[**Current**](#_Mallikharjuna_Sir_Class)  
Java Programming

introduction  
What is Java((old=**oak**)? **1995**->James Gosling(team **Green Team**.)1.4,5,18

**printf**=print as formatted

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language. developed by Sun Microsystems (Formarly Oracle)  
**few implementation dependencies**

Real time applications:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

### Types of Java Applications:

**Standalone application:** Desktop application/ traditional software’s that we generally install : Media Player, Antivirus(**AWT,SWING**)

**Web Applications:** Server Side for dynamic page [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial),

**Enterpraise Application:** An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

4) **Mobile Application**: An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

**There are 4 platforms or editions of Java:**

1) **Java SE** (Java Standard Edition) Standalone Applic.-> nos client server architecture

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](https://www.javatpoint.com/java-string), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2**) Java EE (Java Enterprise Edition)** Web Appl.-> Client Server Appl.

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet(Server works), JSP(view part/like front end), Web Services, EJB, [JPA](https://www.javatpoint.com/jpa-tutorial), etc.

*JDBC->* Java Database Application[ava Database Connectivity] (Middle man between backend and Database management)

Hibernate: drawbacks in jdbc ,tool for communication

Spring and Spring boot: 80 % predefined code

**3) Java ME (Java Micro Edition)**

It is a micro platform that is dedicated to **mobile applications**.

We divide each service separately and merges into one (like different cart,payment,search -> all these comes to one place)

* Before micro sevices monolithic services

**4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

* Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic

## Features of java: sppsraihmddo **amphipods1 srd**

[Simple](https://www.javatpoint.com/features-of-java#Simple): easy to learn, and its syntax( based on C++),Automatic Garbage Collection [unreferenced objects]

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

[Portable](https://www.javatpoint.com/features-of-java#Portable)Java bytecode, [Platform independent](https://www.javatpoint.com/features-of-java#Platform-independent)[ software-based platform.],byte code **wora**

[Secured](https://www.javatpoint.com/features-of-java#Secured)(byte code verifier), No explicit pointer,run inside a VM sandbox, Classloader ((from JRE) load classes dynamically, **Security Manager :** Access management for each class on res,(explicitly: SSL, JAAS, Cryptography)

Classs files are byte format-> intermediate form

, memory management is primarily handled by the **Garbage Collector (GC)**

[Robust](https://www.javatpoint.com/features-of-java#Robust)(means Strong): strong in  memory management,  lack of pointers , automatic garbage collection , exception handling and the type checking

[Architecture neutral](https://www.javatpoint.com/features-of-java#Architecture-neutral) no dependent features (size of primitive types is fixed),

[Interpreted](https://www.javatpoint.com/features-of-java#Interpreted),

[High Performance](https://www.javatpoint.com/features-of-java#High-Performance) slow than compiled language, faster than interpreted lang(JIT Compiler-increases execution/pauses JVM where program is slow)

[Multithreaded](https://www.javatpoint.com/features-of-java#Multithreaded)-s Single task into multi part

[Distributed](https://www.javatpoint.com/features-of-java#Distributed): JDK can be downloaded from the official Oracle website, as well as from other vendors , Java applications can now be distributed as custom runtime images that include only the necessary modules, reducing the size and complexity of the application. Java applications are often distributed as JAR (Java Archive)files, which are packages containing compiled Java classes, libraries, and other resources. Containers make it easy to distribute Java applications across different environments with consistent behavior.

[Dynamic](https://www.javatpoint.com/features-of-java#Dynamic): Java supports dynamic compilation and automatic memory management (garbage collection).

.class file

Java is partial OOPS:

Primitive Datatypes

S**tatic Methods and Variables-** accessed without creating an instance / everything should be accessed through objects

**Variable**

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

**Data types** specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

|  |  |  |
| --- | --- | --- |
| **Comparison Index** | C++ | Java |
| **Platform-independent** | C++ is platform-dependent. | Java is platform-independent. |
| **Mainly used for** | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications. |
| **Design Goal** | C++ was designed for systems and applications programming. It was an extension of the [C programming language](https://www.javatpoint.com/c-programming-language-tutorial). | Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience. |
| **Goto** | C++ supports the [goto](https://www.javatpoint.com/cpp-goto-statement) statement. | Java doesn't support the goto statement. |
| **Multiple inheritance** | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by using [interfaces in java](https://www.javatpoint.com/interface-in-java). |
| **Operator Overloading** | C++ supports [operator overloading](https://www.javatpoint.com/cpp-overloading). | Java doesn't support operator overloading. |
| **Pointers** | C++ supports [pointers](https://www.javatpoint.com/cpp-pointers). You can write a pointer program in C++. | Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java. |
| **Compiler and Interpreter** | C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent. | Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent. |
| **Call by Value and Call by reference** | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |
| **Structure and Union** | C++ supports structures and unions. | Java doesn't support structures and unions. |
| **Thread Support** | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in [thread](https://www.javatpoint.com/multithreading-in-java) support. |
| **Documentation comment** | C++ doesn't support documentation comments. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |
| **Virtual Keyword** | C++ supports virtual keyword so that we can decide whether or not to override a function. | Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default. |
| **unsigned right shift >>>** | C++ doesn't support >>> operator. | Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator. |
| **Inheritance Tree** | C++ always creates a new inheritance tree. | Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the [inheritance](https://www.javatpoint.com/inheritance-in-java) tree in java. |
| **Hardware** | C++ is nearer to hardware. | Java is not so interactive with hardware. |
| **Object-oriented** | C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible. | Java is also an [object-oriented](https://www.javatpoint.com/java-oops-concepts) language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object. |

## Internal Details of Hello Java Program:

**Compilation Process:**

**after javac is the jar tool. The jar file is nothing but a full pack of Java classes.**

|  |  |
| --- | --- |
| javac | Java compiler converts source code into **Java bytecode** |
| java | **The loader of the java apps.** |
| javadoc | Documentation generator, |
| jar | Java Archiver helps manage JAR files. |
| jdb | Debugger, |

**Source Code:** The HelloWorld.java,; Java Compiler (javac)-> into bytecode, .class->**Syntax Check:-> Bytecode Generation**

**Class Loading:**

**ClassLoader**: the JVM’s class loader loads the HelloWorld class into memory. The class loader locates the HelloWorld.class file, reads it, and loads the bytecode into the JVM. 

**Bootstrap ClassLoader:** primary class loader loads **essential Java classes fromJDK**.,

**Application ClassLoader:** This **loads classes from the application’s classpath**, including your **HelloWorld class.**

**Bytecode Verification**: checks for not executing harmful codes ( stack overflows, type errors, and access rights.)

**Execution of the main Method:**

* **JVM:** looks for main method( entry point ).
* **Method Area:** The JVM allocates memory for the HelloWorld class in the Method Area, a part of the JVM’s memory that stores class data, including the main method. static variables.
* **Heap and Stack Memory:** Any objects created within the main method (like args) are allocated on the heap, while primitive data types and references are stored on the stack.

Heap->a shared resource.

Stack->stores methods calls. All local variables stored in their frame, After a thread terminates, its run-time stack will be destroyed by JVM,not a shared resource



**Runtime Execution:**

* **Execution Engine:** interprets the bytecode or, using the Just-In-Time (JIT) compiler bytecode -> native machine code accrd to OS. ->executed by the CPU.
* • System.out.println("Hello, World!"); statement calls PrintStream's println method.  
  • Garbage Collection: JVM's garbage collector frees up memory used by unused objects after main method completion.

**Program Termination:**  
• JVM Shutdown: Performs cleanup operations post-main method execution. Like finalizing objects  
• Exit Code: Returns exit code, indicating successful(0) or error(other than 0))

To set path of java

**Temprory**: set path=C:\Program Files\Java\jdk1.6.0\_23\bin

* **Permenant**: Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

# Difference between JDK, JRE, and JVM

1. JVM -> platform dependent.
2. JRE -> consists of JVM and some other things. Since it include JVM, it is platform dependent.
3. JDK -> consists of JRE, compiler and some other things. Since it includes JRE which in turn includes JVM, it is platform dependent.

The java code before and after compilation is platform independant. You can compile on windows and run the byte code on unix using Unix's jvm.

Java Development Kit (JDK): **z**

* **Purpose:** Provides tools for developing Java applications.
* **It consists mainly:**Standard Edition Java Platform,Enterprise Edition Java Platform,Micro Edition Java Platform

contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

* **Components:**
  + **JRE:** Includes the Java Runtime Environment.
  + **Java Compiler (javac):** Compiles Java source code into bytecode.
  + **Development Tools:** Includes tools for debugging, documentation, and other development tasks.
* **Usage:** Required for writing, compiling, and debugging Java programs.
* **Distribution:** Downloaded from official sources like Oracle, OpenJDK, etc.



## Java Runtime Environment (JRE):

* **Purpose:** Provides the environment to run Java applications.
* **Components:**
  + **JVM:** Executes Java bytecode.-> converts Java bytecode into machine-specific code
  + **Core Libraries:** Includes standard libraries required to run Java programs.
* **Usage:** Needed to run Java applications but does not include development tools.
* **Distribution:** Included in the JDK and available separately for users who only need to run Java applications.

## Java Virtual Machine (JVM)(implementation[JRE] Provider)

* **Purpose:** Executes Java bytecode on a specific platform.it has main tasks as Loads code,Verifies code, Executes code,Provides runtime environment
* **Runtime Instance** : Creates JVM instance when running Java class./when run

JVM provides definitions for the: Memory area, Class file format, Register set, Garbage-collected heap, Fatal error reporting etc.



* **Components:**
  + **Execution Engine:** Interprets or compiles bytecode into native machine code.
  + **Class Loader:** Loads Java classes into memory.
    - Bootstrap CL:1st class loader,super class of Extension classloader.,loads rt.jar(all class files are inside of Java SE)
    - Extension CL: child of bootstrap CL and paresnt of System CL ,which loads jar files in  ‘*$JAVA\_HOME/jre/lib/ext’*
    - *System/Application CL:* child CL of Extension CL,loads class files form class path(default: set to current DIR[can change with -cp -classpath switch])

1. *Example:* *Class c=ClassLoaderExample.****class****;*
2. *System.out.println(c.getClassLoader());*
3. *->sun.misc.Launcher$AppClassLoader@4e0e2f2a*
4. *System.out.println(String.****class****.getClassLoader());*
5. *->* *null*
   * **Memory Management:** Handles allocation and garbage collection.
   * **Class(Method) Area:** Class(Method) Area stores per-class structures such as the runtime **constant pool, field and method data**, the code for methods.
   * **Heap**: It is the runtime data area in which **objects** are allocated.
   * **Program** **Counter** Register:PC (program counter) register contains the **address of the Java virtual machine instruction** currently being executed.
     + A virtual processor
     + **Interpreter:** Read bytecode stream then execute the instructions.
     + **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.
     + **Finds freq.**exec. port (hotspots) instead of executing them makes optimized native machine code  native code is stored in memory and executed directly, bypassing the need for interpretation.
     + JIT is one of the components of JVM.  improves the performance of JVM.
   * **Native Method Stack:** It contains all the native methods used in the application.
   * **Stack**: stores frames. It holds **local variables** and partial results, and plays a part in method invocation and return.
     + Each thread has a private JVM stack, created at the same time as thread.
     + A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
   * **Java Native Interface[JNI]:**is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

* **Usage:** Part of both JRE and JDK; crucial for running Java programs.
* **Platform Dependency:** JVM is platform-specific, meaning different JVM implementations exist for Windows, macOS, Linux, etc.

**Summary:**

* **JDK**: Complete toolkit for Java development, including JRE and additional tools for compiling and debugging.
* **JRE**: Provides the runtime environment to execute Java applications, including JVM and core libraries, but without development tools.
* **JVM**: Core component that executes Java bytecode, responsible for platform independence and managing memory.

Byte Code consisting of binary, hexadecimal, macro instructions like (new, add, swap, etc) and it is not directly understandable by the CPU. It is designed for efficient execution by software such as a virtual machine.intermediate-level

Machine Code is considered as the low-level code.binary code

Variables in java

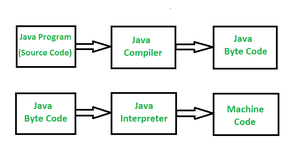
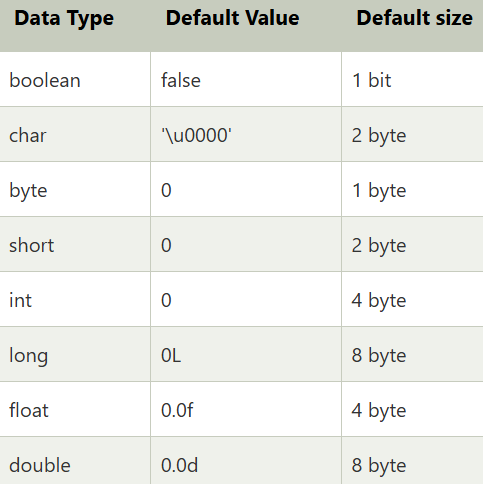
* local variable: inside the body of the method
* instance variable: inside the class but outside the body of the method
* Class/Static variables

Dat Types Primitive: boolean, char, byte, short, int, long, float and double.

Non primitive:  [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).



For signed one bit given for sign



1. **long** a = 100000L, **long** b = -200000L
2. **float** f1 = 234.5f
3. **double** d1 = 12.3
4. **char** letterA = 'A'  -uses 2 bytes because of Unicode not ASCII

|  |  |  |  |
| --- | --- | --- | --- |
| **Literal Type** | **Description** | **Example** | **Output** |
| **Integer Literal** | Represents whole numbers without any fractional part. | int a = 10; | a = 10 |
| **Floating-Point Literal** | Represents numbers with a fractional part, either in decimal or scientific notation. | float b = 3.14f; double c = 2.71828; | b = 3.14 c = 2.71828 |
| **Boolean Literal** | Represents one of the two boolean values: true or false. | boolean flag = true; | flag = true |
| **Character Literal** | Represents a single character enclosed in single quotes. | char ch = 'A'; | ch = 'A' |
| **String Literal** | Represents a sequence of characters enclosed in double quotes. | String str = "Hello"; | str = "Hello" |
| **Null Literal** | Represents the null reference, indicating that a variable does not reference any object. | String s = null; | s = null |
| **Binary Literal** | Represents integer values in binary form, starting with 0b or 0B. | int bin = 0b1010; | bin = 10 |
| **Octal Literal** | Represents integer values in octal (base 8) form, starting with 0. | int oct = 012; | oct = 10 |
| **Hexadecimal Literal** | Represents integer values in hexadecimal (base 16) form, starting with 0x or 0X. | int hex = 0xA; | hex = 10 |
| **Underscore in Numeric Literals** | Allows the use of underscores in numeric literals for better readability. | int num = 1\_000\_000; | num = 1000000 |

Top of Form

Bottom of Form

# **Unicode System**

|  |
| --- |
| Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages. |

* Unary Operator,
* Arithmetic Operator,
* Shift Operator,
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and
* Assignment Operator.

Java Operator Precedence

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | postfix | *expr*++ *expr*-- |
| prefix | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instanceof |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

U as r blta

[**Java keywords(48)**](https://www.javatpoint.com/java-keywords) are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

## Control Flow

Java provides three types of control flow statements.

**Decision Making statements:** if else statements, switch statement:if(){}else if(){}else{}

switch (expression){

    case value1:

     statement1;

     break;

**case valueN:**

**statementN;**

**break;**

    default:

     default statement;

}

(number%2==0)?"even number":"odd number";

**Loop statements:** do while loop, while loop, for loop, for-each loop

**for**(initialization, condition, increment/decrement) {

block of statements

}

**For Each:**

1. for(data\_type var : array\_name/collection\_name){    =🡺 **for**(String name:names)
2. statements
3. }

**do**

{

statements

} **while** (condition);

**Jump statements:** break statement, continue statement

### **Java Variables**

* Variables are the data containers that save the data values during Java program execution, only a name given to a memory location, operations done on the variable affect that memory location. In Java, all variables must be declared before use.
* **Local Variables** : created at the time of declaration and destroyed after exiting from the block
* **Instance Variables:** non-static variables,in a class outside of any method, constructor, or block.use access specifiers Initialization not mandatory initialize using [constructors](https://www.geeksforgeeks.org/constructors-in-java/) while creating an object. We can also use [instance blocks](https://www.geeksforgeeks.org/using-instance-blocks-in-java/) to initialize the instance variables.(stored in heap), *only created once obj created*
* **Static Variables:**using the *static keyword* within a class outside of any method, constructor, or block. only have one copy per class irrespective of no.of obj, created at the start of program execution and destroyed automatically when execution ends. Initialization of a static variable is not mandatory.if access through object->warning
* **Calling**
* Static method lo call cheyyoccu(main method)
* Can call with class name

### Scope of Variables In Java

**Member Variables (Class Level Scope)**

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class.

* We can declare class variables anywhere in class, but outside methods.
* Access specified of member variables doesn’t affect scope of them within a class.
* Member variables can be accessed outside a class with following rules

**Modifier Package Subclass World**  
  
public Yes Yes Yes  
  
protected Yes Yes No  
  
Default (no  
modifier) Yes No No  
  
private No No No

**Local Variables (Method Level Scope)**

Variables declared inside a method have method level scope and can’t be accessed outside the method.

**Loop Variables (Block Scope**    
A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

### Wrapper Classes in Java

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

**Need of Wrapper Classes (Example)**

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

|  |  |
| --- | --- |
| **Primitive Data Type** | **Wrapper Class** |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |

Autoboxing:   Character a = ch;

 Character ch = 'a';

unboxing - Character object to primitive

conversion

**char** a = ch;

int a=130;

byte b=(int)130;

do LCM for 130 and get ……00000000 10000010

pick last 8 bits ……00000000 10000010

10000010-

10000010- do 1 s complement by leaving reserved bit

1111101

11- do 2 s complement

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

1111110

 (1 x 26) + (1 x 25) + (1 x 24) + (1 x 23) + (1 x 22) + (1 x 21) + (0 x 20)  
= 64 + 32 + 16 + 8 + 4 + 2 + 0,  
= 126[1 left -]

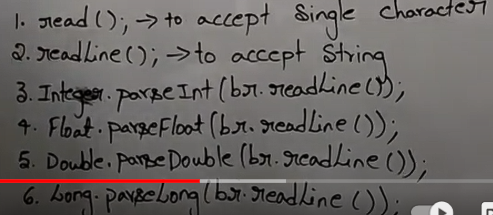
=-126

## IO in JAVA

**Methods to Take Input in Java**

There are**two ways** by which we can take Java input from the user or from a file

1. [BufferedReader Class](https://www.geeksforgeeks.org/java-io-bufferedreader-class-java/) : class that is used to read a sequence of character/s,readLine()->toread a Line
   1. *Note: BufferedReader can throw checked exceptions.*
   2. [InputStreamReader](https://www.geeksforgeeks.org/inputstreamreader-class-in-java/) class, converts the input stream of bytes into a stream of characters, allowing it to be read by BufferedReader, (expects a stream of characters.)



import java.io.\*;

class IOclassA{

public static void main(String args[]) throws IOException

{

System.out.println("welcome to the program");

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

String str=bfr.readLine();

int it=Integer.parseInt(bfr.readLine());

System.out.println(str+" "+it);

}

}

new InputStreamReader(System.in):Converts System.in (which is a byte stream) into a character stream.

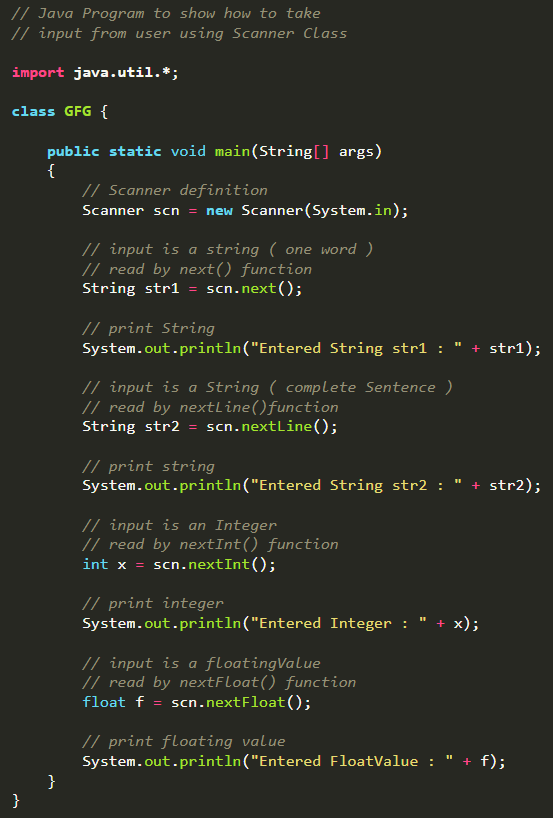
System.in reads raw bytes,InputStreamReader handles the conversion of bytes to characters based on utf-8

Bytes from System.in are converted to characters by InputStreamReader.

Characters are buffered and processed by BufferedReader.

readLine() reads one line of characters as a String.

The String can then be processed (e.g., converted to an int with Integer.parseInt()).

1. [Scanner Class](https://www.geeksforgeeks.org/scanner-class-in-java/)

import java.util.Scanner;

**Syntax:** Scanner scn = new Scanner(System.in);

Inbuilt Scanner functions are as follows:

**Integer**: [nextInt()](https://www.geeksforgeeks.org/scanner-nextint-method-in-java-with-examples/)

**Float:** [nextFloat()](https://www.geeksforgeeks.org/scanner-nextfloat-method-in-java-with-examples/)

**String :** next() and nextLine()

|  |  |
| --- | --- |
| [**nextBoolean()**](https://www.geeksforgeeks.org/scanner-nextboolean-method-in-java-with-examples/) | Used for reading Boolean value |
| [**nextByte()**](https://www.geeksforgeeks.org/scanner-nextbyte-method-in-java-with-examples/) | Used for reading Byte value |
| [**nextDouble()**](https://www.geeksforgeeks.org/scanner-nextdouble-method-in-java-with-examples/) | Used for reading Double value |
| [**nextFloat()**](https://www.geeksforgeeks.org/scanner-nextfloat-method-in-java-with-examples/) | Used for reading Float value |
| [**nextInt()**](https://www.geeksforgeeks.org/scanner-nextint-method-in-java-with-examples/) | Used for reading Int value |
| [**nextLine()**](https://www.geeksforgeeks.org/scanner-nextline-method-in-java-with-examples/) | Used for reading Line value |
| [**nextLong()**](https://www.geeksforgeeks.org/scanner-nextlong-method-in-java-with-examples/) | Used for reading Long value |
| [**nextShort()**](https://www.geeksforgeeks.org/scanner-nextshort-method-in-java-with-examples/) | Used for reading Short value |

**import** **java.io.\***;🡪 Buffer

**import java.util.Scanner;**

**while** (sc.hasNextInt()) {

Read an int value

**int** num = sc.nextInt();

            sum += num;

            count++;

        }

 Scanner sc = **new** Scanner(System.in);

**char** gender = sc.next().charAt(0);

**Constructors of BufferedReader Class**

|  |  |
| --- | --- |
| **Constructor** | **Action Performed** |
| BufferedReader(Reader in) | Creates a buffering character-input stream that uses a default-sized input buffer |
| BufferedReader(Reader in, int sz) | Creates a buffering character-input stream that uses an input buffer of the specified size. |

**Methods of BufferedReader Class**

|  |  |
| --- | --- |
| **Method Name** | **Action** |
| [close()](https://www.geeksforgeeks.org/bufferedreader-close-method-in-java-with-examples/#:~:text=The%20close()%20method%20of,associated%20with%20the%20stream%20operations.&text=Parameters%3A%20This%20method%20does%20not,does%20not%20return%20any%20value.) | Closes the stream and releases any system resources associated with it.Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect. |
| [mark()](https://www.geeksforgeeks.org/bufferedreader-mark-method-in-java-with-examples/) | Marks the present position in the stream. Subsequent calls to reset() will attempt to reposition the stream to this point. |
| [markSupported()](https://www.geeksforgeeks.org/bufferedreader-marksupported-method-in-java-with-examples/) | Tells whether this stream supports the mark() operation, which it does. |
| [read()](https://www.geeksforgeeks.org/bufferedreader-read-method-in-java-with-examples/) | Reads a single character. |
| read(char[] cbuf, int off, int len) | Reads characters into a portion of an array. This method implements the general contract of the corresponding read method of the Reader class. As an additional convenience, it attempts to read as many characters as possible by repeatedly invoking the read method of the underlying stream. |
| [readLine()](https://www.geeksforgeeks.org/bufferedreader-readline-method-in-java-with-examples/) | Reads a line of text. A line is considered to be terminated by any one of a line feed (‘  ’), a carriage return (‘\r’), or a carriage return followed immediately by a line feed. |
| [ready()](https://www.geeksforgeeks.org/bufferedreader-ready-method-in-java-with-examples/) | Tells whether this stream is ready to be read. |
| [reset()](https://www.geeksforgeeks.org/bufferedreader-reset-method-in-java-with-examples/) | Resets the stream to the most recent mark. |
| [skip(long)](https://www.geeksforgeeks.org/bufferedreader-skiplong-method-in-java-with-examples/) | Skips characters. |

String name = System.console().readLine();

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

DataInputStream reader = **new** DataInputStream(System.in);

*Reading integers*

System.out.print("Enter an integer: ");

int inputInt = Integer.parseInt(reader.readLine());

System.err.print("Enter an integer: ");

invoking [**PrintWriter**](https://www.geeksforgeeks.org/java-io-printwriter-class-java-set-1/) or the [**BufferedWriter**](https://www.geeksforgeeks.org/io-bufferedwriter-class-methods-java/)class.

## Formatted Output in Java using printf()

* For Number Formatting
* Formatting Decimal Numbers
* For Boolean Formatting
* For String Formatting
* For Char Formatting
* For Date and Time Formatting

|  |  |  |  |
| --- | --- | --- | --- |
| **Format Specifier** | **Description** | **Example** | **Output** |
| %d | Integer (decimal) | System.out.printf("%d", 123); | 123 |
| %f | Floating-point number | System.out.printf("%f", 123.45); | 123.450000 |
| %.nf | Floating-point number with n decimal places | System.out.printf("%.2f", 123.456); | 123.46 |
| %e | Scientific notation (lowercase) | System.out.printf("%e", 12345.67); | 1.234567e+04 |
| %E | Scientific notation (uppercase) | System.out.printf("%E", 12345.67); | 1.234567E+04 |
| %c | Character | System.out.printf("%c", 'A'); | A |
| %s | String | System.out.printf("%s", "Hello"); | Hello |
| %x | Integer in hexadecimal (lowercase) | System.out.printf("%x", 255); | ff |
| %X | Integer in hexadecimal (uppercase) | System.out.printf("%X", 255); | FF |
| %o | Integer in octal | System.out.printf("%o", 8); | 10 |
| %b | Boolean | System.out.printf("%b", true); | true |
| %n | Line separator | System.out.printf("Hello%nWorld"); | Hello  World |
| %% | Percentage sign | System.out.printf("%%"); | % |
| %t or %T | Date/Time (with various sub-specifiers) | System.out.printf("%tT", date); | 12:08:56 (example time) |
| %h | Hash code of the argument | System.out.printf("%h", obj); | 1f32e3b (example hash code) |

**Notes:**

* **Width and Precision**: You can specify the width and precision for formatting:
  + %10d: Width of 10 characters for an integer.
  + %10.2f: Width of 10 characters with 2 decimal places for a float.
* **Flags**:
  + %-: Left-justifies the output within the specified width.
  + +: Forces a plus or minus sign to appear before the number.
  + 0: Pads the output with leading zeros.

**Example of Complex Formatting:**

java

Copy code

System.out.printf("Item: %-10s Price: %8.2f%n", "Apple", 1.5);

* **Output**: Item: Apple Price: 1.50
* **Explanation**:
  + %-10s: Left-justified string with a width of 10.
  + %8.2f: Right-justified float with a width of 8 and 2 decimal places.
* **public** **static** **void** main(String[] args)
* {
* Date time = **new** Date();
* System.out.printf("Current Time: %tT
* ", time);

* Another Method with all of them Hour
* minutes and seconds seperated
* System.out.printf("Hours: %tH  Minutes: %tM Seconds: %tS
* ",
* time,time, time);

* Another Method to print the time
* Followed by am/pm , time in milliseconds
* nanoseconds and time-zone offset
* System.out.printf("%1$tH:%1$tM:%1$tS %1$tp %1$tL %1$tN %1$tz %n",
* time);
* }

**class** JavaFormatter2 {

main function

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

    {

**double** num = 123.4567;

prints only numeric part of a floating number

        DecimalFormat ft = **new** DecimalFormat("####");

        System.out.println("Without fraction part: num = "

                           + ft.format(num));

this will print it upto 2 decimal places

        ft = **new** DecimalFormat("#.##");

        System.out.println("Formatted to Give precision: num = "

                            + ft.format(num));

automatically appends zero to the rightmost part

of decimal instead of #,we use digit 0

        ft = **new** DecimalFormat("#.000000");

        System.out.println("appended zeroes to right: num = "

                            + ft.format(num));

automatically appends zero to the leftmost of

decimal number instead of #,we use digit 0

        ft = **new** DecimalFormat("00000.00");

        System.out.println("formatting Numeric part : num = "

                            + ft.format(num));

formatting money in dollars

**double** income = 23456.789;

        ft = **new** DecimalFormat("$###,###.##");

        System.out.println("your Formatted Dream Income : "

                             + ft.format(income));

    }

}

**Output**

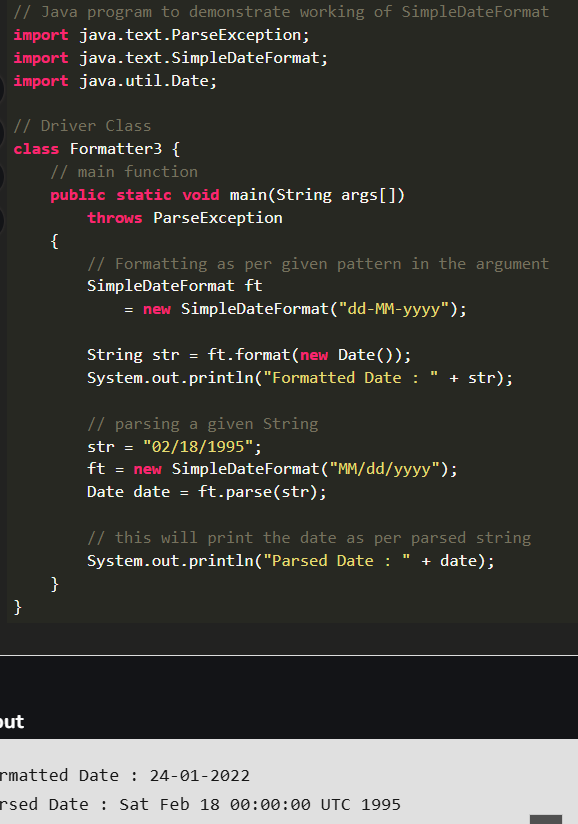
Without fraction part: num = 123

Formatted to Give precision: num = 123.46

appended zeroes to right: num = 123.456700

formatting Numeric part : num = 00123.46

your Formatted Dream Income : $23,456.79

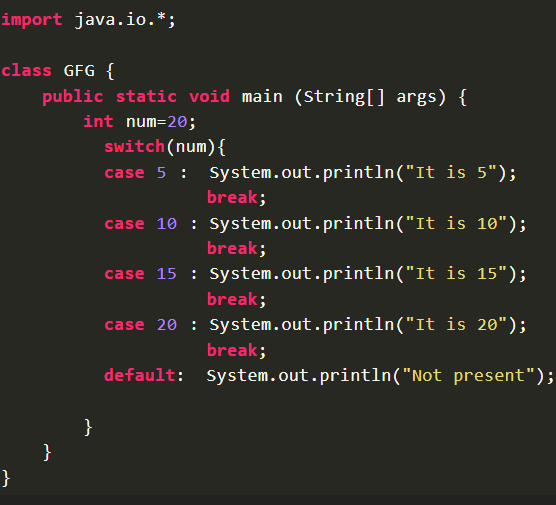


[Fast I/O in Java in Competitive Programming - GeeksforGeeks](https://www.geeksforgeeks.org/fast-io-in-java-in-competitive-programming/?ref=lbp)

At Present: [Decision Making in Java (if, if-else, switch, break, continue, jump) - GeeksforGeeks](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp)

**Java’s Selection statements:**

* [if](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#if)
* [if-else](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#if-else)
* [nested-if](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#nested-if)
* [if-else-if](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#if-else-if)
* [switch-case](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#switch-case)
* [jump](https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/?ref=lbp#jump) – break, continue, return



Expressions can be byte, short, int char, or enumeration.  
• String expressions can be used since JDK7.  
• Duplicate case values are not allowed.  
• Default statement is optional.  
• Break statement terminates statement sequence.  
• Without break keyword, statements in switch blocks fall through.  
• Omitted break keyword continues execution to next case.

[GO END](#_END)

Java supports three jump statements: **break, continue** and **return**

**Loops in Java**

[**while loop:**](https://www.geeksforgeeks.org/java-while-loop-with-examples/) A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

while (boolean condition)

{

loop statements...

}

 for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

for (initialization condition; testing condition;increment/decrement)

{

statement(s)

}**for** (**long** y = 0, z = 4; x < 10 && y < 10;

             x++, y++)

for (type var : array)

{

statements using var;

}

**int** ar[] = { 10, 50, 60, 80, 90 };

**for** (**int** element : ar)

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

    }

**Limitations of for-each loop**   
       decision-making

1. For-each loops are**not appropriate when you want to modify the array**:

for (int num : marks)

{

only changes num, not the array element

num = num\*2;

}

       2. For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop 

for (int num : numbers)

{

if (num == target)

{

return ???;

do not know the index of num

}

}

        3.  For-each **only iterates forward over the array in single steps**

cannot be converted to a for-each loop

for (int i=numbers.length-1; i>0; i--)

{

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

}

        4. For-each **cannot process two decision making statements** at once 

cannot be easily converted to a for-each loop

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

{

if (numbers[i] == arr[i])

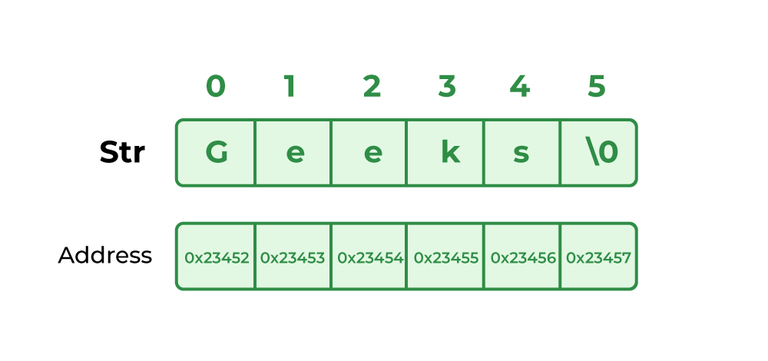
{ ...

}

}

        5. For-each also has some **performance overhead**over simple iteration:

## Strings in Java



**String str = new String("example");**

two ways to create a string in Java:

* String Literal
* Using new Keyword
* String demoString = “GeeksforGeeks”;

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

* String s = new String(“Welcome”);

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

**Interfaces and Classes in Strings in Jav**

[**CharBuffer**](https://www.geeksforgeeks.org/tag/java-charbuffer/)**:** [**String**](https://www.geeksforgeeks.org/string-class-in-java/)**:**

CharSequence Interface: used for representing the sequence of Characters in Java.

 substring, lastoccurence, first occurence, concatenate , toupper, tolower

1. String
2. StringBuffer
3. StringBuilder

String is an immutable class, cannot be changed once created ,  toupper, tolower, etc all these return a new object , thread safe.,best for multi threaded env. And shared obj of SB have over extra head

**Syntax**

String str= "geeks";  
 or  
String str= new String("geeks")

[StringBuffer](https://www.geeksforgeeks.org/stringbuffer-class-in-java/) is a peer class of **String**, ,mutable,thread safe,

**Syntax:**

StringBuffer demoString = new StringBuffer("GeeksforGeeks");

### StringBuffer Methods: Usage Guide

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Syntax** | **Example** |
| append() | Appends the specified string to the end of the StringBuffer. | sb.append(str) | java  StringBuffer sb = new StringBuffer("Hello");  sb.append(" World");  System.out.println(sb);  Output: Hello World |
| length() | Returns the current length (number of characters) in the buffer. | sb.length() | java  StringBuffer sb = new StringBuffer("Hello");  System.out.println(sb.length());  Output: 5 |
| capacity() | Returns the current capacity (allocated storage) of the buffer. | sb.capacity() | java  StringBuffer sb = new StringBuffer("Hello");  System.out.println(sb.capacity());  Output: 21 |
| charAt() | Returns the character at the specified index. | sb.charAt(index) | java  StringBuffer sb = new StringBuffer("Hello");  System.out.println(sb.charAt(1));  Output: e |
| delete() | Removes the characters in a substring of the StringBuffer. | sb.delete(startIndex, endIndex) | java  StringBuffer sb = new StringBuffer("Hello World");  sb.delete(5, 11);  System.out.println(sb);  Output: Hello |
| deleteCharAt() | Removes the character at the specified index. | sb.deleteCharAt(index) | java  StringBuffer sb = new StringBuffer("Hello");  sb.deleteCharAt(1);  System.out.println(sb);  Output: Hllo |
| ensureCapacity() | Ensures that the capacity is at least equal to the specified minimum. | sb.ensureCapacity(minCapacity) | java  StringBuffer sb = new StringBuffer();  sb.ensureCapacity(50);  System.out.println(sb.capacity());  Output: 50 |
| insert() | Inserts the specified string at the specified position. | sb.insert(offset, str) | java  StringBuffer sb = new StringBuffer("Hello");  sb.insert(5, " World");  System.out.println(sb);  Output: Hello World |
| reverse() | Reverses the characters in the StringBuffer. | sb.reverse() | java  StringBuffer sb = new StringBuffer("Hello");  sb.reverse();  System.out.println(sb);  Output: olleH |
| replace() | Replaces the characters in a substring with another string. | sb.replace(startIndex, endIndex, str) | java  StringBuffer sb = new StringBuffer("Hello World");  sb.replace(6, 11, "Java");  System.out.println(sb);  Output: Hello Java |

**Explanation:**

* **append()**: Adds a string to the end of the current StringBuffer.
* **length()**: Tells you how many characters are currently in the buffer.
* **capacity()**: Shows the current storage capacity, which may be more than the number of characters.
* **charAt()**: Retrieves a specific character by its index.
* **delete()**: Removes a sequence of characters between two indices.
* **deleteCharAt()**: Removes a single character at a specified index.
* **ensureCapacity()**: Ensures the buffer can hold at least the specified number of characters.
* **insert()**: Inserts a string at a specified index.
* **reverse()**: Reverses the entire string in the buffer.
* **replace()**: Replaces a substring within the buffer with another string.

[StringBuilder](https://www.geeksforgeeks.org/stringbuilder-class-in-java-with-examples/) in Java represents an alternative to String and StringBuffer Class, mutable, not thread safe. used only within the thread ,  no extra overhead, single threaded program used

### StringTokenizer(Depricated)

[StringTokenizer](https://www.geeksforgeeks.org/stringtokenizer-class-java-example-set-1-constructors/) class in Java is used to break a string into tokens.

import java.util.StringTokenizer;

public class Example {

public static void main(String[] args) {

String input = "John,Doe,123 Main St.,Anytown,USA";

StringTokenizer tokenizer = new StringTokenizer(input, ",");

while (tokenizer.hasMoreTokens()) {

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

}

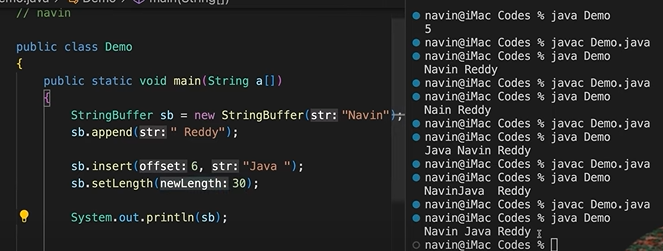
}

}

Key Methods

1. **nextToken()**: Retrieves the next token from the string.
2. **hasMoreTokens()**: Checks if there are more tokens available.
3. **countTokens()**: Returns the number of tokens remaining.
4. **nextElement()**: Similar to nextToken() but returns an Object.
5. **hasMoreElements()**: Similar to hasMoreTokens() but implements the Enumeration interface.

**String buffer gives us mutable string,it gives us size**

****

**String class in Java**

The string is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

### String Creation and Constructors in Java

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Syntax** | **Example** |
| **String Literal** | Creates a String directly as a literal in double quotes. | String s = "GeeksforGeeks"; | java  String s = "GeeksforGeeks"; |
| **Using new Keyword** | Creates a String object explicitly using the new keyword. | String s = new String("GeeksforGeeks"); | java  String s = new String("GeeksforGeeks"); |
| **String(byte[] byte\_arr)** | Constructs a new String by decoding a byte array using the platform's default charset. | String s = new String(byte\_arr); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  String s = new String(b\_arr);  Output: Geeks |
| **String(byte[] byte\_arr, Charset char\_set)** | Constructs a new String by decoding a byte array using the specified charset. | String s = new String(byte\_arr, char\_set); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  Charset cs = Charset.defaultCharset();  String s = new String(b\_arr, cs);  Output: Geeks |
| **String(byte[] byte\_arr, String char\_set\_name)** | Constructs a new String by decoding a byte array using the specified charset name. | String s = new String(byte\_arr, char\_set\_name); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  String s = new String(b\_arr, "US-ASCII");  Output: Geeks |
| **String(byte[] byte\_arr, int start\_index, int length)** | Constructs a new String from a byte array, starting at start\_index with a specified length. | String s = new String(byte\_arr, start\_index, length); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  String s = new String(b\_arr, 1, 3);  Output: eek |
| **String(byte[] byte\_arr, int start\_index, int length, Charset cs)** | Constructs a new String from a byte array, starting at start\_index, with a specified length and Charset. | String s = new String(byte\_arr, start\_index, length, cs); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  Charset cs = Charset.defaultCharset();  String s = new String(b\_arr, 1, 3, cs);  Output: eek |
| **String(byte[] byte\_arr, int start\_index, int length, String cs\_name)** | Constructs a new String from a byte array, starting at start\_index, with a specified length and charset name. | String s = new String(byte\_arr, start\_index, length, cs\_name); | java  byte[] b\_arr = {71, 101, 101, 107, 115};  String s = new String(b\_arr, 1, 4, "US-ASCII");  Output: eeks |
| **String(char[] char\_arr)** | Constructs a new String from a character array. | String s = new String(char\_arr); | java  char[] char\_arr = {'G', 'e', 'e', 'k', 's'};  String s = new String(char\_arr);  Output: Geeks |
| **String(char[] char\_arr, int start\_index, int count)** | Constructs a new String from a character array, starting at start\_index, with a specified count. | String s = new String(char\_arr, start\_index, count); | java  char[] char\_arr = {'G', 'e', 'e', 'k', 's'};  String s = new String(char\_arr, 1, 3);  Output: eek |
| **String(int[] uni\_code\_points, int offset, int count)** | Constructs a new String from an array of Unicode code points. | String s = new String(uni\_code\_points, offset, count); | java  int[] uni\_code = {71, 101, 101, 107, 115};  String s = new String(uni\_code, 1, 3);  Output: eek |
| **String(StringBuffer s\_buffer)** | Constructs a new String from a StringBuffer. | String s = new String(s\_buffer); | java  StringBuffer s\_buffer = new StringBuffer("Geeks");  String s = new String(s\_buffer);  Output: Geeks |
| **String(StringBuilder s\_builder)** | Constructs a new String from a StringBuilder. | String s = new String(s\_builder); | java  StringBuilder s\_builder = new StringBuilder("Geeks");  String s = new String(s\_builder);  Output: Geeks |

### String Methods in Java

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Syntax** | **Example** |
| **int length()** | Returns the number of characters in the string. | s.length() | java  String s = "GeeksforGeeks";  System.out.println(s.length());  Output: 13 |
| **char charAt(int i)** | Returns the character at the specified index i. | s.charAt(i) | java  String s = "GeeksforGeeks";  System.out.println(s.charAt(3));  Output: k |
| **String substring(int i)** | Returns the substring starting from index i to the end of the string. | s.substring(i) | java  String s = "GeeksforGeeks";  System.out.println(s.substring(3));  Output: ksforGeeks |
| **String substring(int i, int j)** | Returns the substring from index i to j-1. | s.substring(i, j) | java  String s = "GeeksforGeeks";  System.out.println(s.substring(2, 5));  Output: eks |
| **String concat(String str)** | Concatenates the specified string to the end of this string. | s.concat(str) | java  String s1 = "Geeks";  String s2 = "forGeeks";  System.out.println(s1.concat(s2));  Output: GeeksforGeeks |
| **int indexOf(String s)** | Returns the index of the first occurrence of the specified substring s. | s.indexOf(s) | java  String s = "Learn Share Learn";  System.out.println(s.indexOf("Share"));  Output: 6 |
| **int indexOf(String s, int i)** | Returns the index of the first occurrence of s, starting at the specified index i. | s.indexOf(s, i) | java  String s = "Learn Share Learn";  System.out.println(s.indexOf("ea", 3));  Output: 13 |
| **int lastIndexOf(String s)** | Returns the index of the last occurrence of the specified substring s. | s.lastIndexOf(s) | java  String s = "Learn Share Learn";  System.out.println(s.lastIndexOf("a"));  Output: 14 |
| **boolean equals(Object obj)** | Compares this string to the specified object. Returns true if they are equal. | s.equals(obj) | java  System.out.println("Geeks".equals("Geeks"));  Output: true  System.out.println("Geeks".equals("geeks"));  Output: false |
| **boolean equalsIgnoreCase(String anotherString)** | Compares this string to another string, ignoring case considerations. | s.equalsIgnoreCase(anotherString) | java  System.out.println("Geeks".equalsIgnoreCase("Geeks"));  Output: true  System.out.println("Geeks".equalsIgnoreCase("geeks"));  Output: true |
| **int compareTo(String anotherString)** | Compares two strings lexicographically. Returns a value less than, equal to, or greater than 0 based on comparison. | s.compareTo(anotherString) | java  String s1 = "Geeks";  String s2 = "forGeeks";  System.out.println(s1.compareTo(s2));  Output: -3 |
| **int compareToIgnoreCase(String anotherString)** | Compares two strings lexicographically, ignoring case considerations. | s.compareToIgnoreCase(anotherString) | java  String s1 = "Geeks";  String s2 = "geeks";  System.out.println(s1.compareToIgnoreCase(s2));  Output: 0 |
| **String toLowerCase()** | Converts all characters in the string to lowercase. | s.toLowerCase() | java  String s = "HeLLo";  System.out.println(s.toLowerCase());  Output: hello |
| **String toUpperCase()** | Converts all characters in the string to uppercase. | s.toUpperCase() | java  String s = "HeLLo";  System.out.println(s.toUpperCase());  Output: HELLO |
| **String trim()** | Removes leading and trailing whitespace from the string. | s.trim() | java  String s = " Learn Share Learn ";  System.out.println(s.trim());  Output: Learn Share Learn |
| **String replace(char oldChar, char newChar)** | Returns a new string by replacing all occurrences of oldChar with newChar. | s.replace(oldChar, newChar) | java  String s = "feeksforfeeks";  System.out.println(s.replace('f', 'g'));  Output: geeksforgeeks |
| **boolean contains(String str)** | Checks if the string contains the specified sequence of characters. | s.contains(str) | java  String s = "geeksforgeeks";  System.out.println(s.contains("geeks"));  Output: true |
| **char[] toCharArray()** | Converts the string to a new character array. | s.toCharArray() | java  String s = "geeksforgeeks";  System.out.println(Arrays.toString(s.toCharArray()));  Output: ['g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's'] |
| **boolean startsWith(String prefix)** | Checks if the string starts with the specified prefix. | s.startsWith(prefix) | java  String s = "geeksforgeeks";  System.out.println(s.startsWith("geeks"));  Output: true |

## Packages In Java

**Package** in [Java](https://www.geeksforgeeks.org/java/) is a mechanism to encapsulate a group of classes, sub packages and interfaces.

Preventing naming conflicts. , Making searching/locating,vProviding controlled access: Packages can be considered as data encapsulation (or data-hiding)

*college.staff.cse*, then there are three directories, *college*, *staff*and *cse*

*college* is accessible through [CLASSPATH](https://en.wikipedia.org/wiki/Classpath_(Java)), Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice.

1. **java.lang:**Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:**Contains classes for supporting input / output operations.
3. **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:**Contains classes for creating Applets.
5. **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc). 6)
6. **java.net:**Contain classes for supporting networking operations.
7. **If we have like”MyClass**

->Myclass2

->Myclass3

->Myclass3.1

->Myclass3.2

As we try to do like import MyClass.\*🡪 it imports Myclass2,3 so if we want to access 3.1 it gives error

1. package myPackage;🡪 this file should be in myPackage folder
2. public class MyClass
3. {
4. public void getNames(String s)
5. {
6. System.out.println(s);
7. }
8. }

Using Static Import

**Static import** is a feature introduced in Java programming language ( versions 5 and above ) that allows members ( fields and methods ) defined in a class as public static to be used in Java code without specifying the class in which the field is defined. Following program demonstrates static import:

Note static keyword after import.

import static java.lang.System.\*;

class StaticImportDemo {

public static void main(String args[])

{

We don't need to use 'System.out'

as imported using static.

out.println("GeeksforGeeks");

}

}

**Procedure to Generate Output:**

**1.**Compile the Welcome.java file:

**Command:** javac Welcome.java

**2.** This command creates a Welcome.class file. To place the class file in the appropriate package directory, use:

**Command:** javac -d . Welcome.java

**3.** This command will create a new folder called FirstPackage. To run the class, use:

**Command:** java FirstPackage.Welcome

The java.util package in Java contains a collection of utility classes that are essential for various programming tasks. Here are some of the most important classes you should know:

**Java Collections and Utility Classes Overview**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class/Interface** | **Purpose** | **Syntax** | **Example (Comment)** |
| **ArrayList** | Implements a resizable array. Part of the List interface, allows dynamic arrays that grow as needed. | ArrayList<Type> list = new ArrayList<>(); | java\nArrayList<String> list = new ArrayList<>();\nlist.add("item"); // Resizable array\n |
| **HashMap** | Implements the Map interface, storing key-value pairs for quick retrieval based on keys. | HashMap<KeyType, ValueType> map = new HashMap<>(); | java\nHashMap<Integer, String> map = new HashMap<>();\nmap.put(1, "value"); // Key-value pairs\n |
| **HashSet** | Implements the Set interface, representing a collection that does not allow duplicate elements. | HashSet<Type> set = new HashSet<>(); | java\nHashSet<String> set = new HashSet<>();\nset.add("item"); // Unique elements\n |
| **LinkedList** | Implements List and Deque interfaces, providing a doubly-linked list data structure. | LinkedList<Type> list = new LinkedList<>(); | java\nLinkedList<String> list = new LinkedList<>();\nlist.add("item"); // Doubly-linked list\n |
| **Iterator** | Provides methods to iterate over a collection like ArrayList, HashSet, etc. | Iterator<Type> it = collection.iterator(); | java\nIterator<String> it = list.iterator();\nwhile(it.hasNext()) { System.out.println(it.next()); }\n |
| **Collections** | Contains static methods for manipulating collections (e.g., sort(), reverse(), shuffle()). | Collections.method(collection); | java\nCollections.sort(list); // Sorts the list\n |
| **Comparator** | Defines a method for comparing two objects, used to control object ordering. | Comparator<Type> comparator = (a, b) -> a.compareTo(b); | java\nCollections.sort(list, comparator); // Custom sorting logic\n |
| **Date** | Represents a specific moment in time, with millisecond precision. | Date date = new Date(); | java\nDate date = new Date();\nSystem.out.println(date); // Current date and time\n |
| **Calendar** | Abstract class providing methods to convert between Date and fields like YEAR, MONTH, DAY, etc. | Calendar cal = Calendar.getInstance(); | java\nCalendar cal = Calendar.getInstance();\nSystem.out.println(cal.get(Calendar.YEAR)); // Year value\n |
| **Random** | Generates pseudo-random numbers of various data types. | Random random = new Random(); | java\nRandom random = new Random();\nint num = random.nextInt(); // Random number generation\n |
| **Scanner** | Parses primitive types and strings using regular expressions, often used for input from console or files. | Scanner sc = new Scanner(System.in); | java\nScanner sc = new Scanner(System.in);\nint input = sc.nextInt(); // User input\n |
| **Properties** | Subclass of Hashtable, used to maintain lists of values with both keys and values as String. | Properties prop = new Properties(); | java\nProperties prop = new Properties();\nprop.setProperty("key", "value"); // Configuration settings\n |
| **PriorityQueue** | Implements a priority queue where elements are ordered based on natural ordering or by a Comparator. | PriorityQueue<Type> pq = new PriorityQueue<>(); | java\nPriorityQueue<Integer> pq = new PriorityQueue<>();\npq.add(10); // Priority-based processing\n |
| **Stack** | Implements a last-in-first-out (LIFO) stack of objects. | Stack<Type> stack = new Stack<>(); | java\nStack<String> stack = new Stack<>();\nstack.push("item"); // LIFO processing\n |
| **TreeMap** | Implements the Map interface with keys sorted according to natural ordering or a Comparator. | TreeMap<KeyType, ValueType> map = new TreeMap<>(); | java\nTreeMap<String, Integer> map = new TreeMap<>();\nmap.put("key", 1); // Sorted keys\n |
| **TreeSet** | Implements the Set interface with elements sorted according to natural ordering or a Comparator. | TreeSet<Type> set = new TreeSet<>(); | java\nTreeSet<String> set = new TreeSet<>();\nset.add("item"); // Sorted unique elements\n |
| **LinkedHashMap** | Extends HashMap but maintains a linked list of entries, preserving the insertion order. | LinkedHashMap<KeyType, ValueType> map = new LinkedHashMap<>(); | java\nLinkedHashMap<Integer, String> map = new LinkedHashMap<>();\nmap.put(1, "value"); // Preserves insertion order\n |
| **Observable** | Class representing an object that can have one or more observers, part of the Observer design pattern. | class ObservableClass extends Observable { ... } | java\nclass ObservableClass extends Observable { ... }\n// Notify observers on state change\n |
| **UUID** | Represents an immutable universally unique identifier (UUID). | UUID uuid = UUID.randomUUID(); | java\nUUID uuid = UUID.randomUUID();\nSystem.out.println(uuid); // Unique ID generation\n |

**Important Java Classes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class/Interface** | **Purpose** | **Syntax** | **Example (Comment)** |
| **java.lang** | The java.lang package contains classes essential for most Java applications. | import java.lang.\*; | java\nimport java.lang.\*;\n// Auto-imported classes like String, Math, and Object\n |

The java.lang package is fundamental to Java programming as it contains classes that are essential for most Java applications. Here are the most important classes you should be familiar with, especially as a fresher preparing for interviews:

**1. Object**

* **Purpose**: The root class of the Java class hierarchy. Every class in Java implicitly extends Object.
* **Key Methods**:
  + equals(): Determines if two objects are equal.
  + hashCode(): Returns a hash code value for the object.
  + toString(): Returns a string representation of the object.
  + clone(): Creates and returns a copy of the object.
  + finalize(): Called by the garbage collector when there are no more references to the object.
* **Usage**: Understanding Object is crucial as many classes override its methods to provide specific behavior.

**2. String**

* **Purpose**: Represents a sequence of characters. It is immutable, meaning once created, it cannot be changed.
* **Key Methods**:
  + length(): Returns the length of the string.
  + charAt(int index): Returns the character at the specified index.
  + substring(int beginIndex, int endIndex): Returns a substring.
  + equals(): Compares two strings for equality.
  + concat(): Concatenates two strings.
* **Usage**: Strings are used frequently in Java, so understanding how they work and how to manipulate them is essential.

**3. StringBuilder and StringBuffer**

* **Purpose**: Both are used to create mutable sequences of characters, but StringBuffer is thread-safe, whereas StringBuilder is not.
* **Key Methods**:
  + append(): Adds a string or character sequence to the end.
  + insert(): Inserts a string or character at the specified index.
  + reverse(): Reverses the sequence of characters.
* **Usage**: Use these classes when you need to modify strings frequently to avoid the overhead of creating many String objects.

**4. Math**

* **Purpose**: Provides methods for performing basic numeric operations such as exponentiation, logarithms, square roots, and trigonometric functions.
* **Key Methods**:
  + abs(): Returns the absolute value of a number.
  + max(), min(): Returns the maximum or minimum of two numbers.
  + pow(): Returns the value of the first argument raised to the power of the second argument.
  + sqrt(): Returns the square root of a number.
* **Usage**: Frequently used for mathematical operations in Java applications.

**5. System**

* **Purpose**: Provides various utilities related to the environment where the application is running, such as input/output and properties.
* **Key Methods**:
  + currentTimeMillis(): Returns the current time in milliseconds.
  + exit(): Terminates the Java virtual machine.
  + gc(): Suggests that the Java garbage collector should run.
  + getProperty(): Retrieves system properties.
  + out, err, in: Standard output, error output, and input streams.
* **Usage**: Essential for interacting with the environment and handling basic I/O operations.

**6. Integer and Other Wrapper Classes (e.g., Double, Boolean)**

* **Purpose**: Wrapper classes provide a way to use primitive data types (like int, double, boolean) as objects.
* **Key Methods**:
  + parseInt(): Converts a string to an integer.
  + valueOf(): Returns an instance of the wrapper class for a given primitive value.
  + toString(): Converts the wrapped value to a string.
* **Usage**: Useful when you need to work with primitives in contexts where objects are required (e.g., in collections).

**7. Thread**

* **Purpose**: Represents a thread of execution in a program. It allows for parallel execution of code.
* **Key Methods**:
  + start(): Starts the thread.
  + run(): Contains the code that is executed when the thread starts.
  + sleep(): Causes the current thread to pause for a specified time.
  + join(): Waits for a thread to die.
  + interrupt(): Interrupts a thread.
* **Usage**: Essential for multi-threading and concurrent programming.

**8. Runnable**

* **Purpose**: An interface that should be implemented by any class whose instances are intended to be executed by a thread.
* **Key Method**:
  + run(): Contains the code that constitutes the thread's task.
* **Usage**: Often used with Thread to define the code that should run in parallel.

**9. Exception and RuntimeException**

* **Purpose**: Exception is the superclass of all exceptions that can be thrown during the normal operation of the Java Virtual Machine. RuntimeException is a subclass that represents exceptions that can occur during runtime.
* **Key Methods**:
  + getMessage(): Returns the detail message string of the throwable.
  + printStackTrace(): Prints the throwable and its backtrace to the standard error stream.
* **Usage**: Understanding exceptions and how to handle them is crucial for writing robust Java applications.

**10. Class**

* **Purpose**: Represents classes and interfaces in a running Java application. Instances of this class provide runtime access to information about classes.
* **Key Methods**:
  + getName(): Returns the name of the class.
  + getMethods(): Returns an array of Method objects representing all the public methods of the class.
  + forName(): Returns the Class object associated with the class or interface with the given string name.
* **Usage**: Used in reflection to inspect or manipulate classes, methods, and fields at runtime.

**11. Enum**

* **Purpose**: A special class that represents a group of constants (unchangeable variables).
* **Usage**: Useful when you need to define a set of named constants, like days of the week or directions (NORTH, SOUTH, EAST, WEST).

**12. Throwable**

* **Purpose**: The superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
* **Key Methods**:
  + getMessage(): Returns the detailed message string of the throwable.
  + getCause(): Returns the cause of the throwable or null if the cause is nonexistent or unknown.
* **Usage**: Fundamental for understanding how Java handles errors and exceptions.

**13. Runtime**

* **Purpose**: Allows the application to interact with the environment in which it is running.
* **Key Methods**:
  + getRuntime(): Returns the runtime object associated with the current Java application.
  + exec(): Executes the specified string command in a separate process.
  + gc(): Runs the garbage collector.
* **Usage**: Useful for system-related tasks, such as executing external programs or managing memory.

**14. ThreadLocal**

* **Purpose**: Provides thread-local variables. Each thread accessing such a variable has its own, independently initialized copy of the variable.
* **Usage**: Essential in concurrent programming where you want to avoid sharing state between threads.

**15. Throwable**

* **Purpose**: The superclass for all exceptions and errors.
* **Usage**: Used for catching and handling errors and exceptions in Java.

The java.io package provides classes for system input and output through data streams, serialization, and the file system. As a fresher, here are the key classes you should know:

### Java.io Package

**1. File**

* **Purpose**: Represents a file or directory path in the file system.
* **Key Methods**:
  + exists(): Checks if the file or directory exists.
  + createNewFile(): Creates a new, empty file if it does not exist.
  + delete(): Deletes the file or directory.
  + list(), listFiles(): Lists the contents of a directory.
  + isDirectory(), isFile(): Checks if the path represents a directory or file.
* **Usage**: Essential for file manipulation and directory management.

**2. FileReader and FileWriter**

* **Purpose**: Used for reading from and writing to files character by character.
* **Key Methods**:
  + read(): Reads a single character or an array of characters.
  + write(): Writes a single character or an array of characters.
  + close(): Closes the file stream.
* **Usage**: Useful when working with text files, where you need to read or write data in character form.

**3. BufferedReader and BufferedWriter**

* **Purpose**: Used to read and write text from an input stream or to an output stream more efficiently by buffering characters.
* **Key Methods**:
  + readLine(): Reads a line of text.
  + write(): Writes text to the output stream.
  + flush(): Flushes the output stream and forces any buffered output bytes to be written out.
* **Usage**: Preferred when working with text data, especially when reading or writing large chunks of text.

**4. InputStream and OutputStream**

* **Purpose**: Abstract classes for reading and writing binary data.
* **Subclasses**:
  + **FileInputStream**: Reads raw byte streams from files.
  + **FileOutputStream**: Writes raw byte streams to files.
* **Key Methods**:
  + read(): Reads the next byte of data from the input stream.
  + write(): Writes the specified byte to the output stream.
  + close(): Closes the stream and releases any system resources associated with it.
* **Usage**: Used for reading and writing binary data, such as images or other non-text files.

**5. ObjectInputStream and ObjectOutputStream**

* **Purpose**: Used to serialize (write) and deserialize (read) objects, allowing objects to be converted to a byte stream and vice versa.
* **Key Methods**:
  + readObject(): Reads an object from the input stream.
  + writeObject(): Writes an object to the output stream.
  + close(): Closes the stream and releases resources.
* **Usage**: Important for saving and loading the state of objects in a file or sending objects over a network.

**6. PrintStream and PrintWriter**

* **Purpose**: Provide methods to print representations of various data values conveniently.
* **Key Methods**:
  + print(), println(): Print data followed by a newline (for println).
  + flush(): Flushes the stream.
* **Usage**: Commonly used for writing formatted text data to an output stream or a file, often used in logging.

**7. FileInputStream and FileOutputStream**

* **Purpose**: Specialized classes for reading and writing raw bytes to and from a file.
* **Key Methods**:
  + read(): Reads a byte or an array of bytes from a file.
  + write(): Writes a byte or an array of bytes to a file.
  + close(): Closes the file stream.
* **Usage**: Ideal for working with binary data, such as images, audio files, or any non-text files.

**8. DataInputStream and DataOutputStream**

* **Purpose**: Allow an application to read and write primitive Java data types (like int, float, etc.) in a portable way.
* **Key Methods**:
  + readInt(), readFloat(), readUTF(): Read different types of data.
  + writeInt(), writeFloat(), writeUTF(): Write different types of data.
* **Usage**: Useful when you need to read or write binary data that includes primitive types along with strings.

**9. BufferedInputStream and BufferedOutputStream**

* **Purpose**: Provide buffering for InputStream and OutputStream to reduce the number of I/O operations by reading and writing chunks of data.
* **Key Methods**:
  + read(): Reads a byte or array of bytes.
  + write(): Writes a byte or array of bytes.
  + flush(): Flushes the output stream and forces any buffered output bytes to be written out.
* **Usage**: Enhances performance when working with large files or streams by reducing the number of reads and writes.

**10. Serializable**

* **Purpose**: An interface that enables an object to be serialized, which means converting the state of an object into a byte stream so that it can be reverted back into a copy of the object.
* **Usage**: Implement this interface when you need to persist the state of an object or transmit it over a network.

**11. PushbackInputStream**

* **Purpose**: Allows one byte of data to be returned (or "pushed back") to the stream, making it available for subsequent reads.
* **Key Methods**:
  + unread(): Pushes back the last byte read so that it can be read again.
* **Usage**: Useful in parsing scenarios where you might need to "unread" data and process it differently based on subsequent input.

**12. RandomAccessFile**

* **Purpose**: Supports both reading from and writing to random access files, which means you can jump to any location in the file and start reading or writing.
* **Key Methods**:
  + seek(): Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
  + read(), write(): Reads or writes bytes at the file-pointer's position.
* **Usage**: Often used in situations where you need to update or read specific parts of a file without loading the entire file into memory.

**13. PipedInputStream and PipedOutputStream**

* **Purpose**: Used to create a pipe that can be connected to another pipe for communication between threads.
* **Usage**: Facilitates the transfer of data between threads through a pipe, commonly used in producer-consumer scenarios.

**Syntax:**

1. /\*\*
2. \*
3. \*We can use various tags to depict the parameter
4. \*or heading or author name
5. \*We can also use HTML tags
6. \*
7. \*/

### javadoc tags

Some of the commonly used tags in documentation comments:

|  |  |  |
| --- | --- | --- |
| **Tag** | **Syntax** | **Description** |
| {@docRoot} | {@docRoot} | to depict relative path to root directory of generated document from any page. |
| @author | @author name - text | To add the author of the class. |
| @code | {@code text} | To show the text in code font without interpreting it as html markup or nested javadoc tag. |
| @version | @version version-text | To specify "Version" subheading and version-text when -version option is used. |
| @since | @since release | To add "Since" heading with since text to generated documentation. |
| @param | @param parameter-name description | To add a parameter with given name and description to 'Parameters' section. |
| @return | @return description | Required for every method that returns something (except void) |

Let's use the Javadoc tag in a Java program.

**Calculate.java**

**import** java.io.\*;

/\*\*

 \* <h2> Calculation of numbers </h2>

 \* This program implements an application

 \* to perform operation such as addition of numbers

 \* and print the result

 \* <p>

 \* <b>Note:</b> Comments make the code readable and

 \* easy to understand.

 \*

 \* @author Anurati

 \* @version 16.0

 \* @since 2021-07-06

 \*/

**public** **class** Calculate{

    /\*\*

     \* This method calculates the summation of two integers.

     \* @param input1 This is the first parameter to sum() method

     \* @param input2 This is the second parameter to the sum() method.

     \* @return int This returns the addition of input1 and input2

     \*/

**public** **int** sum(**int** input1, **int** input2){

**return** input1 + input2;

    }

    /\*\*

    \* This is the main method uses of sum() method.

    \* @param args Unused

    \* @see IOException

    \*/

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

        Calculate obj = **new** Calculate();

**int** result = obj.sum(40, 20);

        System.out.println("Addition of numbers: " + result);

    }

 }

# Mallikharjuna Sir Class

## Core Java

### Language Fundamentals:

* [Programming](#_Programming_types)-Set of Instructions->structured (Java) and Unstructured [as some drawbacks]
* History of java
* Garbage Collector: Removes the objects when not required
* Features
* Jdk,jvm,jre
* Tokens
* Data types and literals
* Variables and constants
* Simple javac ,program console IO.

Operatorers Expressions

Looping Statements

Arrays

OOPS

Exception Handling

Multithredning

Collection Framework

File IO

Serialization and DeSerialization

Java 1.8 Features

**Java Control Statements:**

### Programming types

**1. Level of Abstraction**

* **Low-Level Languages**:
  + Closer to machine code.
  + Provide minimal abstraction from a computer's instruction set architecture.
  + Examples: Assembly language, Machine code.
* **High-Level Languages**:
  + More abstract and closer to human language.
  + Easier to write, read, and maintain.
  + Examples: Python, Java, C++.

**2. Paradigms**

* **Procedural Languages**:
  + Focus on procedures or routines (functions).
  + Use a sequence of instructions to perform tasks.
  + Examples: C, Pascal.
* **Object-Oriented Languages (OOP)**:
  + Based on the concept of "objects," which contain data and methods.
  + Support inheritance, polymorphism, encapsulation, and abstraction.
  + Examples: Java, C++, Python.
* **Functional Languages**:
  + Treat computation as the evaluation of mathematical functions.
  + Avoid changing state and mutable data.
  + Examples: Haskell, Lisp, Erlang.
* **Scripting Languages**:
  + Often interpreted and used for automating tasks or gluing together different systems.
  + Examples: Python, JavaScript, Ruby.
* **Declarative Languages**:
  + Describe *what* the program should accomplish rather than *how*.
  + Examples: SQL (for database queries), HTML (for web page structure).

**3. Syntax and Semantics**

* **Syntax**:
  + The set of rules that define the combinations of symbols considered to be correctly structured programs in a language.
  + Example: Python uses indentation to define blocks of code, while C++ uses curly braces {}.
* **Semantics**:
  + The meaning of the syntactical elements and statements in the language.
  + Example: In Python, print("Hello, World!") prints text to the console, whereas in C++, std::cout << "Hello, World!"; achieves the same goal with different syntax.

**4. Execution Model**

* **Compiled Languages**:
  + Source code is translated into machine code by a compiler.
  + The resulting executable runs directly on the hardware.
  + Examples: C, C++, Go.
* **Interpreted Languages**:
  + Source code is executed directly by an interpreter line-by-line.
  + No need for a separate compilation step.
  + Examples: Python, Ruby, JavaScript.
* **Hybrid Languages**:
  + Combine compilation and interpretation.
  + Example: Java is first compiled to bytecode and then interpreted/executed by the JVM (Java Virtual Machine).

**5. Purpose**

* **General-Purpose Languages**:
  + Designed to be versatile and applicable to various types of problems.
  + Examples: Python, Java, C++.
* **Domain-Specific Languages (DSLs)**:
  + Tailored for specific application domains.
  + Examples: SQL (databases), MATLAB (numerical computation), Verilog (hardware description).

**6. Type System**

* **Static Typing**:
  + Type checking is done at compile-time.
  + Requires explicit declaration of variable types.
  + Examples: Java, C++, Rust.
* **Dynamic Typing**:
  + Type checking is done at runtime.
  + Variables can change types on the fly.
  + Examples: Python, JavaScript, Ruby.
* **Strong vs. Weak Typing**:
  + **Strongly Typed**: Enforces strict type rules; no implicit conversions.
  + **Weakly Typed**: Allows implicit conversions between types.
  + Examples: Python (strong), JavaScript (weak).

**7. Memory Management**

* **Manual Memory Management**:
  + Developers are responsible for allocating and deallocating memory.
  + Example: C, C++.
* **Automatic Memory Management (Garbage Collection)**:
  + The language runtime automatically handles memory allocation and deallocation.
  + Examples: Java, Python, C#.

**8. Concurrency Support**

* **Native Concurrency**:
  + Built-in support for multithreading and parallel processing.
  + Examples: Go (goroutines), Java (threads), Erlang (lightweight processes).
* **Libraries/Frameworks**:
  + Languages that rely on external libraries for concurrency.
  + Examples: Python with threading or asyncio libraries.

**9. Community and Ecosystem**

* **Mature Ecosystem**:
  + Established languages with extensive libraries, frameworks, and tools.
  + Examples: Python, Java, C++.
* **Emerging Languages**:
  + Newer languages with growing communities and ecosystems.
  + Examples: Rust, Kotlin.

### Variable Declaration:

a-z,A-Z,

identifiers should not start with digit

no space between identifier

max of 30 charlength

Camle Case is suggestible

### Java Basic Syntax

**Class:**The class is a blueprint (plan) of the instance of a class (object). It can be defined as a logical template that share common properties and methods.

**Object**: The object is an instance of a class. It is an entity that has behavior and state.

**Method**: The behavior of an object is the method.

**Instance variables**: Every object has its own unique set of instance variables. The state of an object is generally created by the values that are assigned to these instance variables.

### 1. Comments in Java

There are three types of comments in Java.

**i.**Single line Comment

System.out.println("This is an comment.");

**ii.**Multi-line Comment

/\*  
 System.out.println("This is the first line comment.");  
\*/

**iii.**Documentation Comment. Also called a **doc comment**.

/\*\* documentation \*/

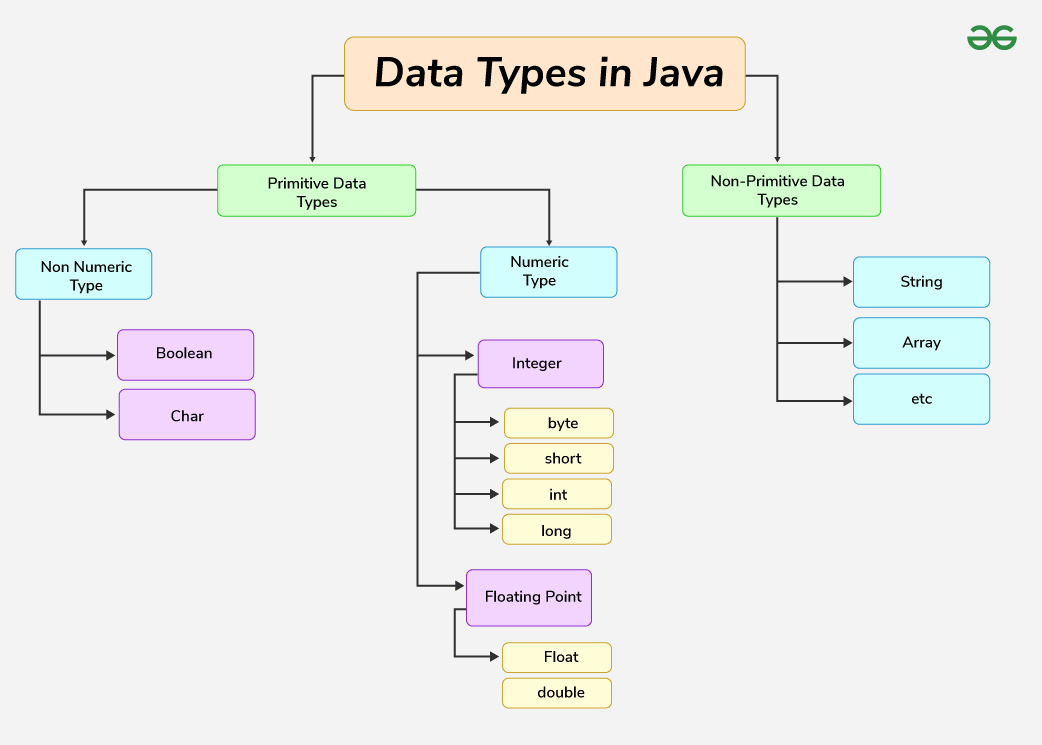
All the method names should start with a lowercase letter,Classes with upper both in camle case

**Access Modifiers:**These modifiers control the scope of class and methods.

* **Access Modifiers:**default, public, protected, private.
* **Non-access Modifiers:**final, abstract, static, transient, synchronized, volatile, native.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access Modifier | Within Class | Within Package | Outside Package by subclass only | Outside Package |
| Private | Yes | No | No | No |
| Default | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes | No |
| Public | Yes | Yes | Yes | Yes |

## DataTypes



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Description** | **Default** | **Size** | **Example Literals** | **Range of values** |
| **boolean** | true or false | false | 8 bits | true, false | true, false |
| **byte** | twos-complement integer | 0 | 8 bits | (none) | -128 to 127 |
| **char** | Unicode character | \u0000 | 16 bits | ‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’, ‘  ’, ‘β’ | characters representation of ASCII values  0 to 255 |
| **short** | twos-complement integer | 0 | 16 bits | (none) | -32,768 to 32,767 |
| **int** | twos-complement intger | 0 | 32 bits | -2,-1,0,1,2 | -2,147,483,648  to  2,147,483,647 |
| **long** | twos-complement integer | 0 | 64 bits | -2L,-1L,0L,1L,2L | -9,223,372,036,854,775,808  to  9,223,372,036,854,775,807 |
| **float** | IEEE 754 floating point | 0.0 | 32 bits | 1.23e100f , -1.23e-100f , .3f ,3.14F | upto 7 decimal digits |
| **double** | IEEE 754 floating point | 0.0 | 64 bits | 1.23456e300d , -123456e-300d , 1e1d | upto 16 decimal digits |

### Non-Primitive Data Type or Reference Data Types

<String\_Type> <string\_variable> = “<sequence\_of\_string>”;

**Example:**

Declare String without using new operator   
String s = "GeeksforGeeks";

Declare String using new operator   
String s1 = new String("GeeksforGeeks");

**Class**

A [class](https://www.geeksforgeeks.org/classes-objects-java/)is a user-defined blueprint or prototype from which objects are created.  It represents the set of properties or methods that are common to all objects of one type. in order:

1. **Modifiers**: A class can be public or has default access, **Class name**,**Superclass(if any Interfaces(if any):**A comma-separated list of interfaces implemented by the class, **Body:**The class body is surrounded by braces, { }.

**Object**

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.**Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.**Identity**: It gives a unique name to an object and enables one object to interact with other objects.

**4. Interface**

Like a class, an [interface](https://www.geeksforgeeks.org/interfaces-in-java/)can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

* Interfaces specify what a class must do and not how. It is the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* A Java library example is [Comparator Interface](https://www.geeksforgeeks.org/comparator-interface-java/). If a class implements this interface, then it can be used to sort a collection.

**5. Array**

An [Array](https://www.geeksforgeeks.org/arrays-in-java/)is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.

* In Java, all arrays are dynamically allocated. (discussed below)
* Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.
* s with [] after the data type.
* The variables in the array beginning with 0.
* Java array can also be used as a static field, a local variable, or a method parameter.
* The **size**of an array must be specified by an int value and not long or short.
* The direct superclass of an array type is Object.
* Every array type implements the interfaces [Cloneable](https://www.geeksforgeeks.org/marker-interface-java/)and [java.io.Serializable](https://www.geeksforgeeks.org/serialization-in-java/).

**Object**

In Object data type although two copies will be created they both will point to the same variable in the heap, hence changes made to any variable will reflect the change in both the variables.

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

*Here complete reference of c is copied to d*

*and both point to same memory in Heap*

int[] d = c;

d[1] = 50;

Initially

Array c: [10, 20, 30, 40]

Array d: [10, 20, 30, 40]

Modifying the value at index 1 to 50 in array d

After modification

Array c: [10, 50, 30, 40]

Array d: [10, 50, 30, 40]

|  |  |  |
| --- | --- | --- |
| **Properties** | **Primitive data types** | **Objects** |
| Origin | Pre-defined data types | User-defined data types |
| Stored structure | Stored in a stack | Reference variable is stored in stack and the original object is stored in heap |
| When copied | Two different variables is created along with different assignment(only values are same) | Two reference variable is created but both are pointing to the same object on the heap |
| When changes are made in the copied variable | Change does not reflect in the original ones. | Changes reflected in the original ones if the object is mutable. |
| Default value | Primitive datatypes do not have null as default value | The default value for the reference variable is null |
| Example | byte, short, int, long, float, double, char, boolean | array, string class, interface etc. |

### Arrays

Single Dimensional Array:

type[] arrayName;

type []arrayName;

type arrayName[];

**with new**: numbers = new int[5];

Student[] arr = new Student[5]; //Student is a user-defined class

Accessing: numbers[0] = 10;

**two dimensional array**

int[][] a;

int a[][];

int [][]a;

int a[] [];

int []a[];

numbers[0] = 20; // Changing the first element to 20 int length = numbers.length; // Getting the length of the array

all arrays are [**dynamically allocated**](https://www.geeksforgeeks.org/what-is-dynamic-memory-allocation/). may be stored in **contiguous memory** arrays are objects in Java length  with []

int length = numbers.length; // Getting the length of the array

int[][] directAssignedArray = intArray;

int[][] cloneArray = intArray.clone();

**Direct Assignment**: intArray == directAssignedArray will return true.

### Jagged Array

Jagged arrays in Java are considered **dynamic memory** rather than static memory.

**int** arr[][] = **new** **int**[2][];

        // Making the above array Jagged

        // First row has 3 columns

        arr[0] = **new** **int**[3];

        // Second row has 2 columns

        arr[1] = **new** **int**[2];

int[][] jaggedArray = new int[3][]; // Original top-level array jaggedArray[0] = new int[2]; // Initial sub-array allocation jaggedArray[1] = new int[3]; jaggedArray[2] = new int[4]; // Reallocate the first sub-array to a new array with a different size jaggedArray[0] = new int[5]; // Reallocate memory for the first sub-array

int[][] arr\_name = {

new int[] {10, 20, 30 ,40},

new int[] {50, 60, 70, 80, 90, 100},

new int[] {110, 120}

};

OR

int[][] arr\_name = {

{10, 20, 30 ,40},

{50, 60, 70, 80, 90, 100},

{110, 120}

};

// Reallocate with a larger sub-array, copying the old data

int[] newSubArray = new int[5];

System.arraycopy(jaggedArray[0], 0, newSubArray, 0, jaggedArray[0].length);

jaggedArray[0] = newSubArray; // Assign the new sub-array back to the jagged array

 // Declare the jagged array

**int**[][] jaggedArray = **new** **int**[numberOfArrays][];

        // Allocate memory to each sub-array

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

            System.out.print("Enter the size of sub-array " + (i + 1) + ": ");

**int** sizeOfSubArray = scan.nextInt();

            jaggedArray[i] = **new** **int**[sizeOfSubArray];

        }

// Initializing array

**int** count = 0;

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

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

                arr[i][j] = count++;

**Simple program with for each loop:**

**for** (**int** element : arr)

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

For-each loops**(Slower)** are**not appropriate when you want to modify the array**: For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop For-each **only iterates forward over the array in single steps**For-each **cannot process two decision making statements** at once  For-each also has some **performance overhead**over simple iteration:

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

**for** (**int** i = 0; i < 1000000; i++) {

            list.add(i);

        }

**for** (**int** i : list) {

**int** a = i;

        }

**System.arraycopy** is a built-in Java method that efficiently copies elements from one array to another.

Parameters Explained:

jaggedArray[0]: The source array from which you want to copy elements (this is {1, 2}).

0: The starting position in the source array (jaggedArray[0]) from where you begin copying (index 0).

newSubArray: The destination array where the elements are copied to.

0: The starting position in the destination array (newSubArray) where the copied data will be placed.

jaggedArray[0].length: The number of elements to copy. This ensures that all elements from the original sub-array are copied over to the new sub-array.

superclass of arrays is a class Object.

 public final field length contains the number of components

Student[] myStudents = **new** Student[]{**new** Student("Dharma"),**new** Student("sanvi"),**new** Student("Rupa"),**new** Student("Ajay")};

*// accessing the elements of the specified array*

**for**(Student m:myStudents){

System.out.println(m);

}

int arr[][]

= { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };

**public** **static** void sum(int[] arr)

*// passing of array to method*

*// returning array*

**return** **new** int[] { 1, 2, 3 };

public int[] getArray() {

int[] array = { 1, 2, 3 }; // Using a variable to store the array

return array; // Returning the array

}

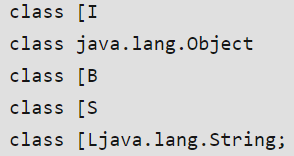
return new int[] { a, b, c }; // Returning an array with values from parameters

return new int[][] { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} }; // Returning a 2D arrayreturn new String[] { "One", "Two", "Three" }; // Returning an array of strings

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

byte byteArray[] = **new** byte[3];

short shortsArray[] = **new** short[3];

System.out.println(intArray.getClass());

System.out.println(

intArray.getClass().getSuperclass());

System.out.println(byteArray.getClass());

System.out.println(shortsArray.getClass());

System.out.println(strArray.getClass());

|  |  |
| --- | --- |
| **Array Types** | **Allowed Element Types** |
| Primitive Type Arrays | Any type which can be implicitly promoted to declared type. |
| Object Type Arrays | Either declared type objects or it’s child class objects. |
| Abstract Class Type Arrays | Its child-class objects are allowed. Objects of any class that extends the abstract class. |
| Interface Type Arrays | Its implementation class objects are allowed. |

int[] intArray = {10, 20, 30}; // Array of `int`

long[] longArray = {10, 20, 30}; // `int` values can be promoted to `long`

double[] doubleArray = {10.0f, 20}; // `float` and `int` values can be promoted to `double`

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

Animal[] animals = {new Dog(), new Cat()}; // Dog and Cat objects allowed in Animal array

abstract class Shape {}

class Circle extends Shape {}

class Square extends Shape {}

Shape[] shapes = {new Circle(), new Square()}; // Circle and Square objects allowed in Shape array

Array Construction:

char size = 10;

int[] arr = new int[size];

## Final Arrays in Java

Final variables can only be **initialized once** and final reference variables cannot be reassigned to **another object**. With final, the **object's data can be changed but not the reference.** This also applies to final arrays, where the array's data can be manipulated but the array itself cannot be reassigned.

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

**final** int[] arr2 = { 1, 2, 3, 4, 5 };

arr=arr2; //error

arr[4] = 1;

numbers[0] = 10;

System.out.println(

"Array after modifying first element: "+ Arrays.toString(numbers));

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Method** | **Description** |
| **1** | [Object get(Object array, int index)](https://www.geeksforgeeks.org/array-get-method-in-java/) | This method returns the value of the indexed component in the specified array object. |
| **2** | [boolean getBoolean(Object array, int index)](https://www.geeksforgeeks.org/array-getboolean-method-in-java/) | This method returns the value of the indexed component in the specified array object as a boolean. |
| **3** | [byte getByte(Object array, int index)](https://www.geeksforgeeks.org/array-getbyte-method-in-java/) | This method returns the value of the indexed component in the specified array object as a byte. |
| **4** | [char getChar(Object array, int index)](https://www.geeksforgeeks.org/array-getchar-method-in-java/) | This method returns the value of the indexed component in the specified array object as a char. |
| **5** | [double getDouble(Object array, int index)](https://www.geeksforgeeks.org/array-getdouble-method-in-java/) | This method returns the value of the indexed component in the specified array object as a double. |
| **6** | [float getFloat(Object array, int index)](https://www.geeksforgeeks.org/array-getfloat-method-in-java/) | This method returns the value of the indexed component in the specified array object as a float. |
| **7** | [int getInt(Object array, int index)](https://www.geeksforgeeks.org/array-getint-method-in-java/) | This method returns the value of the indexed component in the specified array object as an int. |
| **8** | **int getLength(Object array)** | This method returns the length of the specified array object as an int. |
| **9** | [long getLong(Object array, int index)](https://www.geeksforgeeks.org/array-getlong-method-in-java/) | This method returns the value of the indexed component in the specified array object as a long. |
| **10** | [short getShort(Object array, int index)](https://www.geeksforgeeks.org/array-getshort-method-in-java/) | This method returns the value of the indexed component in the specified array object as a short. |
| **11** | **Object newInstance(Class<E> componentType, int length)** | This method creates a new array with the specified component type and length. |
| **12** | **Object newInstance(Class<E> componentType, int… dimensions)** | This method creates a new array with the specified component type and dimensions. |
| **13** | [void set(Object array, int index, Object value)](https://www.geeksforgeeks.org/array-set-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified new value. |
| **14** | [void setBoolean(Object array, int index, boolean z)](https://www.geeksforgeeks.org/array-setboolean-method-in-java-with-examples/) | This method sets the value of the indexed component of the specified array object to the specified boolean value. |
| **15** | [void setByte(Object array, int index, byte b)](https://www.geeksforgeeks.org/array-setbyte-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified byte value. |
| **16** | [void setChar(Object array, int index, char c)](https://www.geeksforgeeks.org/array-setchar-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified char value. |
| **17** | [void setDouble(Object array, int index, double d)](https://www.geeksforgeeks.org/array-setdouble-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified double value. |
| **18** | [void setFloat(Object array, int index, float f)](https://www.geeksforgeeks.org/array-setfloat-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified float value. |
| **19** | [void setInt(Object array, int index, int i)](https://www.geeksforgeeks.org/array-setint-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified int value. |
| **20** | [void setLong(Object array, int index, long l)](https://www.geeksforgeeks.org/array-setlong-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified long value. |
| **21** | [void setShort(Object array, int index, short s)](https://www.geeksforgeeks.org/array-setshort-method-in-java/) | This method sets the value of the indexed component of the specified array object to the specified short value. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. No.** | **Method & Description** | **Example Usage** | **Syntax** | **Output** |
| 1 | Object get(Object array, int index)  Returns the value of the indexed component in the specified array object. | int[] arr = {1, 2, 3}; System.out.println(Array.get(arr, 1)); | Array.get(Object array, int index) | 2 |
| 2 | boolean getBoolean(Object array, int index)  Returns the value of the indexed component in the specified array as a boolean. | boolean[] boolArr = {true, false}; System.out.println(Array.getBoolean(boolArr, 0)); | Array.getBoolean(Object array, int index) | true |
| 3 | byte getByte(Object array, int index)  Returns the value of the indexed component in the specified array as a byte. | byte[] byteArr = {10, 20}; System.out.println(Array.getByte(byteArr, 1)); | Array.getByte(Object array, int index) | 20 |
| 4 | char getChar(Object array, int index)  Returns the value of the indexed component in the specified array as a char. | char[] charArr = {'a', 'b'}; System.out.println(Array.getChar(charArr, 0)); | Array.getChar(Object array, int index) | a |
| 5 | double getDouble(Object array, int index)  Returns the value of the indexed component in the specified array as a double. | double[] dblArr = {1.1, 2.2}; System.out.println(Array.getDouble(dblArr, 1)); | Array.getDouble(Object array, int index) | 2.2 |
| 6 | float getFloat(Object array, int index)  Returns the value of the indexed component in the specified array as a float. | float[] fltArr = {1.1f, 2.2f}; System.out.println(Array.getFloat(fltArr, 0)); | Array.getFloat(Object array, int index) | 1.1 |
| 7 | int getInt(Object array, int index)  Returns the value of the indexed component in the specified array as an int. | int[] intArr = {5, 10, 15}; System.out.println(Array.getInt(intArr, 2)); | Array.getInt(Object array, int index) | 15 |
| 8 | int getLength(Object array)  Returns the length of the specified array object as an int. | int[] lengthArr = {1, 2, 3, 4}; System.out.println(Array.getLength(lengthArr)); | Array.getLength(Object array) | 4 |
| 9 | long getLong(Object array, int index)  Returns the value of the indexed component in the specified array as a long. | long[] longArr = {100L, 200L}; System.out.println(Array.getLong(longArr, 1)); | Array.getLong(Object array, int index) | 200 |
| 10 | short getShort(Object array, int index)  Returns the value of the indexed component in the specified array as a short. | short[] shortArr = {1, 2}; System.out.println(Array.getShort(shortArr, 0)); | Array.getShort(Object array, int index) | 1 |
| 11 | Object newInstance(Class<E> componentType, int length)  Creates a new array with the specified component type and length. | int[] newArr = (int[]) Array.newInstance(int.class, 5); System.out.println(Array.getLength(newArr)); | Array.newInstance(Class<E> componentType, int length) | 5 |
| 12 | Object newInstance(Class<E> componentType, int... dimensions)  Creates a new array with the specified component type and dimensions. | int[][] multiArr = (int[][]) Array.newInstance(int.class, 2, 2); System.out.println(Array.getLength(multiArr)); | Array.newInstance(Class<E> componentType, int... dimensions) | 2 |
| 13 | void set(Object array, int index, Object value)  Sets the value of the indexed component of the specified array object to the specified new value. | int[] arr = {1, 2, 3}; Array.set(arr, 0, 10); System.out.println(arr[0]); | Array.set(Object array, int index, Object value) | 10 |
| 14 | void setBoolean(Object array, int index, boolean z)  Sets the value of the indexed component of the specified array object to the specified boolean value. | boolean[] boolArr = {false}; Array.setBoolean(boolArr, 0, true); System.out.println(boolArr[0]); | Array.setBoolean(Object array, int index, boolean z) | true |
| 15 | void setByte(Object array, int index, byte b)  Sets the value of the indexed component of the specified array object to the specified byte value. | byte[] byteArr = {1}; Array.setByte(byteArr, 0, (byte) 5); System.out.println(byteArr[0]); | Array.setByte(Object array, int index, byte b) | 5 |
| 16 | void setChar(Object array, int index, char c)  Sets the value of the indexed component of the specified array object to the specified char value. | char[] charArr = {'a'}; Array.setChar(charArr, 0, 'z'); System.out.println(charArr[0]); | Array.setChar(Object array, int index, char c) | z |
| 17 | void setDouble(Object array, int index, double d)  Sets the value of the indexed component of the specified array object to the specified double value. | double[] dblArr = {1.1}; Array.setDouble(dblArr, 0, 2.2); System.out.println(dblArr[0]); | Array.setDouble(Object array, int index, double d) | 2.2 |
| 18 | void setFloat(Object array, int index, float f)  Sets the value of the indexed component of the specified array object to the specified float value. | float[] fltArr = {1.1f}; Array.setFloat(fltArr, 0, 3.3f); System.out.println(fltArr[0]); | Array.setFloat(Object array, int index, float f) | 3.3 |
| 19 | void setInt(Object array, int index, int i)  Sets the value of the indexed component of the specified array object to the specified int value. | int[] intArr = {1}; Array.setInt(intArr, 0, 9); System.out.println(intArr[0]); | Array.setInt(Object array, int index, int i) | 9 |
| 20 | void setLong(Object array, int index, long l)  Sets the value of the indexed component of the specified array object to the specified long value. | long[] longArr = {100L}; Array.setLong(longArr, 0, 200L); System.out.println(longArr[0]); | Array.setLong(Object array, int index, long l) | 200 |
| 21 | void setShort(Object array, int index, short s)  Sets the value of the indexed component of the specified array object to the specified short value. | short[] shortArr = {1}; Array.setShort(shortArr, 0, (short) 7); System.out.println(shortArr[0]); | Array.setShort(Object array, int index, short s) | 7 |

length vs length ()

length: length is a final variable applicable only for arrays. by using length variable we can specify the array size.

ex: int a={10,20,30);

System.out.println(a.length);// 3.

Length()

=

length is a final method applicable only for strings. by using length() method we can specify the String length size.

ex: String s="Codegnan";

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

## OOPS in JAVA

[Message Passing in Java - GeeksforGeeks](https://www.geeksforgeeks.org/message-passing-in-java/)->clear it

\*used to solve the client problem

1.classes-> blue print

2.objects. in java everything is an object; physical entity

Created inside JVM->Heap

Reference var-> stack-> we connect it to odj body stored in heap

Classs vs object

Classs is a group of ojects,clas=virtual encapsulation/object=physical encapsulation

2.class is virtual

Object is physical

3.Class is generalization, object is specialization

4.Class is a model or blue print of objects., object is instance of object.

Message passing: Message Passing in terms of computers is communication between processes.used in oop and parllel prog

\*return type to a constructor in Java (such as void), it is no longer considered a constructor but a **regular method**.

public class Example {

// This is a regular method, NOT a constructor

public void Example() { // 'void' return type makes it a method

System.out.println("This is not a constructor!");

}

public static void main(String[] args) {

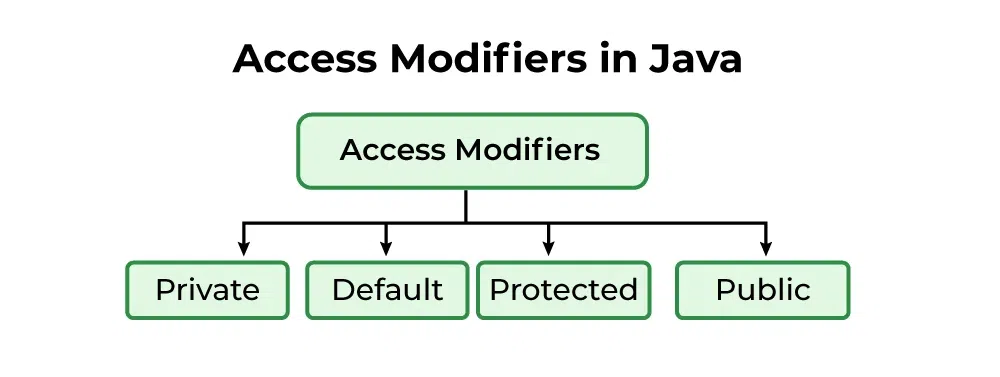
Example obj = new Example(); // Calls the default constructor

obj.Example(); // Calls the method explicitly

}

}

### Access Modifiers in Java



**Default Access Modifier:** When no access modifier is specified ,( class, method, or data member) accessible **only within the same package**.

**package** **p2**;

**import** **p1.\***; //IMPORTING

*// Java program to illustrate error while*

*// using class from different package with*

*// default modifier*

**class** **GeekNew**

{

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

{

*// Accessing class Geek from package p1*

Geek obj = **new** Geek(); //ERROR\*\*

obj.display(); //FUNCTION FROM p1

}

}

**Private ACCESS:**  keyword private, only within the class

* Any other class of the same package will not be able to access these members.
* Top-level classes or interfaces can not be declared as private because
* private means “only visible within the enclosing class”.
* protected means “only visible within the enclosing class and any subclasses”

*// Private Modifier*

**package** **p1**;

*// Class A*

**class** **A** {

**private** void display()

{

System.out.println("GeeksforGeeks");

}

}

*// Class B*

**class** **B** {

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

{

A obj = **new** A();

*// Trying to access private method*

*// of another class*

obj.display();////ERROR\*\*\*

}

}

**Protected Access:**

 keyword **protected**.

 data members declared as protected are **accessible within the same package or subclasses(which inherits) in different packages.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Access Modifier** | **Class Level** | **Package Level** | **Subclass Level (Same Package)** | **Subclass Level (Different Package)** | **World (Other Classes)** |
| **private** | **Yes** | **No** | **No** | **No** | **No** |
| **default** | **Yes** | **Yes** | **Yes** | **No** | **No** |
| **protected** | **Yes** | **Yes** | **Yes** | **Yes** | **No** |
| **public** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** |

### What is Class?

A [class](https://www.geeksforgeeks.org/classes-objects-java)is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

(or, we can say a class is a [data type](https://www.tutorialspoint.com/java/java_basic_datatypes.htm) of an object type). class is a factory (user-defined blueprint) to produce objects.

1. **Modifiers**: A class can be public or have default access (Refer to [this](https://www.geeksforgeeks.org/access-specifiers-for-classes-or-interfaces-in-java) for details).
2. **Class name:** The class name should begin with the initial letter capitalized by convention.
3. **Superclass (if any):** The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body is surrounded by braces, { }.

Properties of Java Classes

* A class does not take any byte of memory.
* A class is just like a real-world entity, but it is not a real-world entity. It's a blueprint where we specify the functionalities.
* A class contains mainly two things: Methods and Data Members.
* A class can also be a nested class.
* Classes follow all of the rules of OOPs such as inheritance, encapsulation, abstraction, etc.

### What is Object?

An **object**is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State**: It is represented by the **attributes** of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the **methods** of an object. It also reflects the response of an object to other objects.
3. **Identity**: It is a **unique name** given to an object that enables it to interact with other objects.

* **Declaration** − A variable declaration with a variable name with an object type.
* **Instantiation** − The 'new' keyword is used to create the object.
* **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

[**Method**](https://www.geeksforgeeks.org/methods-in-java)**:** A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**. In Java, every method must be part of some class, which is different from languages like [C](https://www.geeksforgeeks.org/c-programming-language/), [C++](https://www.geeksforgeeks.org/c-plus-plus/), and [Python](https://www.geeksforgeeks.org/python-programming-language/).

Dog tuffy; At this point, tuffy is just a reference variable that ***can point*** to an object of type Dog, It is uninitialized, meaning it does not yet hold a reference to any instance of the Dog class. NullPointerException

Dog tuffy = new Dog(); -> Declares and initializes tuffy with a new Dog object, meaning tuffy is now pointing to a valid instance of Dog.

1. *access\_modifier* **class** <*class\_name*>  
   {   
    data member;   
    method;   
    constructor;  
    nested class;  
    interface;  
   }

**Instance method:**

We can only access the instance method/variables only with object

### Constructor:

used to declare/create and **initialize** an object;

* There can be only one public class per source file.
* A source file can have multiple non-public classes.
* The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example − the class name is *public class Employee{}* then the source file should be as Employee.java.
* If the class is defined inside a package, then the package statement should be the first statement in the source file.
* If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
* Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

#### Copy constructor in java

Copy Constructor explanation: A Copy Constructor is a special constructor that creates a new object by copying the fields of an existing object of the same class. It's used to create a deep copy of an object, ensuring that the new object is independent of the original. This is particularly useful when you want to create a new instance with the same state as an existing object without modifying the original.

package com.codegnan.oopprogramming;

public class Person {

// instance variables

String name;

int age;

// constructor to intialize name and age

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// copy constructor

public Person(Person anotherPerson) {

this.name = anotherPerson.name;

this.age = anotherPerson.age;

}

public void displayDetails() {

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

System.out.println("Age is :" + age);

}

public static void main(String[] args) {

// creating a person object using the constructor

Person person1=new Person("Alice",32);

System.out.println("Person1 Details");

person1.displayDetails();

System.out.println();

// creating another person object using the copy constructor

Person person2=new Person(person1);

// display details of person2

System.out.println("Details of Person2(Cop")

}

}

public class Person {

private String name;

private int age;

// Regular constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Copy constructor

public Person(Person otherPerson) {

this.name = otherPerson.name;

this.age = otherPerson.age;

}

}

## Java Memory Management

 Java itself manages the memory and needs no explicit intervention of the programmer. Garbage collector itself ensures that the unused space gets cleaned and memory can be freed when not needed.

## super and this

* he super keyword refers to the superclass (parent class) of the current object. It is used to access superclass methods, constructors, and fields that have been overridden or hidden in the subclass.
* **Common Uses**:
  + Calling a superclass constructor from a subclass constructor.
  + Accessing a superclass method that has been overridden in the subclass.
  + Referencing superclass fields that are hidden by subclass fields.

**this**:

* **Purpose**: The this keyword refers to the current instance of the class. It is used to distinguish between instance variables and parameters with the same name, to invoke other constructors within the same class, and to pass the current instance to methods or constructors.
* **Common Uses**:
  + Accessing instance variables when they are shadowed by parameters or local variables.
  + Invoking another constructor in the same class (constructor chaining).
  + Passing the current object to another method or constructor.

## Exception handeling

1. ***New Thread:*** *When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.*
2. ***Runnable State:*** *A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.   
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.*
3. ***Blocked:*** *The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.*
4. ***Waiting state:*** *The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.*
5. ***Timed Waiting:*** *A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.*
6. ***Terminated State:*** *A thread terminates because of either of the following reasons:*
   * *Because it exits normally. This happens when the code of the thread has been entirely executed by the program.*
   * *Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.*

# Collections

AutoUnBoxing

automatic convertion of wrapper class object to it's corresponding primitive type.

Collection Framework

what are the diffrences between arrays and Collections

1. arrays are **fixed in nature**, arrays are able to allow elements **upto** its max size. if we add any elements over its size then JVM raise an Exception be like *java.lang.ArrayIndexOutOfBoundException*.
2. collections are *dynamically growable* in nature that isbn in the case of collections even if we add elements over its size there JVM will not raise any exception, where collections are able to allow extra elements increasing
3. collections are dynamically growable in nature that isbn in the case of collections even if we add elements over its size there JVM will not raise any exception, where collections are able to allow extra elements increasing it's size automatically.
4. 2. arrays are *able to allow homogenious elements*, that is arrays are able to allow only one type of elements, if we ass abbrev any other type of elements then the Compiler will raise an error like Incompatiable types.
5. collection objects are *able to allow hetrogenois elements*, that is collection objects are able to allow diffrent types of elemenmts.
6. 3. arrays can *hold* ***both*** *primitive types values and object values* collections can hold ***only object values.***
7. 4. arrays donot have ay predefined libraries to perform the operations like sorting, if we want to perform sorting over the elements we must write the logic for sorting explicitly.
8. collections have predefined libraries to perform the operations like sorting. no need to write logic explicitly.
9. 5. arrays are Less API dependent, so debugging and testing are simple
10. collections are More API dependent, so debugging and testing are more diffcult.
11. 6. arrays are lightweight collections are heavy weight.
12. what are the diffrences between collection vs Collections
13. collection is an interface which can be used to represent
14. group of objects in single entity. where as Collections is an utility class. present in java. util package to define several utility methods for collection Object.
15. Collection>interface
16. Collections-----> class.
17. Collection (I)
18. 1. collection is a root for all the collections.
19. 2.it has defined a number of methods to perfom operations with the elements of the collection objects.

you cannot instantiate an interface directly, interfaces like List, Set, and Map are used extensively.

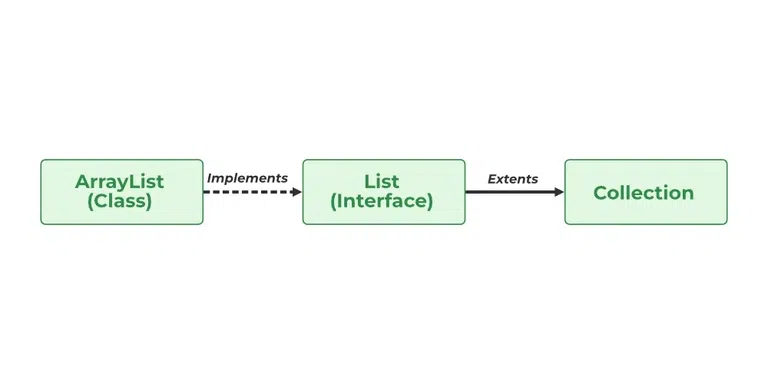
Here, ArrayList, HashSet, and HashMap are concrete classes that implement the List, Set, and Map interfaces, respectively. You’re creating an object of these concrete classes but referring to them using the interface type.

**List Interface in Java with Examples**

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

we can iterate the list in forward and backward directions.

List interface are ArrayList, LinkedList, Stack, and Vector.



**Operations in a Java List Interface**

Since List is an interface, it can be used only with a class that implements this interface. Now, let’s see how to perform a few frequently used operations on the List.

* **Operation 1:**Adding elements to List class using add() method
* al.add("Geeks");
* al.add(1, "For");// adds at pos 1
* **Operation 2:**Updating elements in List class using set() method
* al.set(1, "For"); // changes at pos1
* **Operation 3:** Searching for elements using indexOf(), lastIndexOf methods
* *// use indexOf() to find the first occurrence of an*
* *// element in the list*
* int index = numbers.indexOf(2);
* int lastIndex = numbers.lastIndexOf(2);
* **Operation 4:**Removing elements using remove() method
* *// Now remove element from the above list*
* *// present at 1st index*
* al.remove(1);
* *// Now remove the current object from the updated*
* *// List*
* al.remove("Geeks");
* **Operation 5:**Accessing Elements in List class using get() method
* returns the element at the specified index in the list.
* String first = al.get(0);
* String second = al.get(1);
* String third = al.get(2);
* **Operation 6:**Checking if an element is present in the List class using contains() method

In order to check if an element is present in the list, we can use the [**contains()**](https://www.geeksforgeeks.org/list-contains-method-in-java-with-examples/)method

**Complexity of List Interface in Java**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Space Complexity** |
| **Adding Element in List Interface** | O(1) | O(1) |
| **Remove Element from List Interface** | O(N) | O(N) |
| **Replace Element in List Interface** | O(N) | O(N) |
| **Traversing List Interface** | O(N) | O(N) |

Accessing elements”””  [for loop](https://www.geeksforgeeks.org/loop-java-important-points/) in combination with a [get() method](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) , element at a specific index and the [advanced for a loop](https://www.geeksforgeeks.org/for-each-loop-in-java/). [ **for** (String str : al) ]

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Usage** | **Example Output** |
| add(int index, element) | Adds an element at a particular index in the list. If no index, adds at the end. | List<String> list = new ArrayList<>(); list.add("A"); list.add(1, "B"); System.out.println(list); | [A, B] |
| addAll(int index, Collection) | Adds all elements from a collection starting from the specified index. | List<String> list = new ArrayList<>(Arrays.asList("A", "B")); List<String> newElements = Arrays.asList("C", "D"); list.addAll(1, newElements); | [A, C, D, B] |
| size() | Returns the size of the list. | List<String> list = Arrays.asList("A", "B", "C"); System.out.println(list.size()); | 3 |
| clear() | Removes all elements from the list but retains the reference. | List<String> list = new ArrayList<>(Arrays.asList("A", "B")); list.clear(); System.out.println(list); | [] |
| remove(int index) | Removes an element at the specified index, shifting remaining elements to the left. | List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C")); list.remove(1); System.out.println(list); | [A, C] |
| remove(element) | Removes the first occurrence of the specified element. | List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "B")); list.remove("B"); System.out.println(list); | [A, C, B] |
| get(int index) | Returns the element at the specified index. | List<String> list = Arrays.asList("A", "B", "C"); System.out.println(list.get(1)); | B |
| set(int index, element) | Replaces the element at the given index and returns the old element. | List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C")); list.set(1, "D"); System.out.println(list); | [A, D, C] |
| indexOf(element) | Returns the index of the first occurrence of the element, or -1 if not found. | List<String> list = Arrays.asList("A", "B", "C", "B"); System.out.println(list.indexOf("B")); | 1 |
| lastIndexOf(element) | Returns the index of the last occurrence of the element, or -1 if not found. | List<String> list = Arrays.asList("A", "B", "C", "B"); System.out.println(list.lastIndexOf("B")); | 3 |
| equals(element) | Compares the specified element with the list for equality. | List<String> list1 = Arrays.asList("A", "B", "C"); List<String> list2 = Arrays.asList("A", "B", "C"); System.out.println(list1.equals(list2)); | true |
| hashCode() | Returns the hash code of the list. | List<String> list = Arrays.asList("A", "B", "C"); System.out.println(list.hashCode()); | A unique integer (e.g., 3077) |
| isEmpty() | Checks if the list is empty. | List<String> list = new ArrayList<>(); System.out.println(list.isEmpty()); | true |
| contains(element) | Checks if the list contains the specified element. | List<String> list = Arrays.asList("A", "B", "C"); System.out.println(list.contains("B")); | true |
| containsAll(Collection) | Checks if the list contains all elements from the given collection. | List<String> list = Arrays.asList("A", "B", "C"); List<String> subList = Arrays.asList("A", "C"); System.out.println(list.containsAll(subList)); | true |
| sort(Comparator comp) | Sorts the elements of the list based on the given comparator. | List<String> list = new ArrayList<>(Arrays.asList("C", "A", "B")); list.sort(Comparator.naturalOrder()); System.out.println(list); | [A, B, C] |

This table provides a concise explanation, usage, and expected output for each method in the Java List interface.

We will proceed in this manner.

* ArrayList
* Vector
* Stack
* LinkedList

[Go Back](#_Mallikharjuna_Sir_Class)

### END

Wrapper class example1  
import java.util.ArrayList;

public class WrapperClassExample {

public static void main(String[] args) {

Creating an ArrayList of Integer (Wrapper class for int)

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

Autoboxing: Automatically converting int to Integer when adding to ArrayList

int num1 = 10;

numberList.add(num1);

Autoboxing occurs here

Directly adding a primitive int value (Autoboxing happens automatically)

numberList.add(20);

Autoboxing occurs here

Adding a manually created Integer object

Integer num2 = Integer.valueOf(30);

numberList.add(num2);

Displaying the elements in the ArrayList

System.out.println("ArrayList elements:");

for (Integer number : numberList) {

System.out.println(number);

Unboxing occurs here when printing

}

Using utility methods of the Wrapper class

Converting a string to an int using parseInt method

String str = "100";

int parsedNumber = Integer.parseInt(str);

Utility method

Adding the parsed number to the list (Autoboxing)

numberList.add(parsedNumber);

Unboxing: Automatically converting Integer to int

int sum = 0;

for (Integer number : numberList) {

sum += number;

Unboxing occurs here

}

System.out.println("Sum of all elements: " + sum);

Using a value retrieval method of the wrapper class

Integer wrapperNum = 40;

int intValue = wrapperNum.intValue();

Manual unboxing

System.out.println("Manual unboxed value: " + intValue);

ArrayList now contains: 10, 20, 30, 100, 40

}

}

## Java Programs

public class Main {

public static boolean isPrime(int n) {

// Handle edge cases

if (n <= 1) return false;

if (n == 2 || n == 3) return true; // 2 and 3 are prime numbers

if (n % 2 == 0 || n % 3 == 0) return false; // Exclude multiples of 2 and 3

// Start checking from 5 up to the square root of n

return isPrimeHelper(n, 5);

}

private static boolean isPrimeHelper(int n, int i) {

if (i \* i > n) return true; // If i\*i > n, n is prime

if (n % i == 0 || n % (i + 2) == 0) return false; // Check divisibility by i and i+2

return isPrimeHelper(n, i + 6); // Check the next potential divisors

}

public static void main(String[] args) {

int number = 29;

if (isPrime(number)) {

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

} else {

System.out.println(number + " is not a prime number.");

}

}

}