Contents

[JAVA Introduction 3](#_Toc45318984)

[Types of JAVA Applications 4](#_Toc45318985)

[Features of JAVA 4](#_Toc45318986)

[JAVA Installation 5](#_Toc45318987)

[Java Editors 5](#_Toc45318988)

[Basic Structure of JAVA Programs 5](#_Toc45318989)

[Java JDK, JRE and JVM 7](#_Toc45318990)

[JAVA DATA Types 8](#_Toc45318991)

[Java Variables 8](#_Toc45318992)

[Java Primitive Data Types 9](#_Toc45318993)

[Java literals 10](#_Toc45318994)

[Java Operators 11](#_Toc45318995)

[Java Flow Control 15](#_Toc45318996)

[if..else..if Statement 15](#_Toc45318997)

[switch Statement 15](#_Toc45318998)

[for loop 16](#_Toc45318999)

[for-each Loop 16](#_Toc45319000)

[while Loop 17](#_Toc45319001)

[do...while Loop 17](#_Toc45319002)

[Break 17](#_Toc45319003)

[Continue 17](#_Toc45319004)

[JAVA Arrays 18](#_Toc45319005)

[Single dimensional Arrays 18](#_Toc45319006)

[Multidimensional Arrays 18](#_Toc45319007)

[Copying Arrays 19](#_Toc45319008)

[Using Assignment operator 19](#_Toc45319009)

[Using Looping Construct 19](#_Toc45319010)

[Using arraycopy() method 20](#_Toc45319011)

[Using copyOfRange() 20](#_Toc45319012)

[Java Class 20](#_Toc45319013)

[Java Methods 20](#_Toc45319014)

[Method declration 21](#_Toc45319015)

[Call a Method 21](#_Toc45319016)

[Advantages of using methods 22](#_Toc45319017)

[Java Recursion 22](#_Toc45319018)

[finalize method 22](#_Toc45319019)

[Java Constructors 22](#_Toc45319020)

[Types of Constructor 23](#_Toc45319021)

[Constructors Overloading 24](#_Toc45319022)

[Java String 24](#_Toc45319023)

[Java Strings are Immutable 27](#_Toc45319024)

[Convert String to Integer 27](#_Toc45319025)

[JAVA KEYWORDS 28](#_Toc45319026)

[this Keyword 28](#_Toc45319027)

[final keyword 30](#_Toc45319028)

[instanceof keyword 31](#_Toc45319029)

[super keyword 33](#_Toc45319030)

[OOPs(Object-Oriented Programming System) 34](#_Toc45319031)

[Inheritance 34](#_Toc45319032)

[Types of inheritance 35](#_Toc45319033)

[Polymorphism 35](#_Toc45319034)

[Method overriding 36](#_Toc45319035)

[Method Overloading 36](#_Toc45319036)

[Abstraction 37](#_Toc45319037)

[Abstract Class and Abstract Methods 37](#_Toc45319038)

[Interface 37](#_Toc45319039)

[Encapsulation 38](#_Toc45319040)

[Access Modifiers 39](#_Toc45319041)

[Class 40](#_Toc45319042)

[Nested and Inner Class 40](#_Toc45319043)

[Non-Static Nested Class (Inner Class) 40](#_Toc45319044)

[Static Nested Class 42](#_Toc45319045)

[Anonymous Class 42](#_Toc45319046)

[Anonymous Class Extending a Class 42](#_Toc45319047)

[Singleton Class 44](#_Toc45319048)

[enum Class 45](#_Toc45319049)

[Java Exceptions 46](#_Toc45319050)

[Errors 47](#_Toc45319051)

[Exceptions 47](#_Toc45319052)

[1. RuntimeException 47](#_Toc45319053)

[Java Exception examples 48](#_Toc45319054)

[2. IOException 48](#_Toc45319055)

[Java Exception Handling 49](#_Toc45319056)

[Handle checked exceptions using throw and throws. 49](#_Toc45319057)

[Java throws keyword 49](#_Toc45319058)

[Java throw keyword 50](#_Toc45319059)

[JAVA Collections 50](#_Toc45319060)

[Java Collection Interface 51](#_Toc45319061)

[Java List Interface 52](#_Toc45319062)

[ArrayList Class 52](#_Toc45319063)

[Java Vector Class 56](#_Toc45319064)

[Stack Class 57](#_Toc45319065)

[Queue Interface 58](#_Toc45319066)

[PriorityQueue Class 59](#_Toc45319067)

[Deque Interface 60](#_Toc45319068)

[ArrayDeque Class 61](#_Toc45319069)

[LinkedList Class 63](#_Toc45319070)

[Set Interface 64](#_Toc45319071)

[HashSet Class 65](#_Toc45319072)

[LinkedHashSet Class 66](#_Toc45319073)

[SortedSet Interface 66](#_Toc45319074)

[TreeSet Class 67](#_Toc45319075)

[Map Interface 67](#_Toc45319076)

[HashMap Class 69](#_Toc45319077)

# JAVA Introduction

Java is an [object-oriented](https://www.w3schools.in/java-tutorial/object-oriented-programming-oops/) programming language developed by Sun Microsystems, and it was released in 1995

Java is one of the most important programming languages in today's IT industries.

**JSP** - In Java, JSP (Java Server Pages) is used to create dynamic web pages, such as in PHP and ASP.

**Applets** - Applets are another type of Java programs that are implemented on Internet browsers and are always run as part of a web document.

**J2EE** - Java 2 Enterprise Edition is a platform-independent environment that is a set of different protocols and APIs and is used by various organizations to transfer data between each other.

**JavaBeans** - This is a set of reusable software components that can be easily used to create new and advanced applications.

**Mobile** - In addition to the above technology, Java is widely used in mobile devices nowadays, many types of games and applications are being made in Java.

## Types of JAVA Applications

**Web** **Application** - Java is used to create server-side web applications. Currently, Servlet, JSP, Struts, JSF, etc. technologies are used.

**Standalone Application** - It is also known as the desktop application or window-based application. An application that we need to install on every machine or server such as media player, antivirus, etc. AWT and Swing are used in java for creating standalone applications.

**Enterprise Application** - An application that is distributed in nature, such as banking applications, etc. It has the advantage of high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

**Mobile Application** - Java is used to create application software for mobile devices. Currently, Java ME is used for building applications for small devices, and also Java is a programming language for Google Android application development.

## Features of JAVA

**Object-Oriented** - Java supports the features of object-oriented programming. Its object model is simple and easy to expand.

**Platform independent** - C and C++ are platform dependency languages hence the application programs written in one Operating system cannot run in any other Operating system, but in platform independence language like Java application programs written in one Operating system can able to run on any Operating system.

**Simple** - Java has included many features of C / C ++, which makes it easy to understand.

**Secure** - Java provides a wide range of protection from viruses and malicious programs.  It ensures that there will be no damage and no security will be broken.

**Portable** - Java provides us with the concept of portability. Running the same program with Java on different platforms is possible.

**Robust** - During the development of the program, it helps us to find possible mistakes as soon as possible.

**Multi-threaded** - The multithreading programming feature in Java allows you to write a program that performs several different tasks simultaneously.

**Distributed** - Java is designed for distributed Internet environments as it manages the TCP/IP protocol.

## JAVA Installation

Step 0: Un-Install Older Version(s) of JDK/JRE

Step 1: Download JDK

Step 2: Install JDK

Step 3: Include JDK's "bin" Directory in the PATH

Step 4: Verify the JDK Installation

|  |
| --- |
| // Display the JDK version  javac –version |

## Java Editors

we use a variety of different IDEs (Integrated Development Environments) to develop Java code. These IDEs offer a variety of features: building Java applications, TestNG, debugging, code inspections, code assistance, JUNIT testing, multiple refactoring, visual GUI builder and code editor, Java, Maven build tools, ant, do data modelling and build queries, and more.

* NetBeans
* Eclipse
* IntelliJ IDEA Community Edition
* Android Studio
* Enide Studio 2014
* BlueJ
* jEdit
* jGRASP
* JSource
* JDeveloper
* DrJava

## Basic Structure of JAVA Programs

A Java program involves the following sections:

* Documentation Section
* Package Statement
* Import Statements
* Interface Statement
* Class Definition
* Main Method Class
* Main Method Definition

|  |  |
| --- | --- |
| Section | Description |
| Documentation Section | You can write a comment in this section. Comments are beneficial for the programmer because they help them understand the code. These are optional, but we suggest you use them because they are useful to understand the operation of the program, so you must write comments within the program.  // single line comment  /\* multi line  Comments \*/ |
| Package statement | You can create a package with any name. A package is a group of classes that are defined by a name. That is, if you want to declare many classes within one element, then you can declare it within a package. It is an optional part of the program, i.e., if you do not want to declare any package, then there will be no problem with it, and you will not get any errors. Here, the package is a keyword that tells the compiler that package has been created.  It is declared as:  package package\_name; |
| Import statements | This line indicates that if you want to use a class of another package, then you can do this by importing it directly into your program.  Example:  import calc.add; |
| Interface statement | Interfaces are like a class that includes a group of method declarations. It's an optional section and can be used when programmers want to implement multiple inheritances within a program. |
| Class Definition | A Java program may contain several class definitions. Classes are the main and essential elements of any Java program.  Every valid Java Application must have a class definition (that matches the filename). |
| Main Method Class | Every Java stand-alone program requires the main method as the starting point of the program. This is an essential part of a Java program. There may be many classes in a Java program, and only one class defines the main method. Methods contain data type declaration and executable statements. The main method must be inside the class definition. The compiler executes the codes starting from the main function. |

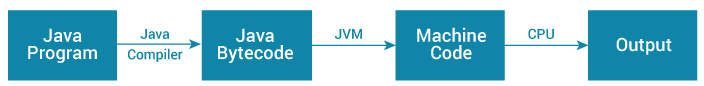
## Java JDK, JRE and JVM

**JVM:**

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since, JVM ​executes the Java bytecode which is platform independent, Java is platform-independent.



**JRE:**

JRE (Java Runtime Environment) is a software package that provides Java class libraries, along with Java Virtual Machine (JVM), and other components to run applications written in Java programming. JRE is the superset of JVM.



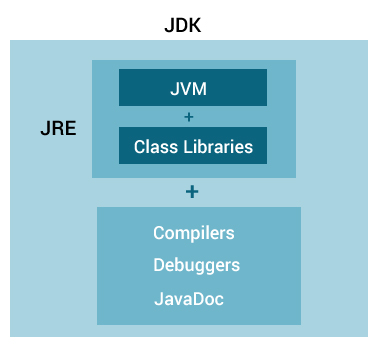
If you need to run Java programs, but not develop them, JRE is what you need.

**JDK:**

JDK (Java Development Kit) is a software development kit to develop applications in Java. When you download JDK, JRE is also downloaded, and don't need to download it separately. In addition to JRE, JDK also contains number of development tools (compilers, JavaDoc, Java Debugger etc).



Here's the relationship between JVM, JRE, and JDK.



## JAVA DATA Types

### Java Variables

A variable is a location in memory (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

**Rules for Naming an Identifier**

* Identifier cannot be a keyword.
* Identifiers are case-sensitive.
* It can have a sequence of letters and digits. However, it must begin with a letter, $ or \_. The first letter of an identifier cannot be a digit.
* It's convention to start an identifier with a letter rather and $ or \_.
* Whitespaces are not allowed.
* Similarly, you cannot use symbols such as @, #, and so on.

|  |
| --- |
| int speedLimit = 80; |

If you choose one word variable name, use all lowercase letters.

If you choose variable name having more than one word, use all lowercase letters for the first word and capitalize the first letter of each subsequent word. For example, speedLimit.

There are 4 types of variables in Java programming language:

* Instance Variables (Non-Static Fields)
* Class Variables (Static Fields)
* Local Variables
* Parameters

### Java Primitive Data Types

All variables must be declared before they can be used.

There are 8 data types predefined in Java programming language, known as primitive data types.

1. **boolean**

The boolean data type has two possible values, either true or false.

Default value: false.

They are usually used for true/false conditions.

1. **byte**

The byte data type can have values from -128 to 127 (8-bit signed two's complement integer).

It's used instead of int or other integer data types to save memory if it's certain that the value of a variable will be within [-128, 127].

Default value: 0

1. **short**

The short data type can have values from -32768 to 32767 (16-bit signed two's complement integer).

It's used instead of other integer data types to save memory if it's certain that the value of the variable will be within [-32768, 32767].

Default value: 0

1. **int**

The int data type can have values from -231 to 231-1 (32-bit signed two's complement integer).

If you are using Java 8 or later, you can use unsigned 32-bit integer with minimum value of 0 and maximum value of 232-1.

Default value: 0

1. **long**

The long data type can have values from -263 to 263-1 (64-bit signed two's complement integer).

If you are using Java 8 or later, you can use unsigned 64-bit integer with minimum value of 0 and maximum value of 264-1.

Default value: 0

1. **double**

The double data type is a double-precision 64-bit floating point.

It should never be used for precise values such as currency.

Default value: 0.0 (0.0d)

1. **float**

The float data type is a single-precision 32-bit floating point.

It should never be used for precise values such as currency.

Default value: 0.0 (0.0f)

1. **char**

It's a 16-bit Unicode character.

The minimum value of char data type is '\u0000' (0). The maximum value of char data type is '\uffff'.

Default value: '\u0000'

### Java literals

To understand literals, let's take an example to assign value to a variable.

boolean flag = false;

Here,

boolean - is data type.

flag - is variable

false - is literal.

A Literal is the source code representation of a fixed value.

**Integer Literals**

Integer literals are used to initialize variables of integer data types byte, short, int and long.

If an integer literal ends with l or L, it's of type long. Tip: it is better to use L instead of l.

Integer literals can be expressed in decimal, hexadecimal and binary number systems.

The numbers starting with prefix 0x represents hexadecimal. Similarly, numbers starting with prefix 0b represents binary.

|  |
| --- |
| // decimal  int decNumber = 34;  // 0x represents hexadecimal  int hexNumber = 0x2F;  // 0b represents binary  int binNumber = 0b10010; |

**Floating-point Literals**

Floating-point literals are used to initialize variables of data type float and double.

If a floating-point literal ends with f or F, it's of type float. Otherwise, it's of type double. A double type can optionally end with D or d. However, it's not necessary.

They can also be expressed in scientific notation using E or e.

|  |
| --- |
| double myDouble = 3.4;  float myFloat = 3.4F; |

**Character and String Literals**

They contain [Unicode (UTF-16) characters](https://en.wikipedia.org/wiki/List_of_Unicode_characters).

For char literals, single quotation is used. For example, 'a', '\u0111' etc.

For String literals, double quotation is used. For example, "programming", "Java 8"

Java also supports a few special escape sequences. For example, \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

|  |
| --- |
| char myChar = 'g';  char newLine = '\n';  String myString = "Java 8"; |

## Java Operators

Operators are special symbols (characters) that carry out operations on operands (variables and values).

**Assignment Operator**

Assignment operators are used in Java to assign values to variables

| Java Assignment Operators | | |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x / 5 |
| <<= | x <<= 5 | x = x << 5 |
| >>= | x >>= 5 | x = x >> 5 |
| &= | x &= 5 | x = x & 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| |= | x |= 5 | x = x | 5 |

**Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

| Java Arithmetic Operators | |
| --- | --- |
| Operator | Meaning |
| + | Addition (also used for string concatenation) |
| - | Subtraction Operator |
| \* | Multiplication Operator |
| / | Division Operator |
| % | Remainder Operator |

**Unary Operators**

Unary operator performs operation on only one operand.

| Operator | Meaning |
| --- | --- |
| + | Unary plus (not necessary to use since numbers are positive without using it) |
| - | Unary minus; inverts the sign of an expression |
| ++ | Increment operator; increments value by 1 |
| -- | decrement operator; decrements value by 1 |
| ! | Logical complement operator; inverts the value of a boolean |

**Equality and Relational Operators**

The equality and relational operators determines the relationship between two operands. It checks if an operand is greater than, less than, equal to, not equal to and so on. Depending on the relationship, it results to either true or false.

| Java Equality and Relational Operators | | |
| --- | --- | --- |
| Operator | Description | Example |
| == | equal to | 5 == 3 is evaluated to false |
| != | not equal to | 5 != 3 is evaluated to true |
| > | greater than | 5 > 3 is evaluated to true |
| < | less than | 5 < 3 is evaluated to false |
| >= | greater than or equal to | 5 >= 5 is evaluated to true |
| <= | less then or equal to | 5 <= 5 is evaluated to true |

Equality and relational operators are used in decision making and loops

**instanceof Operator**

|  |
| --- |
| class instanceofOperator {  public static void main(String[] args) {    String test = "asdf";  boolean result;    result = test instanceof String;  System.out.println(result);  }  } |

When you run the program, the output will be true. It's because test is the instance of String class.

**Logical Operators**

The logical operators || (conditional-OR) and && (conditional-AND) operates on boolean expressions.

**Ternary Operator**

The conditional operator or ternary operator ?: is shorthand for if-then-else statement. The syntax of conditional operator is:

|  |
| --- |
| variable = Expression ? expression1 : expression2 |

**Bitwise and Bit Shift Operators**

To perform bitwise and bit shift operators in Java, these operators are used.

| Java Bitwise and Bit Shift Operators | |
| --- | --- |
| Operator | Description |
| ~ | Bitwise Complement |
| << | Left Shift |
| >> | Right Shift |
| >>> | Unsigned Right Shift |
| & | Bitwise AND |
| ^ | Bitwise exclusive OR |
| | | Bitwise inclusive OR |

# Java Flow Control

## if..else..if Statement

|  |
| --- |
| if (expression1)  {  // codes  }  else if(expression2)  {  // codes  }  else if (expression3)  {  // codes  }  .  .  else  {  // codes  } |

## switch Statement

|  |
| --- |
| switch (variable/expression) {  case value1:  // statements  break;  case value2:  // statements  break;  .. .. ...  .. .. ...  default:  // statements  } |

## for loop

Loop is used in programming to repeat a specific block of code until certain condition is met (test expression is false).

|  |
| --- |
| for (initialization; testExpression; update)  {  // codes inside for loop's body  } |

## for-each Loop

The loop iterates through each element of array/collection.

|  |
| --- |
| for(data\_type item : collection) {  ...  } |

collection is a collection or array variable which you have to loop through.

item is a single item from the collection.

For each iteration, for-each loop

* iterates through each item in the given collection or array (collection),
* stores each item in a variable (item)
* and executes the body of the loop.

|  |
| --- |
| class EnhancedForLoop {  public static void main(String[] args) {    int[] numbers = {3, 4, 5, -5, 0, 12};  int sum = 0;    for (int number: numbers) {  sum += number;  }    System.out.println("Sum = " + sum);  }  } |

## while Loop

|  |
| --- |
| while (testExpression) {  // codes inside body of while loop  } |

The test expression inside parenthesis is a boolean expression.

If the test expression is evaluated to true,

* statements inside the while loop are executed.
* then, the test expression is evaluated again.

This process goes on until the test expression is evaluated to false.

If the test expression is evaluated to false,

* while loop is terminated.

|  |
| --- |
| class Loop {  public static void main(String[] args) {    int i = 1;    while (i <= 10) {  System.out.println("Line " + i);  ++i;  }  }  } |

## do...while Loop

The do...while loop is similar to while loop with one key difference. The body of do...while loop is executed for once before the test expression is checked.

|  |
| --- |
| do {  // codes inside body of do while loop  } while (testExpression); |

## Break

The break statement terminates the loop immediately, and the control of the program moves to the next statement following the loop.

|  |
| --- |
| break; |

## Continue

The continue statement skips the current iteration of a loop.

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

When continue statement is executed, control of the program jumps to the end of the loop. Then, the test expression that controls the loop is evaluated. In case of for loop, the update statement is executed before the test expression is evaluated.

|  |
| --- |
| continue; |

# JAVA Arrays

An array is a container that holds data (values) of one single type

## Single dimensional Arrays

**Declaration**

|  |
| --- |
| dataType[] arrayName;  eg. int[] studentNumber; |

dataType can be a primitive data type like: int, char, Double, byte etc. or an object

Allocate memory for array elements.

|  |
| --- |
| studentNumber = new int[10]; |

once the length of the array is defined, it cannot be changed in the program.

It's possible to declare and allocate memory of an array in one statement. You can replace two statements above with a single statement

|  |
| --- |
| int[] studentNumber = new int[10]; |

**Initialize**

You can initialize arrays during declaration or you can initialize (or change values) later in the program as per your requirement.

1. initialize an array during declaration.

|  |
| --- |
| int[] age = {12, 4, 5, 2, 5}; |

**Access**

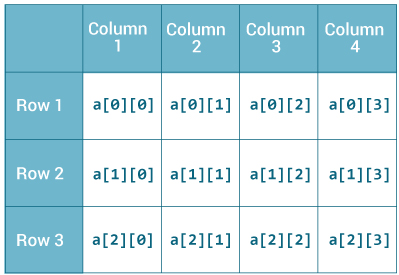
You can access or alter elements of an array by using indices.

The first element of array is age[0], second is age[1] and so on.

## Multidimensional Arrays

declare and initialize

|  |
| --- |
| Double[][] matrix = {{1.2, 4.3, 4.0},  {4.1, -1.1}  };  int[][] a = new int[3][4]; |



Similarly, you can declare a three-dimensional (3d) array. For example,

|  |
| --- |
| String[][][] personalInfo = new String[3][4][2];  int[][][] test = {  {  {1, -2, 3},  {2, 3, 4}  },  {  {-4, -5, 6, 9},  {1},  {2, 3}  }  }; |

## Copying Arrays

### Using Assignment operator

|  |
| --- |
| int [] numbers = {1, 2, 3, 4, 5, 6};  int [] positiveNumbers = numbers; // copying arrays |

Both arrays refers to the same array object. If you change elements of one array in the above example, corresponding elements of the other array is also changed. This is called shallow copy.

### Using Looping Construct

|  |
| --- |
| int [] source = {1, 2, 3, 4, 5, 6};  int [] destination = new int[6];  for (int i = 0; i < source.length; ++i) {  destination[i] = source[i];  } |

The source and destination array doesn't share the same reference (deep copy).

### Using arraycopy() method

|  |
| --- |
| // copying elements from index 2 on n1 array  // copying element to index 1 of n3 array  // 2 elements will be copied  System.arraycopy(n1, 2, n3, 1, 2); |

### Using copyOfRange()

you can use copyOfRange() method defined in java.util.Arrays

|  |
| --- |
| int[] source = {2, 3, 12, 4, 12, -2};  // copying entire source array to destination  int[] destination1 = Arrays.copyOfRange(source, 0, source.length);  // copying from index 2 to 5 (5 is not included)  int[] destination2 = Arrays.copyOfRange(source, 2, 5);  System.out.println("destination2 = " + Arrays.toString(destination2)); |

# Java Class

A class is a blueprint for the object.

|  |
| --- |
| class ClassName {  // variables  // methods  } |

These variables and methods defined within a class are called members of the class.

When class is defined, only the specification for the object is defined; no memory or storage is allocated.

To access members defined within the class, you need to create objects

|  |
| --- |
| ClassName l1 = new ClassName ();  l1.variables = 30;  l1.methods(); |

Variables defined within a class are called instance variable for a reason.

When an object is initialized, it's called an instance. Each instance contains its own copy of these variables.

## Java Methods

Depending on whether a method is defined by the user, or available in standard library, there are two types of methods:

* Standard Library Methods
* User-defined Methods

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (\*.jar) file with JVM and JRE.

User-defined Method

We can also create methods of our own choice to perform some task. Such methods are called user-defined methods.

### Method declration

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**

|  |
| --- |
| public class MyClass {  static void myMethod() {  // code to be executed  }  } |

**modifier** - It defines access types whether the method is public, private and so on.

**static** - If we use the static keyword, it can be accessed without creating objects.

**returnType** - It specifies what type of value a method returns.A method can return native data types (int, float, double, etc), native objects (String, Map, List, etc), or any other built-in and user-defined objects. If the method does not return a value, its return type is void.

**nameOfMethod** - It is an identifier that is used to refer to the particular method in a program.

**parameters** (arguments) - These are values passed to a method. We can pass any number of arguments to a method.

**method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }

### Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

While executing the program code, it encounters myFunction(); in the code.

The execution then branches to the myFunction() method and executes code inside the body of the method.

After the execution of the method body, the program returns to the original state and executes the next statement after the method call.

### Advantages of using methods

1. The main advantage is code reusability. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".
2. Methods make code more readable and easier to debug.

### Java Recursion

A method that calls itself is known as a recursive method. And, this process is known as recursion.

In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.

Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

|  |
| --- |
| class Factorial {  static int factorial( int n ) {  if (n != 0) // termination condition  return n \* factorial(n-1); // recursive call  else  return 1;  }  public static void main(String[] args) {  int number = 4, result;  result = factorial(number);  System.out.println(number + " factorial = " + result);  }  } |

**Advantages and Disadvantages of Recursion**

When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow.

On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.

### ****finalize method****

It is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection, so as to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation. Remember it is not a reserved keyword.

|  |
| --- |
| protected void finalize throws Throwable{} |

Once the finalize method completes immediately Garbage Collector destroy that object. finalize method is present in Object class

## Java Constructors

In Java, every class has its constructor that is invoked automatically when an object of the class is created. A constructor is similar to a method but in actual, it is not a method.

A Java method and Java constructor can be differentiated by its name and return type. A constructor has the same name as that of class and it does not return any value.

|  |
| --- |
| class Test {  Test() {  // constructor body  }  } |

Note: Methods can have the same name as that of the class and if it has a return type (e.g. void). Hence, it's a method, not a constructor.

* Constructors are invoked implicitly when you instantiate objects.
* The two rules for creating a constructor are:T

The name of the constructor should be the same as that of class.

A Java constructor must not have a return type.

* A constructor cannot be abstract or static or final.
* A constructor can be overloaded but can not be overridden.

### Types of Constructor

In Java, constructors can be divided into 3 types:

1. No-Arg Constructor
2. Default Constructor
3. Parameterized Constructor

**No-Arg Constructor**

A Java constructor may or may not have any parameters (arguments). If a constructor does not accept any parameters, it is known as a no-arg constructor

|  |
| --- |
| private Constructor() {  // body of constructor  } |

If the object is to be created outside of the class, you have to declare the constructor public to access it.

**Default Constructor:** If you do not create any constructors, the Java compiler will automatically create a no-argument constructor during run-time. This constructor is known as the default constructor. The default constructor initializes any uninitialized instance variables with default values.

|  |
| --- |
| Type Default Value  boolean false  byte 0  short 0  int 0  long 0L  char \u0000  float 0.0f  double 0.0d  object Reference null |

**Parameterized Constructor**

Similar to methods, we can pass parameters to a constructor. Such constructors are known as a parameterized constructor.

|  |
| --- |
| public Constructor (arg1, arg2, ..., argn) {  // constructor body  } |

### Constructors Overloading

Similar to method overloading, we can also overload constructors in Java. In constructor overloading, there are two or more constructors with different parameters.

There is **no** such thing as **constructor overriding** in Java.

# Java String

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

Unlike other programming languages, strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String.

Java String Methods

Java String provides various methods that allow us to perform different string operations. Here are some of the commonly used string methods.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| concat() | joins the two strings together |
| equals() | compares the value of two strings |
| charAt() | returns the character present in the specified location |
| getBytes() | converts the string to an array of bytes |
| indexOf() | returns the position of the specified character in the string |
| length() | returns the size of the specified string |
| replace() | replaces the specified old character with the specified new character |
| substring() | returns the substring of the string |
| split() | breaks the string into an array of strings |
| toLowerCase() | converts the string to lowercase |
| toUpperCase() | converts the string to uppercase |
| valueOf() | returns the string representation of the specified data |

|  |
| --- |
| //checks the string length  greet.length();    // join two strings  greet.concat(name)    //we can also join two strings using the + operator.  String joinedString = greet + name;    // compare first and second strings  boolean result1 = first.equals(second);  //We can also use the == operator and compareTo() method to make a comparison between 2 strings.  // returns the character at 7  greet.charAt(7);  // returns the substring from index  example.substring(7);  // converts the string to lowercase  example.toLowerCase();  // converts the string to uppercase  example.toUpperCase();  // replaces the character '!' with 'o'  example.replace('!', 'o'); |

We can also use the == operator and compareTo() method to make a comparison between 2 strings.

The escape character \ is used in Java to represent double-quotes in double-quotes

**IndexOf(char b)**

This method returns the index of the character 'b' passed as parameter. If that character is not available in the string, the returned index would be -1.

**IndexOf(char c, int startindex)**

The given method would return the index of the first occurrence of character 'c' after the integer index passed as second parameter "startindex." All the occurrences of character 'c' before the "startindex" integer index would be ignored.

**IndexOf(String substring)**

The above method returns the index of the first character of the substring passed as a parameter to it. If that substring is not available in the string, the returned index would be -1.

**IndexOf(String substring, int startindex)**

This Java method returns the index of the first character in the substring passed as the first parameter, after the "startindex" index value. If substring starts from the passed integer value of "startindex", that substring would be ignored.

**compareTo()** is used for comparing two strings lexicographically. Each character of both strings are converted into a Unicode value. However, if both the strings are equal, then this method returns 0 else it only result either negative or positive value.

You can use method Use "compareToIgnoreCase" in case you don't want the result to be case sensitive.

**contains()** method is Java method to check if String contains another substring or not. It returns boolean value so it can use directly inside if statements.

It is a common case in programming when you want to check if specific String contains a particular substring.

**endsWith()** method is used to check whether the string is ending with user-specified substring or not. Based on this comparison it will return boolean True or False.

**replace()** method replaces every occurrence of a given character with a new character and returns a new string. This method allows the replacement of a sequence of character values.

**replaceAll()** method finds all occurrences of sequence of characters matching a regular expression and replaces them with the replacement string. At the end of call, a new string is returned by the function.

|  |
| --- |
| String str2 = str.replaceAll("\\s", ""); |

**replaceFirst()** method replaces ONLY the first substring which matches a given regular expression. Matching of the string starts from the beginning of a string (left to right). At the end of call, a new string is returned by the function.

|  |
| --- |
| String str1 = str.replaceFirst("s", "9"); |

## Java Strings are Immutable

In Java, creating a string means creating an object of the String class. When we create a string, we cannot change that string in Java. This is why strings are called immutable in Java.

Whenever modifying the String the JVM actually creates new string and variable now points to new String and previous String remains unchanged.

strings are created using string literals and the new keyword

|  |
| --- |
| String greet = "Hello! World";  // create a string using the new keyword  String name = new String("java string"); |

In Java, the JVM maintains a string pool to store all of its strings inside the memory. The string pool helps in reusing the strings.

While creating strings using string literals, the value of the string is directly provided. Hence, the compiler first checks the string pool to see if the string already exists.

If the string already exists, the new string is not created. Instead, the new reference points to the existing string.

If the string doesn't exist, the new string is created.

However, while creating strings using the new keyword, the value of the string is not directly provided. Hence the new string is created all the time.

## Convert String to Integer

If you have a string that contains a numeric value. If you try to perform some arithmetic operation, throws compilation error.Hence, you need to convert a String to int before you peform numeric operations on it.

There are two ways to convert String to Integer in Java,

* String to Integer using Integer.parseInt()
* String to Integer using Integer.valueOf()

NumberFormatException is thrown If you try to parse an invalid number string. For example, String ‘Guru99’ cannot be converted into Integer.

|  |
| --- |
| int iTest = Integer.parseInt(strTest);  //Convert the String to Integer using Integer.valueOf  int iTest = Integer.valueOf(strTest); |

# JAVA KEYWORDS

## this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor.

There are various situations where this keyword is commonly used.

**Using this for Ambiguity Variable Names**

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name

In such a situation, we use this keyword.

|  |
| --- |
| class Main {  int age;  Main(int age){  this.age = age;  }  public static void main(String[] args) {  Main obj = new Main(8);  System.out.println("obj.age = " + obj.age);  }  } |

**this with Getters and Setters**

Another common use of this keyword is in setters and getters methods of a class

to assign value inside the setter method

to access value inside the getter method

|  |
| --- |
| class Main {  String name;  // setter method  void setName( String name ) {  this.name = name;  }  // getter method  String getName(){  return this.name;  }  public static void main( String[] args ) {  Main obj = new Main();  // calling the setter and the getter method  obj.setName("Toshiba");  System.out.println("obj.name: "+obj.getName());  }  } |

**Using this in Constructor Overloading**

While working with constructor overloading, we might have to invoke one constructor from another constructor. In such case, we cannot call the constructor explicitly. Instead we have to use this keyword.

Invoking one constructor from another constructor is called explicit constructor invocation.

|  |
| --- |
| class Complex {  private int a, b;  // constructor with 2 parameters  private Complex( int i, int j ){  this.a = i;  this.b = j;  }  // constructor with single parameter  private Complex(int i){  // invokes the constructor with 2 parameters  this(i, i);  }  // constructor with no parameter  private Complex(){  // invokes the constructor with single parameter  this(0);  }  @Override  public String toString(){  return this.a + " + " + this.b + "i";  }  public static void main( String[] args ) {  // creating object of Complex class  // calls the constructor with 2 parameters  Complex c1 = new Complex(2, 3);    // calls the constructor with a single parameter  Complex c2 = new Complex(3);  // calls the constructor with no parameters  Complex c3 = new Complex();  // print objects  System.out.println(c1);  System.out.println(c2);  System.out.println(c3);  }  } |

huge advantage of using this() is to reduce the amount of duplicate code.

**Passing this as an Argument**

We can use this keyword to pass the current object as an argument to a method

|  |
| --- |
| class ThisExample {  // declare variables  int x;  int y;  ThisExample(int x, int y) {  // assign values of variables inside constructor  this.x = x;  this.y = y;  // value of x and y before calling add()  System.out.println("Before passing this to addTwo() method:");  System.out.println("x = " + this.x + ", y = " + this.y);  // call the add() method passing this as argument  add(this);  // value of x and y after calling add()  System.out.println("After passing this to addTwo() method:");  System.out.println("x = " + this.x + ", y = " + this.y);  }  void add(ThisExample o){  o.x += 2;  o.y += 2;  }  }  class Main {  public static void main( String[] args ) {  ThisExample obj = new ThisExample(1, -2);  }  } |

## final keyword

the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is

* the final variable cannot be reinitialized with another value
* the final method cannot be overridden
* the final class cannot be extended

**final Variable**

If we have tried to change the value of the final variable, we will get a compilation error

It is recommended to use uppercase to declare final variables in Java.

|  |
| --- |
| final int AGE = 32; |

**final Method**

the final method cannot be overridden by the child class.

|  |
| --- |
| public final void display() {  System.out.println("The final method is overridden.");  } |

**final Class**

the final class cannot be inherited by another class

|  |
| --- |
| final class FinalClass {  // create a final method  public void display() {  System.out.println("This is a final method.");  }  } |

## instanceof keyword

the instanceof keyword is a binary operator. It is used to check whether an object is an instance of a particular class or not.

|  |
| --- |
| boolean result = objectName instanceof className; |

The result will be true if an object is an instance of a class and false if it is not.

**Use of instanceof in Inheritance**

In the case of inheritance, the instanceof operator is used to check whether an object of a subclass is also an instance of the superclass.

|  |
| --- |
| class Animal {  }  // Dog class is a subclass of Animal  class Dog extends Animal {  }  class Main {  public static void main(String[] args){  Dog d1 = new Dog();  // checks if d1 is an object of Dog  System.out.println("Is d1 an instance of Dog: "+ (d1 instanceof Dog));    // checks if d1 is an object of Animal  System.out.println("Is d1 an instance of Animal: "+ (d1 instanceof Animal));  }  } |

All the classes are inherited from the Object class. The extends keyword is not used during the inheritance of the Object class. This inheritance happens implicitly in. Hence All Objects are instanceOf Object Class

**Object Upcasting and Downcasting**

In Java, an object of a subclass can be treated as an object of the superclass. This is called upcasting.

Java compiler automatically performs upcasting.

|  |
| --- |
| class Animal {  public void displayInfo() {  System.out.println("I am an animal.");  }  }  class Dog extends Animal {  }  class Main {  public static void main(String[] args) {  Dog d1 = new Dog();  Animal a1 = d1;  a1.displayInfo();  }  } |

Downcasting is a reverse procedure of upcasting.

In the case of downcasting, an object of the superclass is treated as an object of the subclass. We have to explicitly instruct the compiler to downcast in Java.

|  |
| --- |
| class Animal {  }  class Dog extends Animal {  public void displayInfo() {  System.out.println("I am a dog");  }  }  class Main {  public static void main(String[] args) {  Dog d1 = new Dog();  Animal a1 = d1; // Upcasting  if (a1 instanceof Dog){  Dog d2 = (Dog)a1; // Downcasting  d2.displayInfo();  }  }  } |

superclass may also refer to other subclasses. Had we created two subclasses; the Super class object maybe subclass1 or it maybe subclass2 causing ambiguity. To resolve this problem we can use the instanceof operator to check whether the a1 object is an instance of Dog class or not. The downcasting is done only when the expression a1 instanceof Dog is true.

**instanceof in Interface**

The instanceof operator is also used to check whether an object of a class is also an instance of the interface from where the class implements.

|  |
| --- |
| interface Animal {  }  class Dog implements Animal {  }  class Main {  public static void main(String[] args) {  Dog d1 = new Dog();  System.out.println("Is d1 an instance of Animal: "+(d1 instanceof Animal));  }  } |

## super keyword

The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

**Uses of super keyword**

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

|  |
| --- |
| //Access Overridden Methods of the superclass  super.displayInfo();  //Access Attributes of the Superclass  super.type;  //Access superclass constructor  super();  // calling parameterized constructor of the superclass  super("Animal"); |

As we know, when an object of a class is created, its default constructor is automatically called. To explicitly call the superclass constructor from the subclass constructor, we use super(). It's a special form of the super keyword.

super() can be used only inside the subclass constructor and must be the first statement.

The compiler can automatically call the no-arg constructor. If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.

# OOPs(Object-Oriented Programming System)

As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.



## Inheritance

we use the extends keyword to inherit from a class.

Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.

Here are some examples:

A car is a vehicle.

Orange is a fruit.

A surgeon is a doctor.

A dog is an animal.

|  |
| --- |
| class Dog extends Animal {  public void bark() {  System.out.println("I can bark");  }  } |

Objects of the sub class can access the members of both the sub class and the Super class.

### Types of inheritance

There are five types of inheritance.

**Single inheritance** - Class B extends from class A only.

**Multilevel inheritance** - Class B extends from class A; then class C extends from class B.

**Hierarchical inheritance** - Class A acts as the superclass for classes B, C, and D.

***Multiple inheritance***- Class C extends from interfaces A and B.

***Hybrid inheritance*** - Mix of two or more types of inheritance.

Java doesn’t support multiple and hybrid inheritance through classes. However, we can achieve multiple inheritance in Java through interfaces.

**Why use inheritance?**

The most important use is the reusability of code. The code that is present in the parent class doesn’t need to be written again in the child class.

To achieve runtime polymorphism through method overriding.

## Polymorphism

It simply means more than one form. That is, the same entity (method or operator or object) behaves differently in different scenarios.

The + operator in Java is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition. And when we use + operator with strings, it performs string concatenation.

**Types of Polymorphism**

In Java, Polymorphism can be divided into two types:

Run-time Polymorphism

Compile-time Polymorphism

**Run-time Polymorphism**

In Java, run-time polymorphism can be achieved through method overriding.

Suppose the same method is created in the superclass and its subclasses. In this case, the method that will be called depends upon the object used to call the method.

**Compile-time Polymorphism**

The compile-time polymorphism can be achieved through method overloading and operator overloading in Java

### Method overriding

objects of a subclass can also access methods of its superclass.

What happens if the same method is defined in both the superclass and subclass?

Well, in that case, the method in the subclass overrides the method in the superclass.

**Java Overriding Rules**

1. Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
2. We cannot override the method declared as final and static.
3. We should always override abstract methods of the superclass

**Access Specifiers in Method Overriding**

The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.

Suppose, a method myClass() in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

### Method Overloading

Two or more methods can have same name inside the same class if they accept different arguments. This feature is known as method overloading.

|  |
| --- |
| void func() { ... }  void func(int a) { ... }  float func(double a) { ... }  float func(int a, float b) { ... } |

Method overloading is achieved by either:

1. changing the number of arguments.
2. or changing the datatype of arguments.

Method overloading is not possible by changing the return type of methods.

## Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user.

In java, abstraction is achieved by interfaces and abstract classes.

### Abstract Class and Abstract Methods

We use the abstract keyword to create abstract classes and methods.

An abstract method doesn't have any implementation (method body).

A class containing abstract methods should also be abstract.

We cannot create objects of an abstract class.

To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.

A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.

We can access the static attributes and methods of an abstract class using the reference of the abstract class.

only an abstract class can contain abstract methods. If we include abstract methods inside a class that is not abstract, we will get an error.

An abstract class can contain both abstract and non-abstract methods.

|  |
| --- |
| abstract class Animal {  public void displayInfo() {  System.out.println(“I am an animal.”);  }  abstract void makeSound();  } |

### Interface

An interface defines a set of specifications that other classes must implement.

An interface can include abstract methods and constants.

It is important to note that, all methods inside an interface are implicitly public and all fields are implicitly public static final. Hence, it's not necessary to specify the access specifier inside interfaces.

|  |
| --- |
| interface Polygon {  public static final String color = "blue";    public void getArea();  } |

Like abstract classes, we cannot create objects of interfaces. However, we can implement interfaces in other classes. In Java, we use the implements keyword to implement interfaces.

|  |
| --- |
| class Rectangle implements Polygon {  public void getArea(int length, int breadth) {  System.out.println("The area of the rectangle is " + (length \* breadth));  }  } |

Interfaces are also used to achieve multiple inheritance in Java. If a subclass is inherited from two or more classes, it's multiple inheritance.

|  |
| --- |
| class Rectangle implements Line, Polygon{  ...  } |

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces

|  |
| --- |
| interface Line {  //members of Line interface  }  interface Polygon extends Line {  //members of Polygon interface and Line interface  } |

The interface Polygon extends the Line interface. Now, if a class implements Polygon, it should provide implementations for all abstract classes of both Line and Polygon.

Note that an interface can extend multiple interfaces similar to a class implementing multiple interfaces.

|  |
| --- |
| interface A {  ...  }  interface B {  ...  }  Interface C extends A, B {  ...  } |

## Encapsulation

Encapsulation refers to the bundling of fields and methods inside a single class. Bundling similar fields and methods inside a class together also helps in data hiding.

Data hiding can be achieved with the help of access modifiers.

Encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.

It helps to decouple components of a system. These decoupled components can be developed, tested and debugged independently and concurrently. And, any changes in a particular component do not have any effect on other components.

### Access Modifiers

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods

Access modifiers are mainly used for encapsulation. I can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.

**Types of Access Modifier**

There are four access modifiers keywords in Java and they are:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| Default | declarations are visible only within the package (package private) |
| Private | declarations are visible within the class only |
| Protected | declarations are visible within the package or all subclasses |
| Public | declarations are visible everywhere |

**Default Access Modifier**

If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered.

**Private Access Modifier**

When variables and methods are declared private, they cannot be accessed outside of the class.

We cannot declare classes and interfaces private in Java. However, the nested classes can be declared private.

**Protected Access Modifier**

When methods and data members are declared protected, we can access them within the same package as well as from subclasses.

We cannot declare classes or interfaces protected in Java.

**Public Access Modifier**

When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction.

|  | Class | Package | subclass | World |
| --- | --- | --- | --- | --- |
| public | Yes | Yes | Yes | Yes |
| private | Yes | No | No | No |
| protected | Yes | Yes | Yes | No |

## Class

### Nested and Inner Class

you can define a class within another class. Such class is known as nested class.

|  |
| --- |
| class OuterClass {  // ...  class NestedClass {  // ...  }  } |

* Java treats the inner class as a regular member of a class. They are just like methods and variables declared inside a class.
* Since inner classes are members of the outer class, you can apply any access modifiers like private, protected to your inner class which is not possible in normal classes.
* Since the nested class is a member of its enclosing outer class, you can use the dot (.) notation to access the nested class and its members.
* Using the nested class will make your code more readable and provide better encapsulation.
* Non-static nested classes (inner classes) have access to other members of the outer/enclosing class, even if they are declared private.

There are two types of nested classes you can create in Java.

1. Non-static nested class (inner class)
2. Static nested class

#### Non-Static Nested Class (Inner Class)

A non-static nested class is a class within another class. It has access to members of the enclosing class (outer class). It is commonly known as inner class.

Since the inner class exists within the outer class, you must instantiate the outer class first, in order to instantiate the inner class.

We use the dot (.) operator to create an instance of the inner class using the outer class.

|  |
| --- |
| class CPU {  double price;  // nested class  class Processor{  // members of nested class  double cores;  String manufacturer;  double getCache(){  return 4.3;  }  }  // nested protected class  protected class RAM{  // members of protected nested class  double memory;  String manufacturer;  double getClockSpeed(){  return 5.5;  }  }  }  public class Main {  public static void main(String[] args) {  // create object of Outer class CPU  CPU cpu = new CPU();  // create an object of inner class Processor using outer class  CPU.Processor processor = cpu.new Processor();  // create an object of inner class RAM using outer class CPU  CPU.RAM ram = cpu.new RAM();  System.out.println("Processor Cache = " + processor.getCache());  System.out.println("Ram Clock speed = " + ram.getClockSpeed());  }  } |

Inside the Main class,

we first created an instance of an outer class CPU named cpu.

Using the instance of the outer class, we then created objects of inner classes

We can access the members of the outer class by using **this** keyword.

#### Static Nested Class

we can also define a static class inside another class. Such class is known as static nested class. Static nested classes are not called static inner classes.

Unlike inner class, a static nested class cannot access the member variables of the outer class. It is because the static nested class doesn't require you to create an instance of the outer class.

|  |
| --- |
| class MotherBoard {  // static nested class  static class USB{  int usb2 = 2;  int usb3 = 1;  int getTotalPorts(){  return usb2 + usb3;  }  }  }  public class Main {  public static void main(String[] args) {  // create an object of the static nested class  // using the name of the outer class  MotherBoard.USB usb = new MotherBoard.USB();  System.out.println("Total Ports = " + usb.getTotalPorts());  }  } |

### Anonymous Class

A nested class that doesn't have any name is known as an anonymous class. An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class.

|  |
| --- |
| class outerClass {  // defining anonymous class  object1 = new Type(parameterList) {  // body of the anonymous class  }; |

Here, Type can be

a superclass that an anonymous class extends

an interface that an anonymous class implements

Note: Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

#### Anonymous Class Extending a Class

|  |
| --- |
| class Polygon {  public void display() {  System.out.println("Inside the Polygon class");  }  }  class AnonymousDemo {  public void createClass() {  // creation of anonymous class extending class Polygon  Polygon p1 = new Polygon() {  public void display() {  System.out.println("Inside an anonymous class.");  }  };  p1.display();  }  }  class Main {  public static void main(String[] args) {  AnonymousDemo an = new AnonymousDemo();  an.createClass();  }  } |

created an anonymous class that extends the class Polygon and overrides the display() method. When we run the program, an object p1 of the anonymous class is created. The object then calls the display() method of the anonymous class.

Anonymous Class Implementing an Interface

|  |
| --- |
| interface Polygon {  public void display();  }  class AnonymousDemo {  public void createClass() {  // anonymous class implementing interface  Polygon p1 = new Polygon() {  public void display() {  System.out.println("Inside an anonymous class.");  }  };  p1.display();  }  }  class Main {  public static void main(String[] args) {  AnonymousDemo an = new AnonymousDemo();  an.createClass();  }  } |

we have created an anonymous class that implements the Polygon interface

**Advantages of Anonymous Classes**

1. In anonymous classes, objects are created whenever they are required. That is, objects are created to perform some specific tasks.
2. Anonymous classes also help us to make our code concise.

### Singleton Class

Singleton is a design pattern rather than a feature specific to Java. It ensures that only one instance of a class is created.

A design pattern is like our code library that includes various coding techniques shared by programmers around the world.

Here's how we can use singletons in Java.

* create a **private constructor** that restricts to create an object outside of the class
* create a **private attribute** that refers to the singleton object.
* create a **public static method** that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

|  |
| --- |
| class SingletonExample {  // private field that refers to the object  private static SingletonExample singleObject;  private SingletonExample() {  // constructor of the SingletonExample class  }  public static SingletonExample getInstance() {  // write code that allows us to create only one object  // access the object as per our need  }  } |

Singletons can be used while working with databases. They can be used to create a connection pool to access the database while reusing the same connection for all the clients.

|  |
| --- |
| class Database {  private static Database dbObject;  private Database() {  }  public static Database getInstance() {  // create object if it's not already created  if(dbObject == null) {  dbObject = new Database();  }  // returns the singleton object  return dbObject;  }  public void getConnection() {  System.out.println("You are now connected to the database.");  }  }  class Main {  public static void main(String[] args) {  Database db1;  // refers to the only object of Database  db1= Database.getInstance();    db1.getConnection();  }  } |

Since the Database can have only one object, all the clients can access the database through a single connection.

### enum Class

An enum (short for enumeration) is a type that has a fixed set of possible values. We use the enum keyword to declare enums

|  |
| --- |
| enum Size{  SMALL, MEDIUM, LARGE, EXTRALARGE;  public String getSize() {  // this will refer to the object SMALL  switch(this) {  case SMALL:  return "small";  case MEDIUM:  return "medium";  case LARGE:  return "large";  case EXTRALARGE:  return "extra large";  default:  return null;  }  }  public static void main(String[] args) {  // calling the method getSize() using the object SMALL  System.out.println("The size of the pizza is " + Size.SMALL.getSize());  }  } |

The values inside the braces are called enum values (constants). These are the only values that the enum type can hold.

Note: The enum constants are usually represented in uppercase.

The enum constants are always public static final by default.

An enum class can include methods and fields just like regular classes.

When an enum class is created, the compiler will also create instances (objects) of each enum constants.

There are some predefined methods in enum classes that are readily available for use.

ordinal() method

The ordinal() method returns the position of an enum constant.

compareTo() Method

The compareTo() method compares the enum constants based on their ordinal value.

toString() Method

The toString() method returns the string representation of the enum constants.

name() Method

The name() method returns the defined name of an enum constant in string form. The returned value from the name() method is final

valueOf() Method

The valueOf() method takes a string and returns an enum constant having the same string name

values() Method

The values() method returns an array of enum type containing all the enum constants

# Java Exceptions

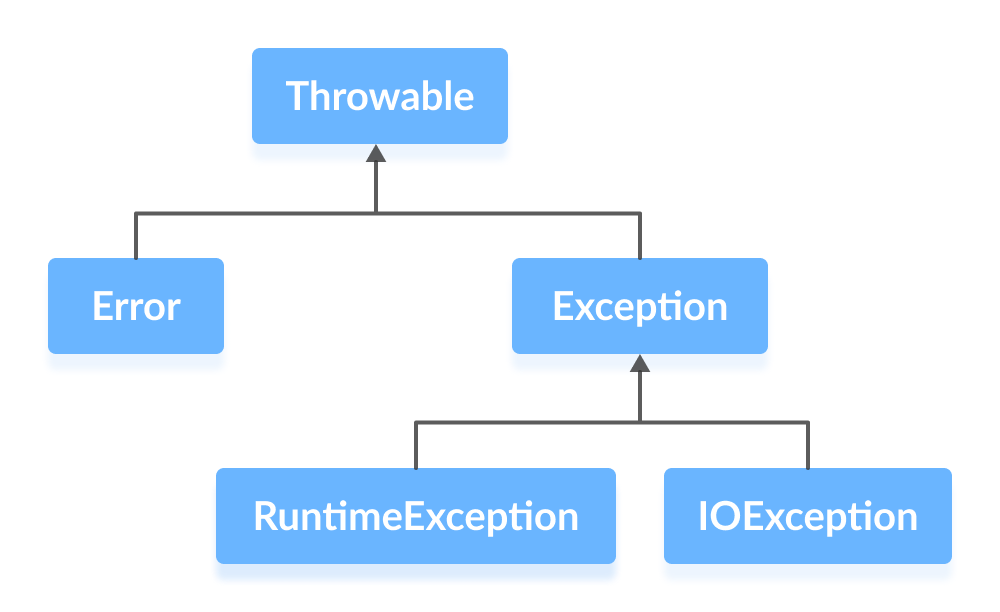
An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

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

**Java Exception hierarchy**

Here is a simplified diagram of the exception hierarchy in Java.



## Errors

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.

## Exceptions

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 state of the program when the exception occurred.

**Java Exception Types**

The exception hierarchy also has two branches: RuntimeException and IOException.

### 1. RuntimeException

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

* Improper use of an API - IllegalArgumentException
* Null pointer access (missing the initialization of a variable) - NullPointerException
* Out-of-bounds array access - ArrayIndexOutOfBoundsException
* Dividing a number by 0 – ArithmeticException

You can think about it in this way. “If it is a runtime exception, it is your fault”.

* The NullPointerException would not have occurred if you had checked whether the variable was initialized or not before using it.
* An ArrayIndexOutOfBoundsException would not have occurred if you tested the array index against the array bounds.

#### Java Exception examples

1: **Arithmetic exception**

Class: Java.lang.ArithmeticException

This is a built-in-class present in java.lang package. This exception occurs when an integer is divided by zero.

2: **ArrayIndexOutOfBounds Exception**

Class: Java.lang.ArrayIndexOutOfBoundsException

This exception occurs when you try to access the array index which does not exist.

3: **NumberFormat Exception**

Class: Java.lang.NumberFormatException

This exception occurs when a string is parsed to any numeric variable.

4: **StringIndexOutOfBound Exception**

Class: Java.lang.StringIndexOutOfBoundsException

An object of this class gets created whenever an index is invoked of a string, which is not in the range. Each character of a string object is stored in a particular index starting from 0.To get a character present in a particular index of a string we can use a method charAt(int) of java.lang.String where int argument is the index.

5: **NullPointer Exception**

Class: Java.lang.NullPointer Exception

An object of this class gets created whenever a member is invoked with a “null” object.

### 2. IOException

An IOException is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

* Trying to open a file that doesn’t exist results in FileNotFoundException
* Trying to read past the end of a file

## Java Exception Handling

To handle exceptions, we will use try...catch...finally blocks.

|  |
| --- |
| try {  // code  } catch (ExceptionType e) {  // catch block  } finally {  // finally block  } |

**Try Block**

The code that might generate an exception is placed in the try block.

Every try block should be immediately followed by the catch or finally block. When an exception occurs, it is caught by the catch block that immediately follows it.

**catch blocks**

catch blocks cannot be used alone and must always be preceded by a try block.

For each try block, there can be zero or more catch blocks. The argument type of each catch block indicates the type of exception that can be handled by it. Multiple catch blocks allow us to handle each exception differently.

**finally block**

For each try block, there can be only one finally block. The finally block is optional. However, if defined, it is always executed (even if the exception doesn't occur). If an exception occurs, it is executed after the try...catch block. If no exception occurs, it is executed after the try block. However, there are some cases when a finally block does not execute:

Use of System.exit() method

An exception occurs in the finally block

The death of a thread

Each exception type that can be handled by the catch block is separated using a vertical bar |.

|  |
| --- |
| try {  // code  } catch (ExceptionType1 | Exceptiontype2 ex) {  // catch block  } |

Catching multiple exceptions in a single catch block reduces code duplication and increases efficiency.

## Handle checked exceptions using throw and throws.

### Java throws keyword

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause so that methods further up in the call stack can handle them or specify them using throws keyword themselves.

|  |
| --- |
| import java.io.\*;  class Main {  public static void findFile() throws IOException {  // code that may produce IOException  File newFile=new File("test.txt");  FileInputStream stream=new FileInputStream(newFile);  }  public static void main(String[] args) {  try{  findFile();  } catch(IOException e){  System.out.println(e);  }  }  } |

throws is also useful when you have checked exception (an exception that must be handled) that you don't want to catch in your current method.

### Java throw keyword

The throw keyword is used to explicitly throw a single exception.

|  |
| --- |
| class Main {  public static void divideByZero() {  throw new ArithmeticException("Trying to divide by 0");  }  public static void main(String[] args) {  divideByZero();  }  } |

# JAVA Collections

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

We do not have to write code to implement these data structures and algorithms manually.

Our code will be much more efficient as the collections framework is highly optimized.

The java.util package contains all the classes and interfaces for the Collection framework.



## Java Collection Interface

The Collection interface is the root interface of the Java collections framework. There is no direct implementation of this interface. However, it is implemented through its subinterfaces like List, Set, and Queue.

The Collection interface is the root interface of the collections framework hierarchy.

Java does not provide direct implementations of the Collection interface but provides implementations of its subinterfaces like List, Set, and Queue.

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.

**Methods**

**add()** - inserts the specified element to the collection

**size()** - returns the size of the collection

**remove()** - removes the specified element from the collection

**iterator()** - returns an iterator to access elements of the collection

**addAll()** - adds all the elements of a specified collection to the collection

**removeAll()** - removes all the elements of the specified collection from the collection

**clear()** - removes all the elements of the collection

### Java List Interface

In Java, the List interface is an ordered collection that allows us to store and access elements *sequentially*. It extends the Collection interface.

We must import java.util.List package in order to use List.

|  |
| --- |
| // ArrayList implementation of List  List<String> list1 = new ArrayList<>();  // LinkedList implementation of List  List<String> list2 = new LinkedList<>(); |

**Methods of List**

The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List.

**add()** - adds an element to a list

**addAll()** - adds all elements of one list to another

**get()** - helps to randomly access elements from lists

**iterator()** - returns iterator object that can be used to sequentially access elements of lists

**set()** - changes elements of lists

**remove()** - removes an element from the list

**removeAll()** - removes all the elements from the list

**clear()**-removes all the elements from the list (more efficient than removeAll() )

**size()** - returns the length of lists

**toArray()** - converts a list into an array

**contains()** - returns true if a list contains specified element

In order to use functionalities of the List interface, we can use these classes:

1. ArrayList
2. LinkedList
3. Vector
4. Stack

#### ArrayList Class

The ArrayList class is an implementation of the List interface that allows us to create resizable-arrays.

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.To handle this issue, we can use the ArrayList class. The ArrayList class present in the java.util package allows us to create resizable arrays. Unlike arrays, array lists (objects of the ArrayList class) can automatically adjust its capacity when we add or remove elements from it. Hence, array lists are also known as dynamic arrays.

|  |
| --- |
| // create Integer type arraylist  ArrayList<Integer> arrayList = new ArrayList<>();  // create String type arraylist  ArrayList<String> arrayList = new ArrayList<>(); |

We can also create array lists using the List interface. It's because the ArrayList class implements the List interface.

|  |
| --- |
| List<String> list = new ArrayList<>(); |

We can not create array lists of primitive data types like int, float, char, etc. Instead, we have to use their corresponding wrapper class.In the case of strings, String is a class and doesn't have a wrapper class. Hence, we have used String as it is.

**Methods of ArrayList**

ArrayList provides various methods that allow us to perform array list operations.

**add() -** To add a single element to the array list, we use the add() method.

|  |
| --- |
| ArrayList<String> animals = new ArrayList<>();  // Add elements  animals.add("Dog"); |

We can also add elements to an array list using indexes

|  |
| --- |
| animals.add(2,"Horse"); |

**addAll()** -To add all the elements of an array list to a new array list

|  |
| --- |
| ArrayList<String> mammals = new ArrayList<>();  mammals.add("Dog");  mammals.add("Cat");  mammals.add("Horse");  ArrayList<String> animals = new ArrayList<>();  animals.add("Crocodile");  // Add all elements of mammals in animals  animals.addAll(mammals); |

**asList()**

Unlike arrays, we cannot initialize array lists directly. However, we can use the asList() method of the Arrays class to achieve the same effect.

In order to use the asList() method, we must import the java.util.Arrays package first.

|  |
| --- |
| // Creating an array list  ArrayList<String> animals = new ArrayList<>(Arrays.asList("Cat", "Cow", "Dog")); |

The asList() method is used to convert the array into an array list.

**get()**

To randomly access elements of an array list, we use the get() method

|  |
| --- |
| // Get the element from the array list  String str = animals.get(0); |

**iterator()**

To sequentially access elements of an array list, we use the iterator() method. We must import java.util.Iterator package to use this method

|  |
| --- |
| // Create an object of Iterator  Iterator<String> iterate = animals.iterator();  // Use methods of Iterator to access elements  while(iterate.hasNext()){  System.out.print(iterate.next());  System.out.print(", "); |

hasNext() returns true if there is a next element in the array list.

next() returns the next element in the array list

**set()**

To update elements of an array list, we can use the set() method

|  |
| --- |
| // Change the element of the array list  animals.set(2, "Zebra"); |

**remove()**

To remove an element from an array list, we can use the remove() method

|  |
| --- |
| // Remove element from index 2  String str = animals.remove(2); |

**removeAll()**

To remove all elements from an array list, we use the removeAll() method

|  |
| --- |
| // Remove all the elements - here animals is ArrayList  animals.removeAll(animals); |

**clear()**

We can also use the clear() method to remove all elements from an array list.

|  |
| --- |
| // Remove all the elements  animals.clear(); |

**size()**

To get the length of the array list, we use the size() method

|  |
| --- |
| // getting the size of the arrayList  animals.size(); |

**clone()**

Creates a new array list with the same element, size, and capacity.

**contains()**

Searches the array list for the specified element and returns a boolean result.

**ensureCapacity()**

Specifies the total element the array list can contain.

**isEmpty()**

Checks if the array list is empty.

**indexOf()**

Searches a specified element in an array list and returns the index of the element.

**trimToSize()**

Reduces the capacity of an array list to its current size.

**Loop Through an ArrayList**

1. Using for loop

|  |
| --- |
| for(int i = 0; i < animals.size(); i++) {  System.out.print(animals.get(i));  System.out.print(", ");  } |

2. Using the forEach loop

|  |
| --- |
| for(String animal : animals) {  System.out.print(animal);  System.out.print(", ");  } |

**Sort Elements of an ArrayList**

To sort elements of an array list, we use the sort() method of the Collections class. In order to use it, we must import the java.util.Collections package first.

By default, the sorting occurs either alphabetically or numerically in ascending order.

|  |
| --- |
| // Sort the array list  Collections.sort(animals); |

**Java ArrayList To Array**

In Java, we can convert array lists into arrays using the toArray() method

|  |
| --- |
| // Create a new array of String type  String[] arr = new String[animals.size()];  // Convert ArrayList into an array  animals.toArray(arr); |

**Java Array to ArrayList**

We can also convert arrays into array lists. For that, we can use the asList() method of the Arrays class.

To use asList(), we must import the java.util.Arrays package first

|  |
| --- |
| // Create an array of String type  String[] arr = {"Dog", "Cat", "Horse"};    // Create an ArrayList from an array  ArrayList<String> animals = new ArrayList<>(Arrays.asList(arr)); |

**Java ArrayList to String**

To convert an array list into a String, we can use the toString() method

|  |
| --- |
| // Convert ArrayList into an String  String str = animals.toString(); |

#### Java Vector Class

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.

|  |
| --- |
| // create Integer type linked list  Vector<Integer> vector= new Vector<>();  // create String type linked list  Vector<String> vector= new Vector<>(); |

**add(element)** - adds an element to vectors

**add(index, element)** - adds an element to the specified position

**addAll(vector)** - adds all elements of a vector to another vector

**get(index)** - returns an element specified by the index

**iterator()** - returns an iterator object to sequentially access vector elements

**remove(index)** - removes an element from specified position

**removeAll()** - removes all the elements

**clear()** - removes all elements. It is more efficient than removeAll()

**set()** changes an element of the vector

**size()** returns the size of the vector

**toArray()** converts the vector into an array

**toString()** converts the vector into a String

**contains()** searches the vector for specified element and returns a boolean result

#### Stack Class

The Stack class extends the Vector class. In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack. In order to create a stack, we must import the java.util.Stack package first.

|  |
| --- |
| // Create Integer type stack  Stack<Integer> stacks = new Stack<>();  // Create String type stack  Stack<String> stacks = new Stack<>(); |

Since Stack extends the Vector class, it inherits all the methods Vector.Besides these methods, the Stack class includes 5 more methods that distinguish it from Vector.

**push()**

To add an element to the top of the stack, we use the push() method.

|  |
| --- |
| Stack<String> animals= new Stack<>();  // Add elements to Stack  animals.push("Dog");  animals.push("Horse");  animals.push("Cat"); |

**pop()**

To remove an element from the top of the stack, we use the pop() method.

|  |
| --- |
| // Remove element stacks  String element = animals.pop(); |

**peek()**

The peek() method returns an object from the top of the stack.

|  |
| --- |
| // Access element from the top  String element = animals.peek(); |

**search()**

To search an element in the stack, we use the search() method. It returns the position of the element from the top of the stack

|  |
| --- |
| // Search an element  int position = animals.search("Horse"); |

**empty()**

To check whether a stack is empty or not, we use the empty() method

|  |
| --- |
| // Check if stack is empty  boolean result = animals.empty(); |

The Stack class provides the direct implementation of the stack data structure. However, it is recommended not to use it. Instead, use the ArrayDeque class (implements the Deque interface) to implement the stack data structure in Java.

### Queue Interface

In order to use the functionalities of Queue, we need to use classes that implement it:

* ArrayDeque
* LinkedList
* PriorityQueue

The Queue interface is also extended by various subinterfaces:

* Deque
* BlockingQueue
* BlockingDeque

In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.

In Java, we must import java.util.Queue package in order to use Queue.

|  |
| --- |
| // LinkedList implementation of Queue  Queue<String> animal1 = new LinkedList<>();  // Array implementation of Queue  Queue<String> animal2 = new ArrayDeque<>();  // Priority Queue implementation of Queue  Queue<String> animal 3 = new PriorityQueue<>(); |

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

element() - Returns the head of the queue. Throws an exception if the queue is empty.

peek() - Returns the head of the queue. Returns null if the queue is empty.

remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

#### PriorityQueue Class

Unlike normal queues, priority queue elements are retrieved in sorted order. Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

In order to create a priority queue, we must import the java.util.PriorityQueue package.

|  |
| --- |
| PriorityQueue<Integer> numbers = new PriorityQueue<>(); |

add() - Inserts the specified element to the queue. If the queue is full, it throws an exception.

offer() - Inserts the specified element to the queue. If the queue is full, it returns false.

peek()-To access elements from a priority queue

remove() - removes the specified element from the queue

poll() - returns and removes the head of the queue

contains(element)-Searches the priority queue for the specified element. If the element is found, it returns true, if not it returns false.

size()- Returns the length of the priority queue.

toArray()- Converts a priority queue to an array and returns it.

#### Deque Interface

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.



In order to use the functionalities of the Deque interface, we need to use classes that implement it:

ArrayDeque

LinkedList

we must import the java.util.Deque package to use Deque.

|  |
| --- |
| // Array implementation of Deque  Deque<String> animal1 = new ArrayDeque<>();  // LinkedList implementation of Deque  Deque<String> animal2 = new LinkedList<>(); |

Besides methods available in the Queue interface, the Deque interface also includes the following methods:

addFirst() - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.

addLast() - Adds the specified element at the end of the deque. Throws an exception if the deque is full.

offerFirst() - Adds the specified element at the beginning of the deque. Returns false if the deque is full.

offerLast() - Adds the specified element at the end of the deque. Returns false if the deque is full.

getFirst() - Returns the first element of the deque. Throws an exception if the deque is empty.

getLast() - Returns the last element of the deque. Throws an exception if the deque is empty.

peekFirst() - Returns the first element of the deque. Returns null if the deque is empty.

peekLast() - Returns the last element of the deque. Returns null if the deque is empty.

removeFirst() - Returns and removes the first element of the deque. Throws an exception if the deque is empty.

removeLast() - Returns and removes the last element of the deque. Throws an exception if the deque is empty.

pollFirst() - Returns and removes the first element of the deque. Returns null if the deque is empty.

pollLast() - Returns and removes the last element of the deque. Returns null if the deque is empty.

##### ArrayDeque Class

The ArrayDeque class implements these two interfaces:

* Java Queue Interface
* Java Deque Interface

In order to create an array deque, we must import the java.util.ArrayDeque package.

|  |
| --- |
| // Creating String type ArrayDeque  ArrayDeque<String> animals = new ArrayDeque<>();  // Creating Integer type ArrayDeque  ArrayDeque<Integer> age = new ArrayDeque<>(); |

add() - inserts the specified element at the end of the array deque

addFirst() - inserts the specified element at the beginning of the array deque

addLast() - inserts the specified at the end of the array deque (equivalent to add())

offer() - inserts the specified element at the end of the array deque

offerFirst() - inserts the specified element at the beginning of the array deque

offerLast() - inserts the specified element at the end of the array deque

getFirst() - returns the first element of the array deque

getLast() - returns the last element of the array deque

peek() - returns the first element of the array deque

peekFirst() - returns the first element of the array deque (equivalent to peek())

peekLast() - returns the last element of the array deque

remove() - returns and removes an element from the first element of the array deque

remove(element) - returns and removes the specified element from the head of the array deque

removeFirst() - returns and removes the first element from the array deque (equivalent to remove())

removeLast() - returns and removes the last element from the array deque

poll() - returns and removes the first element of the array deque

pollFirst() - returns and removes the first element of the array deque (equivalent to poll())

pollLast() - returns and removes the last element of the array deque

clear()-To remove all the elements from the array deque

element()- Returns an element from the head of the array deque.

contains(element)- Searches the array deque for the specified element.

If the element is found, it returns true, if not it returns false.

size() -Returns the length of the array deque.

toArray()- Converts array deque to array and returns it.

clone() - Creates a copy of the array deque and returns it.

push() - adds an element to the top of the stack

pop() - returns and removes an element from the top of the stack

### LinkedList Class

Interfaces implemented by LinkedList

* Java List Interface
* Java Queue Interface
* Java Deque Interface

LinkedList is a linear data structure.LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.

Each element in the LinkedList is called the Node. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

JAVA API has a doubly linked list

|  |
| --- |
| LinkedList<String> list=new LinkedList<String>(); |

add(Object item): It adds the item at the end of the list.

add(int index, Object item): It adds an item at the given index of the the list.

addAll(Collection c): It adds all the elements of the specified collection c to the list.

addAll(int index, Collection c): It adds all the elements of collection c to the list starting from a give index in the list

addFirst(Object item): It adds the item (or element) at the first position in the list.

addLast(Object item): It inserts the specified item at the end of the list.

clear(): It removes all the elements of a list.

clone(): It returns the copy of the list.

contains(Object item): It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

get(int index): It returns the item of the specified index from the list.

getFirst(): It fetches the first item from the list.

getLast(): It fetches the last item from the list.

indexOf(Object item): It returns the index of the specified item.

lastIndexOf(Object item): It returns the index of last occurrence of the specified element.

poll(): It returns and removes the first item of the list.

pollFirst(): same as poll() method. Removes the first item of the list.

pollLast(): It returns and removes the last element of the list.

remove(): It removes the first element of the list.

remove(int index): It removes the item from the list which is present at the specified index.

remove(Object obj): It removes the specified object from the list.

removeFirst(): It removes the first item from the list.

removeLast(): It removes the last item of the list.

emoveFirstOccurrence(Object item): It removes the first occurrence of the specified item.

removeLastOccurrence(Object item): It removes the last occurrence of the given element.

set(int index, Object item): It updates the item of specified index with the give value.

size(): It returns the number of elements of the list.

### Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.

Unlike the List interface, sets cannot contain duplicate elements.

In order to use functionalities of the Set interface, we can use these classes:

HashSet

LinkedHashSet

EnumSet

TreeSet

In Java, we must import java.util.Set package in order to use Set.

|  |
| --- |
| // Set implementation using HashSet  Set<String> animals = new HashSet<>(); |

Some of the commonly used methods of the Collection interface that's also available in the Set interface are:

add() - adds the specified element to the set

addAll() - adds all the elements of the specified collection to the set

iterator() - returns an iterator that can be used to access elements of the set sequentially

remove() - removes the specified element from the set

removeAll() - removes all the elements from the set that is present in another specified set

retainAll() - retains all the elements in the set that are also present in another specified set

clear() - removes all the elements from the set

size() - returns the length (number of elements) of the set

toArray() - returns an array containing all the elements of the set

contains() - returns true if the set contains the specified element

containsAll() - returns true if the set contains all the elements of the specified collection

hashCode() - returns a hash code value (address of the element in the set)

Union - to get the union of two sets x and y, we can use x.addAll(y)

Intersection - to get the intersection of two sets x and y, we can use x.retainAll(y)

Subset - to check if x is a subset of y, we can use y.containsAll(x)

#### HashSet Class

The HashSet class of the Java Collections framework provides the functionalities of the hash table data structure.

In Java, HashSet is commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes.

The hashcode of an element is a unique identity that helps to identify the element in a hash table.

HashSet cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

In order to create a hash set, we must import the java.util.HashSet package first.

|  |
| --- |
| // HashSet with 8 capacity and 0.6 load factor  HashSet<Integer> numbers = new HashSet<>(8, 0.6); |

Here, the first parameter is capacity, and the second parameter is loadFactor.

capacity - The capacity of this hash set is 8. Meaning, it can store 8 elements.

loadFactor - The load factor of this hash set is 0.6. This means, whenever our hash set is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

By default,

the capacity of the hash set will be 16

the load factor will be 0.75

add() - inserts the specified element to the set

addAll() - inserts all the elements of the specified collection to the set or union

remove() - removes the specified element from the set

removeAll() - removes all the elements from the set

retainAll() - To perform the intersection between two sets

containsAll() - To check if a set is a subset of another set or not

clone()- Creates a copy of the HashSet

contains()- Searches the HashSet for the specified element and returns a boolean result

isEmpty()- Checks if the HashSet is empty

size()- Returns the size of the HashSet

clear() -Removes all the elements from the HashSet

#### LinkedHashSet Class

Elements of LinkedHashSet are stored in hash tables similar to HashSet.

However, linked hash sets maintain a doubly-linked list internally for all of its elements. The linked list defines the order in which elements are inserted in hash tables.

In order to create a linked hash set, we must import the java.util.LinkedHashSet package first.

|  |
| --- |
| // LinkedHashSet with 8 capacity and 0.75 load factor  LinkedHashSet<Integer> numbers = new LinkedHashSet<>(8, 0.75); |

#### SortedSet Interface

The SortedSet interface of the Java Collections framework is used to store elements with some order in a set.

To use SortedSet, we must import the java.util.SortedSet package first.

|  |
| --- |
| // SortedSet implementation by TreeSet class  SortedSet<String> animals = new TreeSet<>(); |

The SortedSet interface includes all the methods of the Set interface. It's because Set is a super interface of SortedSet.

Besides methods included in the Set interface, the SortedSet interface also includes these methods:

comparator() - returns a comparator that can be used to order elements in the set

first() - returns the first element of the set

last() - returns the last element of the set

headSet(element) - returns all the elements of the set before the specified element

tailSet(element) - returns all the elements of the set after the specified element including the specified element

subSet(element1, element2) - returns all the elements between the element1 and element2 including element1

##### TreeSet Class

The TreeSet class of the Java collections framework provides the functionality of a tree data structure.

In order to create a tree set, we must import the java.util.TreeSet package first.

|  |
| --- |
| TreeSet<Integer> numbers = new TreeSet<>(); |

higher(element) - Returns the lowest element among those elements that are greater than the specified element.

lower(element) - Returns the greatest element among those elements that are less than the specified element.

ceiling(element) - Returns the lowest element among those elements that are greater than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.

floor(element) - Returns the greatest element among those elements that are less than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.

headSet() method returns all the elements of a tree set before the specified element (which is passed as an argument).

tailSet() method returns all the elements of a tree set after the specified element (which is passed as a parameter) including the specified element.

subSet(e1, bv1, e2, bv2)

The subSet() method returns all the elements between e1 and e2 including e1.

### Map Interface

The Map interface of the Java collections framework provides the functionality of the map data structure.

It implements the Collection interface.

In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values. A map cannot contain duplicate keys. And, each key is associated with a single value.

We can access and modify values using the keys associated with them.

In order to use functionalities of the Map interface, we can use these classes:

* HashMap
* EnumMap
* LinkedHashMap
* WeakHashMap
* TreeMap

In Java, we must import the java.util.Map package in order to use Map.

|  |
| --- |
| // Map implementation using HashMap  Map<Key, Value> numbers = new HashMap<>(); |

The Map interface includes all the methods of the Collection interface. It is because Collection is a super interface of Map.

Besides methods available in the Collection interface, the Map interface also includes the following methods:

put(K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.

putAll() - Inserts all the entries from the specified map to this map.

putIfAbsent(K, V) - Inserts the association if the key K is not already associated with the value V.

get(K) - Returns the value associated with the specified key K. If the key is not found, it returns null.

getOrDefault(K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.

containsKey(K) - Checks if the specified key K is present in the map or not.

containsValue(V) - Checks if the specified value V is present in the map or not.

replace(K, V) - Replace the value of the key K with the new specified value V.

replace(K, oldValue, newValue) - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.

remove(K) - Removes the entry from the map represented by the key K.

remove(K, V) - Removes the entry from the map that has key K associated with value V.

keySet() - Returns a set of all the keys present in a map.

values() - Returns a set of all the values present in a map.

entrySet() - Returns a set of all the key/value mapping present in a map.

#### HashMap Class

In order to create a hash map, we must import the java.util.HashMap package first.

|  |
| --- |
| // HashMap creation with 8 capacity and 0.6 load factor  HashMap<Key, Value> numbers = new HashMap<>(8, 0.6f) |

put() - inserts the specified key/value mapping to the map

|  |
| --- |
| // Using put()  evenNumbers.put("Two", 2); |

putAll() - inserts all the entries from specified map to this map

|  |
| --- |
| // Using putAll() - numbers & even numbers are Hash Maps  numbers.putAll(evenNumbers); |

putIfAbsent() - inserts the specified key/value mapping to the map if the specified key is not present in the map

|  |
| --- |
| // Using putIfAbsent()  evenNumbers.putIfAbsent("Six", 6); |

entrySet() - returns a set of all the key/value mapping of the map

keySet() - returns a set of all the keys of the map

values() - returns a set of all the values of the map

|  |
| --- |
| // Using entrySet()  System.out.println("Key/Value mappings: " + numbers.entrySet());  // Using keySet()  System.out.println("Keys: " + numbers.keySet());  // Using values()  System.out.println("Values: " + numbers.values());    Output \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  HashMap: {One=1, Two=2, Three=3}  Key/Value mappings: [One=1, Two=2, Three=3]  Keys: [One, Two, Three]  Values: [1, 2, 3] |

get() - Returns the value associated with the specified key. Returns null if the key is not found.

getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

|  |
| --- |
| HashMap: {One=1, Two=2, Three=3}  // Using get()  int value1 = numbers.get("Three");  Returned Number: 3  // Using getOrDefault()  int value2 = numbers.getOrDefault("Five", 5);  Returned Number: 5 |

remove(key) - returns and removes the entry associated with the specified key from the map

remove(key, value) - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

|  |
| --- |
| HashMap: {One=1, Two=2, Three=3}  // remove method with single parameter  int value = numbers.remove("Two");  Removed value: 2  // remove method with two parameters  boolean result = numbers.remove("Three", 3);  Is the entry Three removed? True  Updated HashMap: {One=1} |

replace(key, value) - replaces the value associated with the specified key by a new value

replace(key, old, new) - replaces the old value with the new value only if old value is already associated with the specified key

replaceAll(function) - replaces each value of the map with the result of the specified function

|  |
| --- |
| // Using replace()  numbers.replace("Second", 22);  // Using replaceAll()  numbers.replaceAll((key, oldValue) -> oldValue + 2); |

compute() - Computes a new value using the specified function. It then associates the computed value to the specified key.

computeIfAbsent() - If the specified key is not mapped to any value, the method will compute a new value using the specified function. It then associates the new value with the key.

computeIfPresent() - If the specified key is already mapped to any value, this method will compute a new value using the specified function. It then associates the new value with the key.

|  |
| --- |
| // Using compute()  numbers.compute("First", (key, oldValue) -> oldValue + 2);    // Using computeIfAbsent()  numbers.computeIfAbsent("Three", key -> 5);    // Using computeIfPresent()  numbers.computeIfPresent("Second", (key, oldValue) -> oldValue \* 2); |

clear()-Removes all the entries from the map

containsKey()-Checks if the map contains the specified key and returns a boolean value

containsValue()-Checks if the map contains the specified value and returns a boolean value

size()-Returns the size of the map

isEmpty()-Checks if the map is empty and returns a boolean value

In a HashMap, we can

* iterate through its keys
* iterate through its values
* iterate through its keys/values

1. Using the forEach loop

|  |
| --- |
| import java.util.Map.Entry;  class Main {  public static void main(String[] args) {  // Creating a HashMap  HashMap<String, Integer> numbers = new HashMap<>();  numbers.put("One", 1);  numbers.put("Two", 2);  numbers.put("Three", 3);  System.out.println("HashMap: " + numbers);  // Accessing the key/value pair  System.out.print("Entries: ");  for(Entry<String, Integer> entry: numbers.entrySet()) {  System.out.print(entry);  System.out.print(", ");  }  // Accessing the key  System.out.print("\nKeys: ");  for(String key: numbers.keySet()) {  System.out.print(key);  System.out.print(", ");  }  // Accessing the value  System.out.print("\nValues: ");  for(Integer value: numbers.values()) {  System.out.print(value);  System.out.print(", ");  }  }  } |

1. Using iterator() Method

|  |
| --- |
| import java.util.HashMap;  import java.util.Iterator;  import java.util.Map.Entry;  class Main {  public static void main(String[] args) {  // Creating a HashMap  HashMap<String, Integer> numbers = new HashMap<>();  numbers.put("One", 1);  numbers.put("Two", 2);  numbers.put("Three", 3);  System.out.println("HashMap: " + numbers);  // Creating an object of Iterator  Iterator<Entry<String, Integer>> iterate1 = numbers.entrySet().iterator();  // Accessing the Key/Value pair  System.out.print("Entries: ");  while(iterate1.hasNext()) {  System.out.print(iterate1.next());  System.out.print(", ");  }  // Accessing the key  Iterator<String> iterate2 = numbers.keySet().iterator();  System.out.print("\nKeys: ");  while(iterate2.hasNext()) {  System.out.print(iterate2.next());  System.out.print(", ");  }  // Accessing the value  Iterator<Integer> iterate3 = numbers.values().iterator();  System.out.print("\nValues: ");  while(iterate3.hasNext()) {  System.out.print(iterate3.next());  System.out.print(", ");  }  }  } |

# Data Structures

A data structure is a particular way of organizing data in a computer so that it can be used effectively.

## linear data structures

The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular linear data structures.

1. Array

2. Linked List

3. Stack

4. Queue

### Array

Array is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

Let size of array be n.

Let size of array be n.

Accessing Time: O(1) [This is possible because elements

are stored at contiguous locations]

Search Time: O(n) for Sequential Search:

O(log n) for Binary Search [If Array is sorted]

Insertion Time: O(n) [The worst case occurs when insertion

happens at the Beginning of an array and

requires shifting all of the elements]

Deletion Time: O(n) [The worst case occurs when deletion

happens at the Beginning of an array and

requires shifting all of the elements]

### linked list

A **linked list** is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

Types of Linked List :

1. Singly Linked List : In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL

2. Doubly Linked List : In this type of Linked list, there are two references associated with each node, One of the reference points to the next node and one to the previous node. Advantage of this data structure is that we can traverse in both the directions and for deletion we don’t need to have explicit access to previous node. Eg. NULL<-1<->2<->3->NULL

3. Circular Linked List : Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Advantage of this data structure is that any node can be made as starting node. This is useful in implementation of circular queue in linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Accessing time of an element : O(n)

Search time of an element : O(n)

Insertion of an Element : O(1) [If we are at the position

where we have to insert

an element]

Deletion of an Element : O(1) [If we know address of node

previous the node to be

deleted]

*\*\*Java supports double linked list*

### Stack:

A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

Push: Add an element to the top of a stack

Pop: Remove an element from the top of a stack

IsEmpty: Check if the stack is empty

IsFull: Check if the stack is full

Peek: Get the value of the top element without removing it

|  |
| --- |
| // Stack implementation in Java  class Stack {  private int arr[];  private int top;  private int capacity;  // Creating a stack  Stack(int size) {  arr = new int[size];  capacity = size;  top = -1;  }  // Add elements into stack  public void push(int x) {  if (isFull()) {  System.out.println("OverFlow\nProgram Terminated\n");  System.exit(1);  }  System.out.println("Inserting " + x);  arr[++top] = x;  }  // Remove element from stack  public int pop() {  if (isEmpty()) {  System.out.println("STACK EMPTY");  System.exit(1);  }  return arr[top--];  }  // Utility function to return the size of the stack  public int size() {  return top + 1;  }  // Check if the stack is empty  public Boolean isEmpty() {  return top == -1;  }  // Check if the stack is full  public Boolean isFull() {  return top == capacity - 1;  }  public void printStack() {  for (int i = 0; i <= top; i++) {  System.out.println(arr[i]);  }  }  public static void main(String[] args) {  Stack stack = new Stack(5);  stack.push(1);  stack.push(2);  stack.push(3);  stack.push(4);  stack.pop();  System.out.println("\nAfter popping out");  stack.printStack();  }  } |

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

Insertion and Deletion are allowed on one end.

### Queue

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection.(The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

## Hierarchical data structures

**1. Binary Tree**

**2. Binary Search Tree**

**3. Binary Heap**

**4. Hashing**

### Binary Tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

**Binary Tree Representation:** A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.  
1. Data  
2. Pointer to left child  
3. Pointer to right child

A Binary Tree can be traversed in two ways:

Depth First Traversal: Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)  
Breadth First Traversal: Level Order Traversal

The maximum number of nodes at level ‘l’ = 2l-1.

Maximum number of nodes = 2h + 1 – 1.

Here h is height of a tree. Height is considered

as the maximum number of edges on a path from root to leaf.

Minimum possible height =  ceil(Log2(n+1)) - 1

In Binary tree, number of leaf nodes is always one

more than nodes with two children.

Time Complexity of Tree Traversal: O(n)

They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing heirarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

**Binary Search Tree**  
In Binary Search Tree is a Binary Tree with following additional properties:  
1. The left subtree of a node contains only nodes with keys less than the node’s key.  
2. The right subtree of a node contains only nodes with keys greater than the node’s key.  
3. The left and right subtree each must also be a binary search tree.

Time Complexities:

Search : O(h)

Insertion : O(h)

Deletion : O(h)

Extra Space : O(n) for pointers

**h:** Height of BST

**n:** Number of nodes in BST

If Binary Search Tree is Height Balanced,

then h = O(Log n)

Self-Balancing BSTs such as AVL Tree, Red-Black

Tree and Splay Tree make sure that height of BST

remains O(Log n)

**Binary Heap**

A Binary Heap is a Binary Tree with following properties.  
1) It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.  
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

Get Minimum in Min Heap: O(1) [Or Get Max in Max Heap]

Extract Minimum Min Heap: O(Log n) [Or Extract Max in Max Heap]

Decrease Key in Min Heap: O(Log n) [Or Decrease Key in Max Heap]

Insert: O(Log n)

Delete: O(Log n)

# Algorithms