ "in Ascending order : " + list);

**output**:-

Unsorted ArrayList: [Geeks, For, ForGeeks, GeeksForGeeks, A computer portal]

Sorted ArrayList in Ascending order : [A computer portal, For, ForGeeks, Geeks, GeeksForGeeks]

----------------------------

**Get first and last elements from ArrayList in Java**

List<Integer> al = new ArrayList<Integer>();

al.add(3); al.add(1); al.add(4); al.add(5); al.add(2);

if (list != null && !list.isEmpty()) {

System.out.println("First element is: "+ list.get(0));

System.out.println("Last element is: " + list.get(list.size() - 1));

**output:-**

First element is: 3

Last element is: 2

----------------------------

**Convert a List of String to a comma separated String in Java**

List<String> list = new ArrayList<>(Arrays.asList("Geeks","ForGeeks","GeeksForGeeks"));

String string = String.join(", ", list);

System.out.println("Comma separated String: " + string);

------------------

How to Add Element at First and Last Position of LinkedList in Java?

LinkedList<String> linkedList= new LinkedList<String>();

linkedList.add("e");

linkedList.add("e");

linkedList.add("k");

linkedList.addFirst("G");

linkedList.addLast("s"); //// Inserting at last position

System.out.println("Updated Linked list: "+ linkedList); // Print the updated LinkedList

**output**:- Updated Linked list: [G, e, e, k, s]

----------------------

**Find common elements in two ArrayLists in Java**

Given two ArrayLists, the task is to print all common elements in both the ArrayLists in Java .

Using Collections.retainAll() method

Collections1.retainAll(Collections2)

This method keeps only the common elements of both Collection in Collection1.

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

list1.add("Hii"); list1.add("Geeks"); list1.add("for"); list1.add("Geeks");

ArrayList<String> list2 = new ArrayList<String>();

list2.add("Hii"); list2.add("Geeks"); list2.add("Gaurav");

list1.retainAll(list2);

System.out.println("Common elements: "+ list1);

**output**:-

Common elements: [Hii, Geeks, Geeks]

------------------------------

Remove repeated elements from ArrayList in Java

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

list1.add("Hii"); list1.add("Geeks"); list1.add("for"); list1.add("Geeks");

List<String> gfg1 = list1.stream()

.distinct().collect(Collectors.toList());

System.out.println("Modified List : " + gfg1);

**output**:- Modified List : [Geeks, for]

-----------------------------------------------------

**Exceptions in Java**

-----------------------------------------------------

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

* Major reasons why an exception Occurs
* Invalid user input
* Device failure
* Loss of network connection
* Physical limitations (out of disk memory)
* Code errors
* Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Let us discuss the most important part which is the differences between Error and Exception that is as follows:

**Error**: An Error indicates a serious problem that a reasonable application should not try to catch.

**Exception**: Exception indicates conditions that a reasonable application might try to catch.

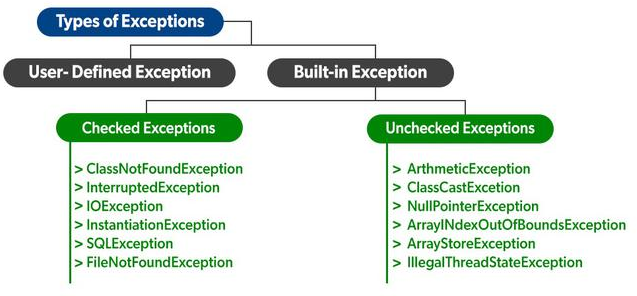
**Exception Hierarchy**

All exception and error types are subclasses of class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



**Types of Exceptions**

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

* Built-in Exceptions
* Checked Exception
* Unchecked Exception
* User-Defined Exceptions

Let us discuss the above-defined listed exception that is as follows:

**A. Built-in Exceptions:**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.

**B. User-Defined Exceptions:**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

The advantages of Exception Handling in Java are as follows:

Provision to Complete Program Execution

Easy Identification of Program Code and Error-Handling Code

Propagation of Errors

Meaningful Error Reporting

Identifying Error Types

Methods to print the Exception information:

**1.printStackTrace()–** This method prints exception information in the format of Name of the exception: description of the exception, stack trace.

int a=5;int b=0;

try{ System.out.println(a/b); }

catch(ArithmeticException e){ e.printStackTrace(); }

output:- java.lang.ArithmeticException: / by zero at GFG.main(File.java:10)

**2.toString()** – This method prints exception information in the format of Name of the exception: description of the exception.

int a=5;int b=0;

try{ System.out.println(a/b); }

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

output:- java.lang.ArithmeticException: / by zero

**3.getMessage()** -This method prints only the description of the exception.

int a=5;int b=0;

try{ System.out.println(a/b); }

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

output:- / by zero

**How Does JVM handle an Exception?**

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called Call Stack. Now the following procedure will happen.

The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.

The run-time system starts searching from the method in which the exception occurred, and proceeds through the call stack in the reverse order in which methods were called.

If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.

If the run-time system searches all the methods on the call stack and couldn’t have found the appropriate handler, then the run-time system handover the Exception Object to the default exception handler, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.

String str = null;

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

output:- java.lang.nullpointerexception

**How Programmer Handles an Exception?**

Customized Exception Handling: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Need for try-catch clause(Customized Exception Handling)

Consider the below program in order to get a better understanding of the try-catch clause.

--------------------------------------------

**Types of Errors in Java with Examples**

Error is an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

The most common errors can be broadly classified as follows:

**1. Run Time Error:**

Run Time errors occur or we can say, are detected during the execution of the program. Sometimes these are discovered when the user enters an invalid data or data which is not relevant. Runtime errors occur when a program does not contain any syntax errors but asks the computer to do something that the computer is unable to reliably do. During compilation, the compiler has no technique to detect these kinds of errors. It is the JVM (Java Virtual Machine) that detects it while the program is running. To handle the error during the run time we can put our error code inside the try block and catch the error inside the catch block.

For example: if the user inputs a data of string format when the computer is expecting an integer, there will be a runtime error. Example 1: Runtime Error caused by dividing by zero

s.o.p(15/0);

output:- java.lang.ArithmeticException:

**Example 2:** Runtime Error caused by Assigning/Retrieving Value from an array using an index which is greater than the size of the array

int arr[] = new int[5];

arr[9] = 250;

output:- java.lang.ArrayIndexOutOfBoundsException: 9

error: ';' expected

System.out.println("Welcome to " + s)

error: ')' expected

for (i = 1/ i <= 5; i++ {

**Java Program to Handle the Exception Hierarchies**

Exceptions are the events that occur due to the programmer error or machine error which causes a disturbance in the normal flow of execution of the program and terminates the program.

**Exception Handling:** The process of dealing with exceptions is known as Exception Handling.

**Hierarchy of Exceptions:** Object class is the parent class of all the classes of java and all the errors or Exceptions objects inherited by throwable class. The throwable class has two subclass Errors and Exception.

**Errors Class:** This Class contains those errors which are difficult to handle. They occur during runtime of a program For e.g. StackOverflowError, OutOfMemoryError etc.

**StackOverflowError:** This error occurs due to an inbound condition in the recursion and using an infinite loop in a program. This cause exceeds the stack memory and causes StackOverflowError in the program.

**Exception class:**

This class contains all the exceptions that can be handled easily There are two subclasses inherited it one is Runtime Exception(unchecked Exception) and checked Exception.

1**.Checked exceptions:** These exceptions are the subclass of the Exception class. These types of exceptions occur during the compile time of the program by the javac. These exceptions can be handled by the try-catch block otherwise the program will give a compilation error. ClassNotFoundException, IOException, SQLException etc are the examples of the checked exceptions.

**I/O Exception:** This Program throw I/O exception because of due FileNotFoundException is a checked exception in Java. Anytime, we want to read a file from the filesystem, Java forces us to handle error situations where the file is not present in the given location.

Assumption: consider myfile.txt file does not exit

**2. Unchecked Exception:**

These types of exceptions occur during the runtime of the program. These are the exceptions that are not checked at a compiled time by the compiler. In Java exceptions under Error and RuntimeException classes are unchecked exceptions, This Exception occurs due to bad programming. Runtime Exceptions like IndexoutOfBoundException, Nullpointer Exception, etc can be handled with the help of Try-Catch Block

(Array)IndexoutOfBoundException: This Exception occurs due to accessing the index greater than and equal to the size of the array length. The program will automatically be terminated after this exception.

Multiple Catch Blocks: Sometimes in a program particular code can throw multiple Exceptions object that can be handled using multiple catch blocks under a single try block

Handling ArithmeticException and IndexoutOfBound Exception using Multiple catch blocks.

--------------

**Java Program to Handle the Exception Methods**

An unlikely event which disrupts the normal flow of the program is known as an Exception. Java Exception Handling is an object-oriented way to handle exceptions. When an error occurs during the execution of the program, an exception object is created which contains the information about the hierarchy of the Exception and other information which is essential for debugging.

**Types of Exceptions:**

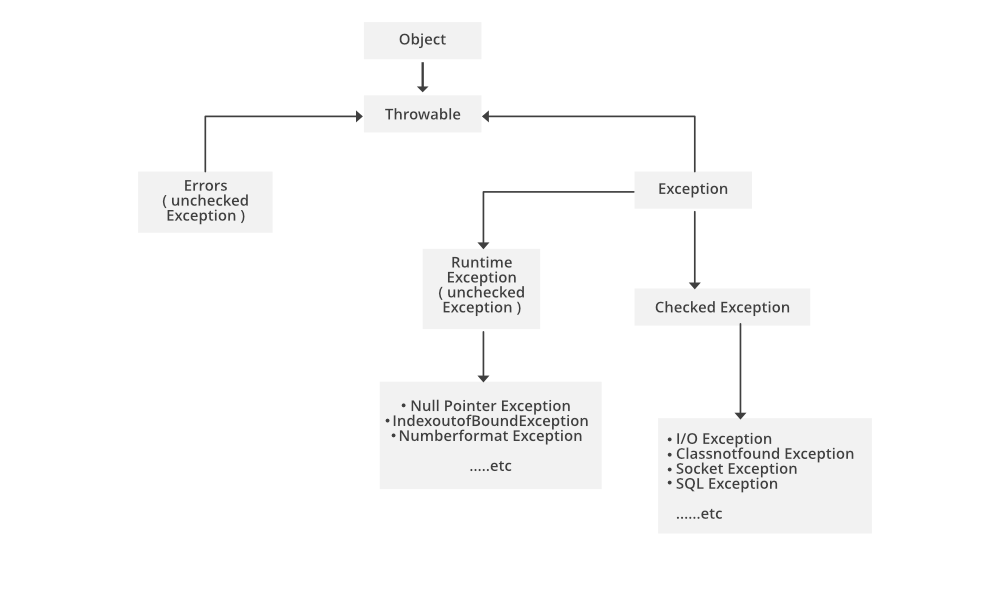
* Checked Exceptions
* Unchecked Exceptions

--------------

**Java Program to Handle Checked Exception**

Checked exceptions are the subclass of the Exception class. These types of exceptions need to be handled during the compile time of the program. These exceptions can be handled by the try-catch block or by using throws keyword otherwise the program will give a compilation error.

ClassNotFoundException, IOException, SQLException etc are the examples of the checked exceptions.



I/O Exception: This Program throw I/O exception because of due FileNotFoundException is a checked exception in Java. Anytime, we want to read a file from the file system, Java forces us to handle error situations where the file is not present in the given location.

Implementation: Consider GFG.txt file does not exist.

FileInputStream GFG = new FileInputStream("/Desktop/GFG.txt");

This file does not exist in the location This constructor FileInputStream throws FileNotFoundException which is a checked exception

---------------------

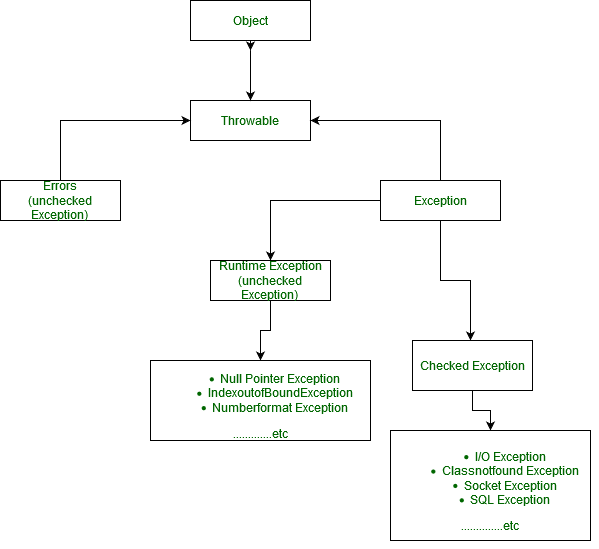
**Java Program to Handle Unchecked Exception**

Exceptions are the issues arising at the runtime resulting in an abrupt flow of working of the program. Remember exceptions are never thrown at the compile-time rather always at runtime be it of any type. No exception is thrown at compile time. Throwable Is super-class of all exceptions and errors too. Now there is an urgency to deal with them for which a concept is defined in Java language known as ‘Exception Handling Techniques’

There are two types of exceptions defined as follows

* Checked Exceptions
* Unchecked Exceptions
* Real-world Illustration: Exceptions

Approach: Now, in order to deal with exceptions, the concept proposed out are exception handling techniques. Straight away diving onto the concept for unchecked exceptions.



**Unchecked Exception**

These types of Exceptions occur during the runtime of the program. These are the exceptions that are not checked at a compiled time by the compiler. In Java exceptions under Error and Runtime Exception classes are unchecked exceptions, This Exception occurs due to bad programming.

Errors class Exceptions like StackOverflow, OutOfMemoryError exception, etc are difficult to handle

Runtime Exceptions like IndexoutOfBoundException, Nullpointer Exception, etc can be handled with the help of try-Catch Block

There are 2 major Unchecked Exceptions which are faced generally by programmers namely IndexOutOfBoundsExcepetion and NullPointerException. They are discussed below with the help of an example also, we will implement them and discuss how to handle them out. Both the major approaches are proposed as below:

IndexOutOfBoundsException

NullPointerException

Case 1: (Array)IndexoutOfBoundException: This Exception occurs due to accessing the index greater than and equal to the size of the array length. The program will automatically be terminated after this exception. In simpler terms, a memory is being tried to accessed which the current data structure is not holding by itself. Here this exception is defined over data structure namely ‘Arrays‘.

// Array containing 4 elements

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

System.out.println(a[5]);

Handling ArrayIndexoutOfBoundException: Try-catch Block we can handle this exception try statement allows you to define a block of code to be tested for errors and catch block captures the given exception object and perform required operations. The program will not terminate.

**ArrayIndexOutOfBoundsException**

Case 2: NullPointerException: This exception occurs when trying to access the object reference that has a null value.

-------------------------------------

**Java Program to Handle Divide By Zero and Multiple Exceptions**

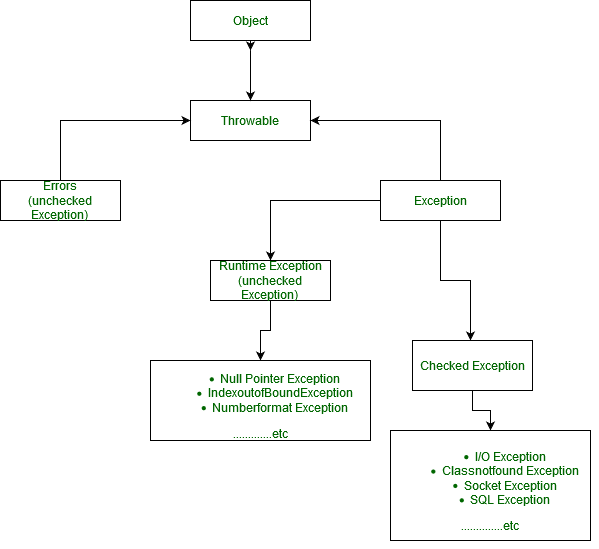
Exceptions These are the events that occur due to the programmer error or machine error which causes a disturbance in the normal flow of execution of the program.

Handling Multiple exceptions: There are two methods to handle multiple exceptions in java.

Using a Single try-catch block try statement allows you to define a block of code to be tested for errors, and we can give exception objects to the catch blow because this all the exceptions inherited by the Exception class.

The second method is to create individual catch blocks for the different exception handler.

**Hierarchy of the exceptions:**



**Divide by zero**: This Program throw Arithmetic exception because of due any number divide by 0 is undefined in Mathematics.

-----------------------------------

**Collections Programs**

Any group of individual objects which are represented as a single unit is known as the collection of the objects. In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

The Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main “root” interfaces of Java collection classes.

What is a Framework?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

Need for a Separate Collection Framework

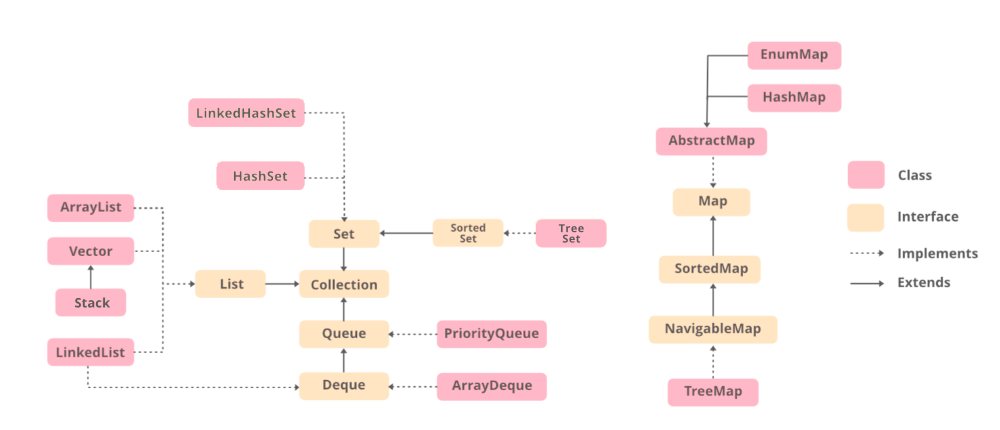
Before the Collection Framework(or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors, or Hashtables. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

Let’s understand this with an example of adding an element in a hashtable and a vector.

Consistent API: The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.

Reduces programming effort: A programmer doesn’t have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.

Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.



Before understanding the different components in the above framework, let’s first understand a class and an interface.

Class: A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

Interface: Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body). Interfaces specify what a class must do and not how. It is the blueprint of the class.

Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface.

|  |
| --- |
| add(Object) This method is used to add an object to the collection. |
| addAll(Collection c) This method adds all the elements in the given collection to this collection. |
| clear() This method removes all of the elements from this collection. |
| contains(Object o) This method returns true if the collection contains the specified element. |
| containsAll(Collection c) This method returns true if the collection contains all of the elements in the given collection. |
| equals(Object o) This method compares the specified object with this collection for equality. |
| hashCode() This method is used to return the hash code value for this collection. |
| isEmpty() This method returns true if this collection contains no elements. |
| iterator() This method returns an iterator over the elements in this collection. |
| max() |
| This method is used to return the maximum value present in the collection. |
| parallelStream() This method returns a parallel Stream with this collection as its source. |
| remove(Object o) This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object. |
| removeAll(Collection c) This method is used to remove all the objects mentioned in the given collection from the collection. |
| removeIf(Predicate filter) This method is used to remove all the elements of this collection that satisfy the given predicate. |
| retainAll(Collection c) This method is used to retain only the elements in this collection that are contained in the specified collection. |
| size() This method is used to return the number of elements in the collection. |
| spliterator() This method is used to create a Spliterator over the elements in this collection. |
| stream() This method is used to return a sequential Stream with this collection as its source. |
| toArray() This method is used to return an array containing all of the elements in this collection. |
|  |

**1. Iterable Interface:** This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the

**Iterator iterator();**

**2. Collection Interface:** This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.

**3. List Interface:** This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

List <T> al = new ArrayList<> ();

List <T> ll = new LinkedList<> ();

List <T> v = new Vector<> ();

Where T is the type of the object

A. **ArrayList**: ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list. ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.

B. **LinkedList:** LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

C. **Vector**: A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

D. **Stack**: Stack class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be referred to as the subclass of Vector.

4. **Queue Interface**: As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold on a first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example,

Queue <T> pq = new PriorityQueue<> ();

Queue <T> ad = new ArrayDeque<> ();

**Priority Queue**: A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

5. **Deque Interface**: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class.

6. **Set Interface**: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes.

Set<T> hs = new HashSet<> ();

Set<T> lhs = new LinkedHashSet<> ();

Set<T> ts = new TreeSet<> ();

A. **HashSet**: The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements.

HashSet<String> hs = new HashSet<String>();

hs.add("Geeks"); hs.add("For"); hs.add("Geeks"); hs.add("Is"); hs.add("Very helpful");

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

while (itr.hasNext()) { System.out.println(itr.next()); }

output:-Geeks For Is Very helpful

B. L**inkedHashSet**: A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

7. **Sorted Set Interface**: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class.

8. **TreeSet**: The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used

TreeSet<String> ts = new TreeSet<String>();

ts.add("Geeks"); ts.add("For"); ts.add("Geeks"); ts.add("Is"); ts.add("Very helpful");

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

while (itr.hasNext()) { System.out.println(itr.next()); }

output:-

For Geeks Is Very helpful

8. **Map Interface**: A map is a data structure that supports the key-value pair mapping for the data. This interface doesn’t support duplicate keys because the same key cannot have multiple mappings. A map is useful if there is data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like HashMap, TreeMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. For example,

Map<T> hm = new HashMap<> ();

Map<T> tm = new TreeMap<> ();

**HashMap**: HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally

HashMap<Integer, String> hm = new HashMap<Integer, String>();

hm.put(1, "Geeks"); hm.put(2, "For"); hm.put(3, "Geeks");

System.out.println("Value for 1 is " + hm.get(1));

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

System.out.println(e.getKey() + " " + e.getValue()); }

output:-

Value for 1 is Geeks 1 Geeks 2 For 3 Geeks

|  |
| --- |
| List Interface |
| Abstract List Class |
| Abstract Sequential List Class |
| Array List |
| Vector Class |
| Stack Class |
| LinkedList Class |
| Queue Interface |
| Blocking Queue Interface |
| AbstractQueue Class |
| PriorityQueue Class |
| PriorityBlockingQueue Class |
| ConcurrentLinkedQueue Class |
| ArrayBlockingQueue Class |
| DelayQueue Class |
| LinkedBlockingQueue Class |
| LinkedTransferQueue |
| Deque Interface |
| BlockingDeque Interface |
| ConcurrentLinkedDeque Class |
| ArrayDeque Class |
| Set Interface |
| Abstract Set Class |
| CopyOnWriteArraySet Class |
| EnumSet Class |
| ConcurrentHashMap Class |
| HashSet Class |
| LinkedHashSet Class |
| SortedSet Interface |
| NavigableSet Interface |
| TreeSet |
| ConcurrentSkipListSet Class |
| Map Interface |
| SortedMap Interface |
| NavigableMap Interface |
| ConcurrentMap Interface |
| TreeMap Class |
| AbstractMap Class |
| ConcurrentHashMap Class |
| EnumMap Class |
| HashMap Class |
| IdentityHashMap Class |
| LinkedHashMap Class |
| HashTable Class |
| Properties Class |
| Other Important Concepts |
| How to convert HashMap to ArrayList |
| Randomly select items from a List |
| How to add all items from a collection to an ArrayList |
| Conversion of Java Maps to List |
| Array to ArrayList Conversion |
| ArrayList to Array Conversion |
| Differences between Array and ArrayList |

------------------

**3 Different ways to print Fibonacci series in Java**

**Method 1** – Iterative: Initialize the first and second numbers to 0 and 1. Following this, we print the first and second numbers. Then we send the flow to the iterative while loop where we get the next number by adding the previous two number and simultaneously we swap the first number with the second and the second with the third.

int num1 = 0, num2 = 1, n=10;

int counter = 0;

while (counter < N) {

System.out.print(num1 + " ");

int num3 = num2 + num1;

num1 = num2;

num2 = num3;

counter = counter + 1; }

Method 2 – Using Recursion: Since Fibonacci Number is the summation of the two previous numbers. We can use recursion as per the following condition:

**Get the number whose Fibonacci series needs to be calculated.**

Recursively iterate from value N to 1:

Base case: If the value called recursively is less than 1, the return 1 the function.

Recursive call: If the base case is not met, then recursively call for previous two value as:

recursive\_function(N – 1) + recursive\_function(N – 2);

static int fib(int n)

{ if (n <= 1)

return n;

// Recursive call

return fib(n - 1)+ fib(n - 2);

}

for (int i = 0; i < N; i++) { System.out.print(fib(i) + " "); }

Method 3 – Using Dynamic Programming: We can avoid the repeated work done in method 2 by storing the Fibonacci numbers calculated so far.

-----------------------------------

**Program to Convert a Vector to List in Java**

Vector<String> vec = new Vector<String>();

vec.add("1"); vec.add("2"); vec.add("3"); vec.add("4"); vec.add("5");

// print Vector elements

System.out.println("Vector: " + vec);

List<String> list = Collections.list(vec.elements());

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

**Output**:-Vector: [1, 2, 3, 4, 5]

List:[1, 2, 3, 4, 5]

**Convert a String to a List of Characters in Java**

**Using Java 8 Stream:**

* Get the String.
* Create a List of Characters.
* Convert to String to IntStream using chars() method.
* Convert IntStream to Stream using mapToObj() method.
* Collect the elements as a List Of Characters using collect()
* Return the List.

public static List<Character>

convertStringToCharList(String str)

{

List<Character> chars = str

.chars()

.mapToObj(e -> (char)e)

.collect(Collectors.toList());

return chars;

}

String str = "Geek";

List<Character> chars = convertStringToCharList(str);

System.out.println(chars);

**output**:- [G, e, e, k]

Using Java 8 Stream:

* Get the String.
* Use the AbstractList interface to convert the String into List of Character
* Return the List.

----------------------------------

**Convert an Iterator to a List in Java**

* Get the Iterator.
* Create an empty list.
* Add each element of the iterator to the list using forEachRemaining() method.
* Return the list.

Iterator<Integer> iterator = Arrays.asList(1, 2, 3, 4, 5).iterator();

List<Integer> list = getListFromIterator(iterator);

System.out.println(list);

public static <T> List<T> getListFromIterator(Iterator<T> iterator)

{

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

iterator.forEachRemaining(list::add);

return list;

}

public static <T> List<T> getListFromIterator(Iterator<T> iterator)

{

Iterable<T> iterable = () -> iterator;

List<T> list = StreamSupport

.stream(iterable.spliterator(), false)

.collect(Collectors.toList());

return list;

}

main method

// Get the Iterator

Iterator<Integer> iterator = Arrays.asList(1, 2, 3, 4, 5).iterator();

List<Integer> list = getListFromIterator(iterator);

System.out.println(list);

**output**:- [1, 2, 3, 4, 5]

--------------------------------

**Program to Convert List to Map in Java**

The List is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.

Using Collectors.toMap() method: This method includes creation of a list of the student objects, and uses Collectors.toMap() to convert it into a Map.

Approach:

Get the List to be converted into Map

Convert the List into stream using List.stream() method

Create map with the help of Collectors.toMap() method

Collect the formed Map using stream.collect() method

Return the formed Map

LinkedHashMap<Integer, String>

map = lt.stream()

.collect(Collectors.toMap( Student::getId, Student::getName,(x, y)

-> x + ", " + y,LinkedHashMap::new));

// print map

map.forEach(

(x, y) -> System.out.println(x + "=" + y));

}

output:

1=Geeks

2=For

3=Geeks

----------------

**Java Program to Print an Integer (Entered by the User)**

Scanner reader = new Scanner(System.in);

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

// nextInt() reads the next integer from the keyboard

int number = reader.nextInt();

// println() prints the following line to the output screen

System.out.println("You entered: " + number);

**Output**:-

Enter a number: 10

You entered: 10

**Program to Add Two Integers**

int first = 10;

int second = 20;

int sum = first + second;

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

output:- The sum is: 30

**Swap two numbers using temporary variable**

float first = 1.20f, second = 2.45f;

System.out.println("First number = " + first);

System.out.println("Second number = " + second);

// Value of first is assigned to temporary

float temporary = first;

// Value of second is assigned to first

first = second;

// Value of temporary (which contains the initial value of first) is assigned to second

second = temporary;

System.out.println("First number = " + first);

System.out.println("Second number = " + second);

**Output**:-

First number = 1.2

Second number = 2.45

First number = 2.45

Second number = 1.2

**Swap two numbers without using temporary variable**

first = first - second;

second = first + second;

first = second - first;

**Check whether a number is even or odd using if...else statement**

if(num % 2 == 0)

System.out.println(num + " is even");

else

System.out.println(num + " is odd");

**Check whether a number is even or odd using ternary operator**

Scanner reader = new Scanner(System.in);

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

int num = reader.nextInt();

String evenOdd = (num % 2 == 0) ? "even" : "odd";

System.out.println(num + " is " + evenOdd);

**Check whether an alphabet is vowel or consonant using if..else statement**

if(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' )

System.out.println(ch + " is vowel");

else

System.out.println(ch + " is consonant");

Check whether an alphabet is vowel or consonant using switch statement

char ch='z';

switch (ch) {

case 'a': case 'e': case 'i': case 'o': case 'u':

System.out.println(ch + " is vowel");

break;

default:

System.out.println(ch + " is consonant");

}

-----------------------------

**Java Program to Find the Largest Among Three Numbers**

double n1 = -4.5, n2 = 3.9, n3 = 2.5;

if( n1 >= n2 && n1 >= n3)

System.out.println(n1 + " is the largest number.");

else if (n2 >= n1 && n2 >= n3)

System.out.println(n2 + " is the largest number.");

else

System.out.println(n3 + " is the largest number.");

**output**:- 3.9 is the largest number

**method 2:-**

double n1 = -4.5, n2 = 3.9, n3 = 5.5;

if(n1 >= n2) {

if(n1 >= n3)

System.out.println(n1 + " is the largest number.");

else

System.out.println(n3 + " is the largest number.");

} else {

if(n2 >= n3)

System.out.println(n2 + " is the largest number.");

else

System.out.println(n3 + " is the largest number.");

}

**output**:-

5.5 is the largest number.

---------

**Java Program to Check Leap Year**

// year to be checked

int year = 1900;

boolean leap = false;

// if the year is divided by 4

if (year % 4 == 0) {

// if the year is century

if (year % 100 == 0) {

// if year is divided by 400 then it is a leap year

if (year % 400 == 0)

leap = true;

else

leap = false;

}//100

// if the year is not century

else

leap = true;

}//4

else

leap = false;

if (leap)

System.out.println(year + " is a leap year.");

else

System.out.println(year + " is not a leap year.");

**output**:- 1900 is not a leap year.

---------------

**Java Program to Check Whether a Character is Alphabet or Not**

char c = 'a';

if (Character.isAlphabetic(c)) {

System.out.println(c + " is an alphabet.");

}

else {

System.out.println(c + " is not an alphabet.");

}

char c = 'A';

String output = (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')

? c + " is an alphabet."

: c + " is not an alphabet.";

System.out.println(output);

**output**:- A is an alphabet.

**Check if a Number is Positive or Negative using if else**

double number = 12.3;

// true if number is less than 0

if (number < 0.0)

System.out.println(number + " is a negative number.");

// true if number is greater than 0

else if ( number > 0.0)

System.out.println(number + " is a positive number.");

// if both test expression is evaluated to false

else

System.out.println(number + " is 0.");

}

**output**:-

12.3 is a positive number.

-12.3 is a negative number.

**Java Program to Calculate the Sum of Natural Numbers**

int num = 100, sum = 0;

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

{

// sum = sum + i;

sum += i;

}

System.out.println("Sum = " + sum);

**output**:- Sum = 5050

int num = 50, i = 1, sum = 0;

while(i <= num)

{ sum += i; i++; }

System.out.println("Sum = " + sum);

**Find Factorial of a number using for loop**

public class Factorial {

public static void main(String[] args) {

int num = 10;

long factorial = 1;

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

{

// factorial = factorial \* i;

factorial \*= i;

}

System.out.printf("Factorial of %d = %d", num, factorial);

}

**output**:- Factorial of 10 = 3628800

**Find Factorial of a number using BigInteger**

int num = 30;

BigInteger factorial = BigInteger.ONE;

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

{

// factorial = factorial \* i;

factorial = factorial.multiply(BigInteger.valueOf(i));

}

System.out.printf("Factorial of %d = %d", num, factorial);

**output**:- Factorial of 30 = 265252859812191058636308480000000

**Find Factorial of a number using while loop**

public class Factorial {

public static void main(String[] args) {

int num = 5, i = 1;

long factorial = 1;

while(i <= num)

{

factorial \*= i;

i++;

}

System.out.printf("Factorial of %d = %d", num, factorial); } }

**output** Factorial of 5 = 120

**Java Program to Generate Multiplication Table**

In this program, you'll learn to generate multiplication table of a given number. This is done by using a for and a while loop in Java.

To understand this example, you should have the knowledge of the following Java programming topics:

**Java for Loop**

Java while and do...while Loop

Example 1: Generate Multiplication Table using for loop

public class MultiplicationTable {

public static void main(String[] args) {

int num = 5;

for(int i = 1; i <= 10; ++i)

{ System.out.printf("%d \* %d = %d \n", num, i, num \* i); } }

**Generate Multiplication Table using while loop**

public class MultiplicationTable {

public static void main(String[] args) {

int num = 9, i = 1;

while(i <= 10)

{ System.out.printf("%d \* %d = %d \n", num, i, num \* i);

i++; }

**Java Program to Display Fibonacci Series**

method-1

int n = 10, firstTerm = 0, secondTerm = 1;

System.out.println("Fibonacci Series till " + n + " terms:");

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

System.out.print(firstTerm + ", ");

// compute the next term

int nextTerm = firstTerm + secondTerm;

firstTerm = secondTerm;

secondTerm = nextTerm;

}

**method-2**

int i = 1, n = 10, firstTerm = 0, secondTerm = 1;

System.out.println("Fibonacci Series till " + n + " terms:");

while (i <= n) {

System.out.print(firstTerm + ", ");

int nextTerm = firstTerm + secondTerm;

firstTerm = secondTerm;

secondTerm = nextTerm;

i++;

}

**method-3**

int n = 100, firstTerm = 0, secondTerm = 1;

System.out.println("Fibonacci Series Upto " + n + ": ");

while (firstTerm <= n) {

System.out.print(firstTerm + ", ");

int nextTerm = firstTerm + secondTerm;

firstTerm = secondTerm;

secondTerm = nextTerm; }

**Java Program to Find GCD of two Numbers**

The HCF or GCD of two integers is the largest integer that can exactly divide both numbers (without a remainder).

// find GCD between n1 and n2

int n1 = 81, n2 = 153;

// initially set to gcd

int gcd = 1;

for (int i = 1; i <= n1 && i <= n2; ++i) {

// check if i perfectly divides both n1 and n2

if (n1 % i == 0 && n2 % i == 0)

gcd = i; }

System.out.println("GCD of " + n1 +" and " + n2 + " is " + gcd);

**Output** GCD of 81 and 153 is 9

int n1 = 72, n2 = 120, lcm;

// maximum number between n1 and n2 is stored in lcm

lcm = (n1 > n2) ? n1 : n2;

// Always true

while(true) {

if( lcm % n1 == 0 && lcm % n2 == 0 ) {

System.out.printf("The LCM of %d and %d is %d.", n1, n2, lcm);

break; }

++lcm; }

**Output**:- The LCM of 72 and 120 is 360.

**Java Program to Display Alphabets (A to Z) using loop**

**Display uppercased alphabet using for loop**

char c;

for(c = 'A'; c <= 'Z'; ++c)

System.out.print(c + " ");

**output**:- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

**Display lowercase alphabet using for loop**

char c;

for(c = 'a'; c <= 'z'; ++c)

System.out.print(c + " ");

**output**:- a b c d e f g h i j k l m n o p q r s t u v w x y z

**Java Program to Count Number of Digits in an Integer**

int count = 0, num = 0003452;

while (num != 0) {

// num = num/10

num /= 10;

++count; }

System.out.println("Number of digits: " + count);

**output**:- Number of digits: 4

**Java Program to Reverse a Number**

int num = 1234, reversed = 0;

System.out.println("Original Number: " + num);

// run loop until num becomes 0

while(num != 0) {

// get last digit from num

int digit = num % 10;

reversed = reversed \* 10 + digit;

// remove the last digit from num

num /= 10;

}

System.out.println("Reversed Number: " + reversed);

**output**:- Reversed Number: 4321

**Java Program to Check Palindrome**

String str = "Radar", reverseStr = "";

int strLength = str.length();

for (int i = (strLength - 1); i >=0; --i) {

reverseStr = reverseStr + str.charAt(i); }

if (str.toLowerCase().equals(reverseStr.toLowerCase())) {

System.out.println(str + " is a Palindrome String."); }

else { System.out.println(str + " is not a Palindrome String."); }

**output**:- Radar is a Palindrome String.

int num = 3553, reversedNum = 0, remainder;

int originalNum = num;

while (num != 0) {

remainder = num % 10;

reversedNum = reversedNum \* 10 + remainder;

num /= 10; }

if (originalNum == reversedNum) {

System.out.println(originalNum + " is Palindrome."); }

else { System.out.println(originalNum + " is not Palindrome."); }

**output**:- 3553 is Palindrome.

**Java Program to Check Whether a Number is Prime or Not**

int num = 29;

boolean flag = false;

for (int i = 2; i <= num / 2; ++i) {

// condition for nonprime number

if (num % i == 0) {

flag = true;

break; } }

if (!flag)

System.out.println(num + " is a prime number.");

else{ System.out.println(num + " is not a prime number."); }

**method-2**

int num = 33, i = 2;

boolean flag = false;

while (i <= num / 2) {

// condition for nonprime number

if (num % i == 0) {

flag = true; break; }

++i; }

if (!flag)

System.out.println(num + " is a prime number.");

else{ System.out.println(num + " is not a prime number."); }

**output**:- 33 is not a prime number.

**Java Program to Display Prime Numbers Between Two Intervals**

int low = 20, high = 50;

while (low < high) {

boolean flag = false;

for(int i = 2; i <= low/2; ++i) {

// condition for nonprime number

if(low % i == 0) {

flag = true;

break; } }

if (!flag && low != 0 && low != 1)

System.out.print(low + " ");

++low; }

**output**:- 23 29 31 37 41 43 47

**Java Program to Calculate Average Using Arrays**

double[] numArray = { 45.3, 67.5, -45.6, 20.34, 33.0, 45.6 };

double sum = 0.0;

for (double num: numArray) { sum += num; }

double average = sum / numArray.length;

System.out.format("The average is: %.2f", average);

output:- The average is: 27.69

**Java Program to Find Largest Element of an Array**

double[] numArray = { 23.4, -34.5, 50.0, 33.5, 55.5, 43.7, 5.7, -66.5 };

double largest = numArray[0];

for (double num: numArray) {

if(largest < num) largest = num; }

System.out.format("Largest element = %.2f", largest);

**output**:- Largest element = 55.50

**Java Program to Print an Array**

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

for (int element: array) { System.out.println(element);}

System.out.println(Arrays.toString(array));

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

System.out.println(Arrays.deepToString(array));

**output**:- [[1, 2], [3, 4], [5, 6, 7]]

**Java Program to Convert String to Date**

// Format y-M-d or yyyy-MM-d

String string = "2017-07-25";

LocalDate date = LocalDate.parse(string, DateTimeFormatter.ISO\_DATE);

System.out.println(date);

**output**:- 2017-07-25

String string = "July 25, 2017";

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMMM d, yyyy", Locale.ENGLISH);

LocalDate date = LocalDate.parse(string, formatter);

System.out.println(date);

**output**:- 2017-07-25

**Java Program to Compare Strings**

String style = "Bold";

String style2 = "Bold";

if(style == style2)

System.out.println("Equal");

else

System.out.println("Not Equal");

**output**:- Equal

String style = new String("Bold");

String style2 = new String("Bold");

if(style.equals(style2))

System.out.println("Equal");

else

System.out.println("Not Equal");

**output**:- Equal

String style = new String("Bold");

String style2 = new String("Bold");

if(style == style2)

System.out.println("Equal");

else

System.out.println("Not Equal");

**output**:- Not Equal

**Java Program to Clear the StringBuffer**

StringBuffer str = new StringBuffer();

str.append("Java");

str.append(" is");

str.append(" popular.");

System.out.println("StringBuffer: " + str);

str.delete(0, str.length());

System.out.println("Updated StringBuffer: " + str);

**output**:-

StringBuffer: Java is popular.

Updated StringBuffer:

**Java Program to Implement switch statement on strings**

String language = "Java";

switch(language) {

case "Java":

System.out.println(language + " is famous for enterprise applications.");

break;

case "JavaScript":

System.out.println(language + " is famous for frontend and backend.");

break;

case "Python":

System.out.println(language + " is famous for ML and AI.");

break;

default:

System.out.println(language + " not found on record.");

break;

}

**output**

Java is famous for enterprise applications.

**Java Program to Check if a string contains a substring**

String txt = "This is Programiz";

String str1 = "Programiz";

String str2 = "Programming";

boolean result = txt.contains(str1);

if(result) {System.out.println(str1 + " is present in the string."); }

else { System.out.println(str1 + " is not present in the string."); }

result = txt.contains(str2);

if(result) { System.out.println(str2 + " is present in the string."); }

else { System.out.println(str2 + " is not present in the string."); }

**output**:-

Programiz is present in the string.

Programming is not present in the string.