**Introduction of java**

**History Of Java**

Java is an [Object-Oriented programming](https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/) language developed by **James Gosling** in the early 1990s. The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc. Originally [C++](https://www.geeksforgeeks.org/c-plus-plus/) was considered to be used in the project but the idea was rejected for several reasons(For instance C++ required more memory). Gosling endeavoured to alter and expand [C++](https://www.geeksforgeeks.org/c-plus-plus/) however before long surrendered that for making another stage called **Green**. James Gosling and his team called their project “**Greentalk**” and its file extension was **. gt** and later became to known as “**OAK**”.

**Why “Oak”?**

The name **Oak** was used by **Gosling** after an **oak tree** that remained outside his office. Also, Oak is an image of solidarity and picked as a national tree of numerous nations like the U.S.A., France, Germany, Romania, etc. But they had to later rename it as “**JAVA**” as it was already a trademark by **Oak Technologies**. **“JAVA”** Gosling and his team did a brainstorm session and after the session, they came up with several names such as **JAVA, DNA, SILK, RUBY, etc.** **Java** name was decided after much discussion since it was so unique.

The name Java originates from a sort of **espresso bean**, Java. Gosling came up with this name while having a coffee near his office. Java was created on the principles like **Robust, Portable, Platform Independent, High Performance, Multithread, etc.** and was called one of the **Ten Best Products of 1995** by the **TIME MAGAZINE**. Currently, Java is used in **internet programming, mobile devices, games, e-business solutions, etc.**

The [Java language](https://www.geeksforgeeks.org/java/) has experienced a few changes since **JDK 1.0** just as various augmentations of classes and packages to the standard library. In Addition to the language changes, considerably more sensational changes have been made to the Java Class Library throughout the years, which has developed from a couple of hundred classes in JDK 1.0 to more than three thousand in J2SE 5.

Feature of java

**1. Platform Independent**

**Compiler** converts source code to [**byte code**](https://www.geeksforgeeks.org/byte-code-in-java/)and then the JVM executes the bytecode generated by the compiler. This byte code can run on any platform be it Windows, Linux, or macOS which means if we compile a program on **Windows**, then we can run it on**Linux** and vice versa. Each operating system has a different**JVM**, but the output produced by all the OS is the same after the execution of the **byte code**. That is [**why we call java a platform-independent language.**](https://www.geeksforgeeks.org/java-platform-independent/)

**2. Object-Oriented Programming**

**Java**is an [**object-oriented language**](https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/), promoting the use of**objects** and **classes**. Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

The**four main concepts of Object-Oriented programming** are:

* [***Abstraction***](https://www.geeksforgeeks.org/abstraction-in-java-2/)
* [***Encapsulation***](https://www.geeksforgeeks.org/encapsulation-in-java/)
* [***Inheritance***](https://www.geeksforgeeks.org/inheritance-in-java/)
* [***Polymorphism***](https://www.geeksforgeeks.org/polymorphism-in-java/)

**3. Simplicity**

[**Java’s syntax**](https://www.geeksforgeeks.org/java-basic-syntax/) is simple and easy to learn, especially for those familiar with **C** or **C++**. It eliminates complex features like pointers and multiple inheritances, making it easier to ***write, debug,***and ***maintain code.***

**4. Robustness**

**Java language is robust which means reliable**. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, exception handling, and memory allocation.

**5. Security**

**In java, we don’t have pointers**, so we cannot access [**out-of-bound arrays**](https://www.geeksforgeeks.org/array-index-out-of-bounds-exception-in-java/)i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so. That’s why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the **os(operating system)** environment which makes java programs more secure.

**6. Distributed**

**We can create distributed applications using the java programming language.** Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

**7. Multithreading**

[**Java supports multithreading**](https://www.geeksforgeeks.org/multithreading-in-java/), enabling the**concurrent execution** of multiple parts of a program. This feature is particularly useful for applications that require high performance, such as games and real-time simulations.

**8. Portability**

As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable. [**WORA(Write Once Run Anywhere)**](https://www.geeksforgeeks.org/why-is-java-write-once-and-run-anywhere/)makes java application to generates a [**‘.class’ file**](https://www.geeksforgeeks.org/java-class-file/) that corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

**9. High Performance**

**Java architecture** is defined in such a way that it reduces overhead during the runtime and at some times java uses[**Just In Time (JIT) compiler**](https://www.geeksforgeeks.org/just-in-time-compiler/)where the compiler compiles code on-demand basis where it only compiles those methods that are called making applications to execute faster.

JVM,JRE, and JDK

JVM

The [**JVM**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) is an integral part of the**Java platform**, responsible for executing Java bytecode. It ensures that the output of Java programs is consistent across different platforms.

The Java Virtual Machine (JVM) is designed to run the bytecode generated by the Java compiler.Each operating system has its own version of the JVM, but all JVMs follow the same rules and standards. This means Java programs can run the same way on any device with a JVM, regardless of the operating system**.** This is why Java is called a platform-independent language.

JRE

JDK includes JRE. [**JRE**](https://www.geeksforgeeks.org/jre-in-java/) installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

JDK

While we were using the term[**JDK**](https://www.geeksforgeeks.org/jdk-in-java/) when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including **compiler,** **Java Runtime Environment (JRE)**, J**ava Debuggers**, J**ava Docs**, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

Java Program Structure

A Java program structure is built around the concept of packages, which group related classes together, and each class contains methods that define specific functionalities.

 A package acts as a namespace to organize classes, while classes encapsulate data (variables) and behaviours (methods) within a program.

* **Package:**
  + A collection of related classes, interfaces, and sub-packages.
  + Used to organize code and prevent naming conflicts.
  + Declared at the beginning of a Java file using the "package" keyword.
* **Class:**
  + A blueprint for creating objects, containing data (variables) and methods (functions).
  + Declared using the "class" keyword followed by the class name.
* **Method:**
  + A block of code that performs a specific task within a class.
  + Defined with a return type, method name, parameters, and a body of code.

Data Types, Variables and Operators

Primitive Data Types in Java

**1.boolean Data Type**

The boolean data type represents a logical value that can be either true or false. Conceptually, it represents a single bit of information, but the actual size used by the virtual machine is implementation-dependent and typically at least one byte (eight bits) in practice. Values of the boolean type are not implicitly or explicitly converted to any other type using casts. However, programmers can write conversion code if needed.

**Size :** Virtual machine dependent (typically 1 byte, 8 bits)

**2. byte Data Type**

The byte data type is an 8-bit signed two’s complement integer. The byte data type is useful for saving memory in large arrays.

**Size :**1 byte (8 bits)

**3. short Data Type**

The short data type is a 16-bit signed two’s complement integer. Similar to byte, a short is used when memory savings matter, especially in large arrays where space is constrained.

**Size :** 2 bytes (16 bits)

**4. int Data Type**

It is a 32-bit signed two’s complement integer.

**Size :**4 bytes ( 32 bits )

**5. long Data Type**

The long data type is a 64-bit signed two’s complement integer. It is used when an int is not large enough to hold a value, offering a much broader range.

**Size :** 8 bytes (64 bits)

**6. float Data Type**

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

**Size :**4 bytes (32 bits)

**7. double Data Type**

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

**Size :** 8 bytes (64 bits)

**8. char Data Type**

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

**Size :**2 bytes (16 bits)u

Variable Declaration and Initialization

Variable Declaration

Java programming language requires variables to operate and handle data. Java creates several variables as per data format and data types. The variable declaration means creating a variable in a program for operating different information.

The programming language requires four basic things to declare a variable in the program.

1. Data-type
2. Variable name
3. Initial value
4. Semicolon

**Data-type:** It represent the type of value variable.

**Variable name:** The Java variable declaration requires a unique name. We prefer to declare small and understandable variable names.

**Initial value:** Java language requires the initial value of the variable. Declare variable with initial value does not necessary in the main class. We must assign the initial value in the default constructor. The "final variable" needs to declare the initial value.

**Semicolon:** The semicolon represents the end of the variable declaration statement.

Initialization

Create several variables with the different data formats. Here, we can use int, String, Boolean and other data types.

* Create variables with required data types in the default method.
* Use variable name and its value.
* Return this value in the method as per data format.

Operators

**1. Arithmetic Operators**

|  |  |
| --- | --- |
| Operator | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulo Operation (Remainder after division) |

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

**2. Assignment Operators**

Assignment operators are used in Java to assign values to variables. For example,

int age;

|  |  |  |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| = | a = b; | a = b; |
| += | a += b; | a = a + b; |
| -= | a -= b; | a = a - b; |
| \*= | a \*= b; | a = a \* b; |
| /= | a /= b; | a = a / b; |
| %= | a %= b; | a = a % b; |

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age.

**3. Relational Operators**

Relational operators are used to check the relationship between two operands. For example,

// check if a is less than b

a < b;

Here, < operator is the relational operator. It checks if a is less than b or not.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Is Equal To | 3 == 5 returns **false** |
| != | Not Equal To | 3 != 5 returns **true** |
| > | Greater Than | 3 > 5 returns **false** |
| < | Less Than | 3 < 5 returns **true** |
| >= | Greater Than or Equal To | 3 >= 5 returns **false** |
| <= | Less Than or Equal To | 3 <= 5 returns **true** |

It returns either true or false.

**4. Logical Operators**

Logical operators are used to check whether an expression is true or false. They are used in decision making.

**Example 4: Logical Operators**

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && (Logical AND) | expression1 **&&** expression2 | true only if both expression1 and expression2 are true |
| || (Logical OR) | expression1 **||** expression2 | true if either expression1 or expression2 is true |
| ! (Logical NOT) | **!**expression | true if expression is false and vice versa |

**5. Java Unary Operators**

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

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

**6. Java Bitwise Operators**

Bitwise operators in Java are used to perform operations on individual bits.

Here, ~ is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**).

The various bitwise operators present in Java are:

|  |  |
| --- | --- |
| Operator | Description |
| ~ | Bitwise Complement |
| << | Left Shift |
| >> | Right Shift |
| >>> | Unsigned Right Shift |
| & | Bitwise AND |
| ^ | Bitwise exclusive OR |

Type Conversion and Type Casting

| **S.NO** | **TYPE CASTING** | **TYPE CONVERSION** |
| --- | --- | --- |
| 1. | In type casting, a data type is converted into another data type by a programmer using casting operator. | Whereas in type conversion, a data type is converted into another data type by a compiler. |
| 2. | Type casting can be applied to **compatible data types** as well as **incompatible data types**. | Whereas type conversion can only be applied to **compatible datatypes**. |
| 3. | In type casting, casting operator is needed in order to cast a data type to another data type. | Whereas in type conversion, there is no need for a casting operator. |
| 4. | In typing casting, the destination data type may be smaller than the source data type, when converting the data type to another data type. | Whereas in type conversion, the destination data type can’t be smaller than source data type. |
| 5. | Type casting takes place during the program design by programmer. | Whereas type conversion is done at the compile time. |
| 6. | Type casting is also called narrowing conversion because in this, the destination data type may be smaller than the source data type. | Whereas type conversion is also called widening conversion because in this, the destination data type can not be smaller than the source data type. |
| 7. | Type casting is often used in coding and competitive programming works. | Whereas type conversion is less used in coding and competitive programming as it might cause incorrect answer. |
| 8. | Type casting is more efficient and reliable. | Whereas type conversion is less efficient and less reliable. |

Control Flow Statements

If-Else Statements

The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in Java.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

1.if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

**Syntax:**

1. **if**(condition) {
2. //code to be executed
3. }

2.if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

**Syntax:**

1. **if**(condition) {
2. //code if condition is true
3. } **else** {
4. //code if condition is false
5. }

3.if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

**Syntax:**

1. **if**(condition1) {
2. //code to be executed if condition1 is true
3. } **else** **if**(condition2) {
4. //code to be executed if condition2 is true
5. }
6. **else** **if**(condition3) {
7. //code to be executed if condition3 is true
8. }
9. ...
10. **else** {
11. //code to be executed if all the conditions are false
12. }

4. Nested if statement

The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

**Syntax:**

1. **if**(condition) {
2. //code to be executed
3. **if**(condition) {
4. //code to be executed
5. }
6. }

Switch Case Statements

The Java *switch statement* executes one statement from multiple conditions. It is like [if-else-if](https://www.javatpoint.com/java-if-else) ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, we can use [strings](https://www.javatpoint.com/java-string) in the switch statement.

In Java, switch statement mainly provides a more detailed alternative that avoids the usage of nested or several if-else statements when associated with an individual variable.

The syntax of the Java switch statement contains the **switch** keyword which is followed by the expression that needs to be evaluated using parentheses. The mentioned expression must definitely evaluate to a definite data type which is primitive such as int, char, or enum.

**Syntax:**

1. **switch**(expression){
2. **case** value1:
3. //code to be executed;
4. **break**;  //optional
5. **case** value2:
6. //code to be executed;
7. **break**;  //optional
8. ......

**default**:

1. code to be executed **if** all cases are not matched;
2. }

Loops

Looping in programming languages is a feature that facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition-checking time.

**there are three types of Loops which are listed below:**

* [for loop](https://www.geeksforgeeks.org/java-for-loop-with-examples/)
* [while loop](https://www.geeksforgeeks.org/java-while-loop-with-examples/)
* [do-while loop](https://www.geeksforgeeks.org/java-do-while-loop-with-examples/)

**1. for loop**

The for loop is used when we know the number of iterations (we know how many times we want to repeat a task). The for statement consumes the initialization, condition, and increment/decrement in one line thereby providing a shorter, easy-to-debug structure of looping.

**Syntax:**

for (initialization; condition; increment/decrement) {

// code to be executed

}

**2. while Loop**

A while loop is used when we want to check the condition before running the code.

**Syntax:**

*while (condition) {*

*// code to be executed*

*}*

***3. do-while Loop***

*The do-while loop in Java ensures that the code block executes at least once before the condition is checked.*

***Syntax:***

*do {*

*// code to be executed*

*} while (condition);*

Break and Continue Keywords

**Break**

The break statement in java is used to terminate from the loop immediately. When a break statement is encountered inside a loop, the loop iteration stops there, and control returns from the loop immediately to the first statement after the loop. Basically, break statements are used in situations when we are not sure about the actual number of iteration for the loop, or we want to terminate the loop based on some condition.

**Syntax :**

break;

In Java, a break statement is majorly used for:

* To exit a loop.
* Used as a “civilized” form of goto.
* Terminate a sequence in a switch statement.

**Continue:**

The continue statement in Java is used to skip the current iteration of a loop. We can use continue statement inside any types of loops such as for, while, and do-while loop. Basically continue statements are used in the situations when we want to continue the loop but do not want the remaining statement after the continue statement.

**Syntax:**

continue;

**Difference between break and continue:**

| **Break** | **Continue** |
| --- | --- |
| The break statement is used to terminate the loop immediately. | The continue statement is used to skip the current iteration of the loop. |
| break keyword is used to indicate break statements in java programming. | continue keyword is used to indicate continue statement in java programming. |
| We can use a break with the switch statement. | We can not use a continue with the switch statement. |
| The break statement terminates the whole loop early. | The continue statement brings the next iteration early. |
| It stops the execution of the loop. | It does not stop the execution of the loop. |

Classes and Objects

Class and Object in Java

Classes

A class in Java is a set of objects which shares common characteristics and common properties. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

**Components of Classes**

In general, class declarations can include these components, in order:

* **Modifiers**: A class can be public or has default access (Refer [this](https://www.geeksforgeeks.org/access-specifiers-for-classes-or-interfaces-in-java/)for details).
* **Class keyword:** Class keyword is used to create a class.
* **Class name**: The name should begin with an initial letter (capitalized by convention).
* **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.
* **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.
* **Body**: The class body is surrounded by braces, { }.

**Object**

An **object in Java** is a basic unit of Object-Oriented Programming and represents real-life entities. [Objects](https://www.geeksforgeeks.org/object-class-in-java/) are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods.

Constructors and Overloading

Constructors

 A Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

**Types of Constructors in Java**

* Default Constructor
* Parameterized Constructor
* Copy Constructor

1. **Default Constructor**

A constructor that has no parameters is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor.

1. **Parameterized Constructor**

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

1. **Copy Constructor**

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

Overloading

Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.

Method overloading in Java is also known as[***Compile-time Polymorphism***](https://www.geeksforgeeks.org/compile-time-polymorphism-in-java)***, Static Polymorphism, or***[***Early binding***](https://www.geeksforgeeks.org/difference-between-early-and-late-binding-in-java). In Method overloading compared to the parent argument, the child argument will get the highest priority.

**Different Ways of Method Overloading in Java**

* Changing the Number of Parameters.
* Changing Data Types of the Arguments.
* Changing the Order of the Parameters of Methods

This Keyword

The this keyword refers to the current object in a method or constructor.

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

this can also be used to:

* Invoke current class constructor
* Invoke current class method
* Return the current class object
* Pass an argument in the method call
* Pass an argument in the constructor call

Methods

Defining Methods

**Java Methods** are blocks of code that perform a specific task. A method allows us to reuse code, improving both efficiency and organization. All **methods in Java** must belong to a **class**. Methods are similar to functions and expose the behavior of objects.

**Syntax of a Method**

*<access\_modifier> <return\_type> <method\_name>( list\_of\_parameters)  
{  
//body  
}*

**Types of Methods in Java**

**1. Predefined Method**

**Predefined methods**are the method that is already defined in the Java class libraries. It is also known as the standard library method or built-in method.

**2. User-defined Method**

The method written by the user or programmer is known as a **user-defined method.** These methods are modified according to the requirement.

**Method Parameters and Return Types**

**Method Parameters**

Information can be passed to methods as a parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

**Return Values**

In the [previous page](https://www.w3schools.com/java/java_methods_param.asp), we used the void keyword in all examples, which indicates that the method should not return a value.

If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method.

Method Overloading

Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.

Method overloading in Java is also known as[***Compile-time Polymorphism***](https://www.geeksforgeeks.org/compile-time-polymorphism-in-java)***, Static Polymorphism, or***[***Early binding***](https://www.geeksforgeeks.org/difference-between-early-and-late-binding-in-java). In Method overloading compared to the parent argument, the child argument will get the highest priority.

**Different Ways of Method Overloading in Java**

* Changing the Number of Parameters.
* Changing Data Types of the Arguments.
* Changing the Order of the Parameters of Methods

Static Methods and Variables

The static keyword is used to construct methods that will exist regardless of whether or not any instances of the class are generated. Any method that uses the static keyword is referred to as a static method.

**Syntax to declare the static method:**

Access\_modifier static void methodName()  
{   
 // Method body.  
}

The name of the class can be used to invoke or access static methods.

**Syntax to call a static method:**

className.methodName();

Object-Oriented Programming (OOPs) Concepts

Basics of OOP

There are four pillars been here in OOPS which are listed below.

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**Abstraction**

[**Abstraction**](https://www.geeksforgeeks.org/abstraction-in-java-2/) is a process of hiding implementation details and exposing only the functionality to the user. In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.

**There are two ways to achieve abstraction in Java:**

* Abstract class (0 to 100%)
* Interface (100%)

**Inheritance**

Inheritance is the process of one class inheriting properties and methods from another class in Java. Inheritance is used when we have is-a relationship between objects. Inheritance in Java is implemented using extends keyword.

There are 5 different types of inheritance in java as follows:

* **Single Inheritance:** Class B inherits Class A using extends keyword
* **Multilevel Inheritance:** Class C inherits class B and B inherits class A using extends keyword
* **Hierarchy Inheritance:** Class B and C inherits class A in hierarchy order using extends keyword
* **Multiple Inheritance:** Class C inherits Class A and B. Here A and B both are superclass and C is only one child class. Java is not supporting Multiple Inheritance, but we can implement using Interfaces.
* **Hybrid Inheritance:** Class D inherits class B and class C. Class B and C inherits A. Here same again Class D inherits two superclass, so Java is not supporting Hybrid Inheritance as well.

**Polymorphism**

Polymorphism is the ability to perform many things in many ways. The word Polymorphism is from two different Greek words- poly and morphs. “Poly” means many, and “Morphs” means forms. So polymorphism means many forms. The polymorphism can be present in the case of inheritance also. The functions behave differently based on the actual implementation.

Here are two types of polymorphism as listed below:

1. Static or Compile-time Polymorphism
2. Dynamic or Run-time Polymorphism

**Encapsulation**

Encapsulation is the process of wrapping code and data together into a single unit.

In order to achieve encapsulation in java follow certain steps as proposed below:

* Declare the variables as private
* Declare the [setters and getters](https://www.geeksforgeeks.org/advantages-of-getter-and-setter-over-public-fields-in-java-with-examples) to set and get the variable values

**Inheritance**

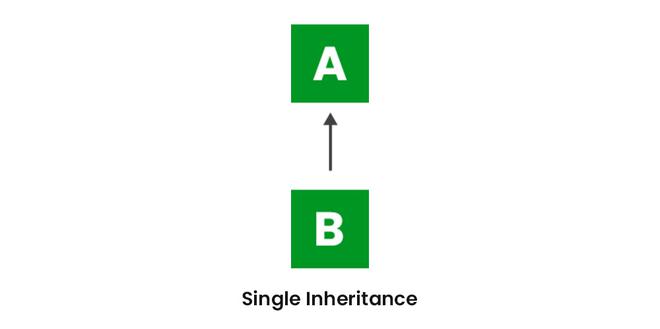
**Inheritance Types**

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance

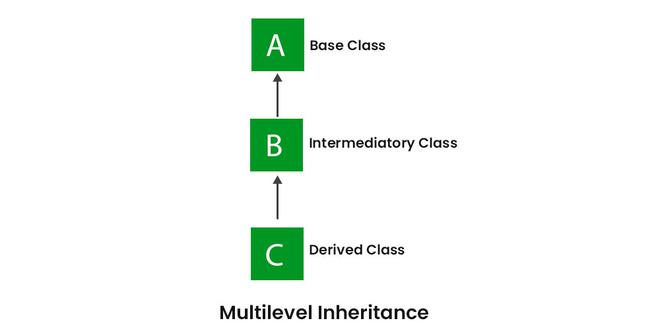
**1. Single Inheritance**

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, ‘A’ is a parent class and ‘B’ is a child class. The class ‘B’ inherits all the properties of the class ‘A’.



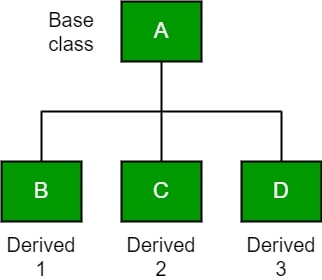
**2. Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the[grandparent’s members](https://www.geeksforgeeks.org/g-fact-91).



**3. Hierarchical Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



Method Overriding and Dynamic Method Dispatch

**Method Overriding**

Overriding in Java occurs when a subclass implements a method which is already defined in the superclass or Base Class. The method in the subclass must have the same signature as in the superclass. It allows the subclass to modify the inherited methods.

Method overriding is a key concept in Java that enables Run-time [polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java/). It allows a subclass to provide its specific implementation for a method inherited from its parent class. The actual method executed is determined by the object’s runtime type, not just the reference variable’s type. This dynamic behaviour is crucial for creating flexible and extensible object-oriented designs.

Constructors and Destructors

Constructor Types

**Types of Constructors in Java**

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

* Default Constructor
* Parameterized Constructor
* Copy Constructor

**1. Default Constructor in Java**

A constructor that has no parameters is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can’t change the default constructor. The default constructor can be implicit or explicit.

**Implicit Default Constructor:**If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn’t take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.

**Explicit Default Constructor:**If we define a constructor that takes no parameters, it’s called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically.Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

**2. Parameterized Constructor in Java**

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Copy Constructor

**Copy Constructor**

In Java, a copy constructor is a special type of constructor that creates an object using another object of the same Java class. It returns a duplicate copy of an existing object of the class.

We can assign a value to the final field but the same cannot be done while using the clone() method. It is used if we want to create a deep copy of an existing object. It is easier to implement in comparison to the clone() method.

The primary use case of a copy constructor is to create a new object with the same state as an existing object. It is particularly useful in scenarios where we need to duplicate an object while maintaining its integrity. By leveraging a copy constructor, we ensure that the new object is independent of the original one, thereby preventing unintended side effects that might occur with shallow copying.

Use of Copy Constructor

We can use the copy constructor if we want to:

* Create a copy of an object that has multiple fields.
* Generate a deep copy of the heavy objects.
* Avoid the use of the Object.clone() method.

Constructor Overloading

Java supports Constructor Overloading in addition to overloading methods. In Java, overloaded constructor is called based on the parameters specified when a [new](https://www.geeksforgeeks.org/new-operator-vs-newinstance-method-java/) is executed.

Object Life Cycle and Garbage Collection

**Garbage collection in Java** is an automatic memory management process that helps Java programs run efficiently. Java programs compile to bytecode that can be run on a[Java Virtual Machine (JVM)](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). When Java programs run on the JVM, [objects](https://www.geeksforgeeks.org/classes-objects-java/) in the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

**Working of Garbage Collection**

* **Java garbage collection** is an **automatic process** that manages memory in the heap.
* It identifies which objects are still in use **(referenced)**and which are not in use (**unreferenced).**
* **Unreferenced objects** can be**deleted**to free up memory.
* The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

**Types of Activities in Java Garbage Collection**

Two types of garbage collection activity usually happen in Java. These are:

1. **Minor or incremental Garbage Collection:**This occurs when unreachable objects in the Young Generation heap memory are removed.
2. **Major or Full Garbage Collection:** This happens when objects that survived minor garbage collection are removed from the Old Generation heap memory. It occurs less frequently than minor garbage collection.

Arrays and Strings

One-Dimensional and Multidimensional Arrays

|  |  |  |
| --- | --- | --- |
| **Basis** | **One Dimension Array** | **Multidimensional** **Array** |
| **Definition** | Store a single list of the element of a similar data type. | Store a ‘list of lists’ of the element of a similar data type. |
| **Representation** | Represent multiple data items as a list. | Represent multiple data items as a table consisting of rows and columns. |
| **Declaration** | The declaration varies for different programming language:   1. For C++,   ***datatype variable\_name[row]*** 2. For Java,   ***datatype [] variable\_name= new datatype[row]*** | The declaration varies for different programming language:   1. For C++,  ***datatype variable\_name[row][column]*** 2. For Java,   ***datatype [][] variable\_name= new datatype[row][column]*** |
| **Dimension** | One | Two |
| **Size(bytes)** | size of(datatype of the variable of the array) \* size of the array | size of(datatype of the variable of the array)\* the number of rows\* the number of columns. |
| **Address calculation.** | Address of a[index] is equal to (base Address+ Size of each element of array \* index). | Address of a[i][j] can be calculated in two ways row-major and column-major   1. **Column Major:**Base Address + Size of each element (number of rows(j-lower bound of the column)+(i-lower bound of the rows)) 2. **Row Major:**Base Address + Size of each element (number of columns(i-lower bound of the row)+(j-lower bound of the column)) |
| **Example** | int arr[5];  //an array with one row and five columns will be created.  {a , b , c , d , e} | int arr[2][5];  //an array with two rows and five columns will be created.                 a  b  c  d  e                 f  g   h  i   j |

String Handling in Java

**String, StringBuilder, and StringBuffer**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| **Introduction** | Introduced in JDK 1.0 | Introduced in JDK 1.5 | Introduced in JDK 1.0 |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread Safety** | Thread Safe | Not Thread Safe | Thread Safe |
| **Memory Efficiency** | High | Efficient | Less Efficient |
| **Performance** | High(No-Synchronization) | High(No-Synchronization) | Low(Due to Synchronization) |
| **Usage** | This is used when we want immutability. | This is used when Thread safety is not required. | This is used when Thread safety is required. |

Array of Objects

An **array of objects** is used to store multiple instances of a [class](https://www.geeksforgeeks.org/classes-objects-java/) within a single [array](https://www.geeksforgeeks.org/arrays-in-java/). This allows us to easily manage a collection of objects when working with large datasets or collections.

**Creating an Array Of Objects In Java**

In Java, we can create an array of objects just like any other array. The only difference is that the array elements are references to objects rather than primitive types.

**1. Declaration**: To declare an array of objects, specify the class name followed by square brackets [].

**2. Instantiation**: After declaring the array, instantiate it using the [new](https://www.geeksforgeeks.org/new-operator-java/) keyword, specifying the size of the array.

**3. Initialization**: Each element of the array must be initialized individually, either via [constructor](https://www.geeksforgeeks.org/constructors-in-java/) or **setter methods**.

Inheritance and Polymorphism

Inheritance Types and Benefits

Inheritance in Java allows a class to acquire the properties and behaviors of another class, promoting code reusability and organization. There are five types of inheritance in Java:

1. **Single Inheritance** – One class inherits from another.
2. **Multilevel Inheritance** – A class inherits from another, which in turn inherits from a third class.
3. **Hierarchical Inheritance** – Multiple classes inherit from a single superclass.
4. **Multiple Inheritance (through Interfaces)** – A class implements multiple interfaces.
5. **Hybrid Inheritance** – Combination of multiple and hierarchical inheritance (not directly supported in Java).

**Benefits of Inheritance**

* **Code Reusability**: Reduces code duplication.
* **Method Overriding**: Allows customization of inherited methods.
* **Polymorphism**: Enables dynamic method invocation.
* **Better Organization**: Promotes a structured class hierarchy.

Method Overriding

**Method Overriding in Java**

Method Overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

**Rules for Method Overriding**

* The method must have the **same name** and **parameters** as in the superclass.
* The return type must be the same or a **covariant type** (subtype of superclass return type).
* The access modifier **cannot be more restrictive** than the overridden method.
* The method **cannot be overridden if it is static, final, or private**.

Dynamic Binding

**Dynamic Binding (Run-Time Polymorphism)**

Dynamic Binding (Late Binding) occurs when the method to be executed is determined at runtime. This is achieved through **method overriding**.

Keyword and Method Hiding

**super Keyword in Java**

The super keyword is used in a subclass to:

* Access the superclass **constructor**.
* Call the **superclass method** if it has been overridden.
* Access the **superclass variable** if it's hidden.

**Method Hiding in Java**

Method Hiding occurs when a subclass defines a **static method** with the same name as a static method in the superclass. Unlike overriding, the method resolution depends on the reference type.

Interfaces and Abstract Classes

Abstract Classes and Methods

**Abstract Class in Java**

An **abstract class** is declared using the abstract keyword. It can contain:

* **Abstract methods** (methods without a body, must be overridden in subclasses)
* **Concrete methods** (methods with implementation)
* **Constructors**
* **Instance variables and static variables**

**Abstract Methods in Java**

An abstract method is a method without a body, declared using the abstract keyword. It must be implemented by subclasses.

Rules for Abstract Methods

* Declared inside an abstract class.
* Cannot have a body (i.e., no implementation).
* Must be overridden by subclasses.

Interfaces: Multiple Inheritance in Java

An **interface** in Java is a blueprint of a class that contains **only abstract methods (before Java 8)** and **default/static methods (Java 8 and later)**. It provides **full abstraction** and allows **multiple inheritance** in Java.

**Key Characteristics of Interfaces:**

* Declared using the interface keyword.
* Methods are **implicitly public and abstract**.
* Variables are **implicitly public, static, and final** (constants).
* A class **implements** an interface using the implements keyword.
* Supports **multiple inheritance**, unlike abstract classes.
* Interfaces enable multiple inheritance in Java.
* They contain abstract methods (before Java 8) and default/static methods (from Java 8).
* Multiple interfaces can be implemented by a single class.
* Default methods must be overridden if there's a conflict.
* Interfaces are used for defining behaviors, while abstract classes are used for partial implementation.

Packages and Access Modifiers

Java Packages

A **package** in Java is a way to **group related classes and interfaces** together. Packages help organize code, prevent naming conflicts, and control access.

**Types of Java Packages**

Java provides two types of packages:

1. **Built-in Packages** (Predefined in Java)
2. **User-defined Packages** (Created by the programmer)

**Built-in Packages in Java**

Java provides many predefined packages that contain useful classes and methods.

**Common Built-in Packages:**

| **Package** | **Description** |
| --- | --- |
| java.lang | Provides core classes (e.g., String, Math, Object, System) |
| java.util | Contains utility classes (e.g., ArrayList, HashMap, Collections) |
| java.io | Provides input and output classes (e.g., FileReader, BufferedReader) |
| java.net | Supports networking (e.g., URL, Socket) |
| java.sql | For database interaction (e.g., Connection, ResultSet) |
| java.time | Provides date and time functionalities |
| javax.swing | Supports GUI applications |

**User-Defined Packages**

A **user-defined package** is a custom package created by the programmer to organize code.

**Steps to Create and Use a Package:**

1. **Create a package** using the package keyword.
2. **Compile** the file using javac -d . FileName.java.
3. **Use** the package in another class using import package\_name.\*;

Access Modifiers

**Types of Access Modifiers**

There are **4 types of access modifiers** available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

**1. Default Access Modifier**

When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default. The default access modifiers are accessible only within the same package.

**2. Private Access Modifier**

The **private access modifier** is specified using the keyword **private**. The methods or data members declared as private are accessible ***only within the class in which they are declared***.

* 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**“.

**3. Protected Access Modifier**

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

**4. Public Access Modifier**

The **public access modifier** is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are ***accessible from everywhere*** in the program. There is no restriction on the scope of public data members.

Exception Handling

Types of Exceptions

| **Feature** | **Checked Exception** | **Unchecked Exception** |
| --- | --- | --- |
| **Behavior** | Checked exceptions are checked at compile time. | Unchecked exceptions are checked at run time. |
| **Base class** | Derived from Exception | Derived from [RuntimeException](https://www.geeksforgeeks.org/java-program-to-handle-runtime-exceptions/" \t "_blank) |
| **Cause** | External factors like file I/O and database connection cause the checked Exception. | Programming bugs like logical Error cause the unchecked Exception. |
| **Handling Requirement** | checked exception must be handled using [try-catch block](https://www.geeksforgeeks.org/try-catch-throw-and-throws-in-java/) or must be declared using [throw keyword](https://www.geeksforgeeks.org/throw-throws-java/) | No handling is required |
| **Examples** | [IOException](https://www.geeksforgeeks.org/handle-an-ioexception-in-java/), [SQLException](https://www.geeksforgeeks.org/how-to-handle-sqlexception-in-jdbc/" \t "_blank), [FileNotFoundException](https://www.geeksforgeeks.org/java-io-filenotfoundexception-in-java/" \t "_blank). | [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/), [ArrayIndexOutOfBoundsException.](https://www.geeksforgeeks.org/array-index-out-of-bounds-exception-in-java/" \t "_blank) |

try, catch, finally, throw, throws

**try-catch**

try-catch block in Java is a mechanism to handle exceptions. This ensures that the application continues to run even if an error occurs. The code inside the try block is executed, and if any exception occurs, it is then caught by the catch block.

**Syntax of try Catch Block**

*try {*

*// Code that might throw an exception*

*} catch (ExceptionType e) {*

*// Code that handles the exception*

*}*

**Difference Between throw and throws**

The main **differences between throw and throws in Java**are follows:

| **Feature** | **throw** | **throws** |
| --- | --- | --- |
| **Definition** | It is used to explicitly throw an exception. | It is used to declare that a method might throw one or more exceptions. |
| **Location** | It is used inside a method or a block of code. | It is used in the method signature. |
| **Usage** | It can throw both checked and unchecked exceptions. | It is only used for checked exceptions. Unchecked exceptions do not require throws. |
| **Responsibility** | The method or block throws the exception. | The method’s caller is responsible for handling the exception. |
| **Flow of Execution** | Stops the current flow of execution immediately. | It forces the caller to handle the declared exceptions. |
| **Example** | throw new ArithmeticException(“Error”); | public void myMethod() throws IOException {} |

Multithreading

Introduction to Threads

Threads are lightweight subprocesses, representing the smallest unit of execution with separate paths. The main advantage of multiple threads is efficiency (allowing multiple things at the same time). For example, in MS Word, one thread automatically formats the document while another thread is taking user input. Additionally, multithreading ensures quick response, as other threads can continue execution even if one gets stuck, keeping the application responsive.

**Life Cycle of Thread**

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State

Creating Threads

we can create threads using two approaches:

1. **Extending the Thread class**
2. **Implementing the Runnable interface**

**Creating a Thread by Extending the Thread Class**

* We create a class that extends Thread.
* We **override the run() method**, which contains the code to execute in the thread.
* We create an object of the class and call **start()** to begin execution.

**Creating a Thread by Implementing Runnable Interface**

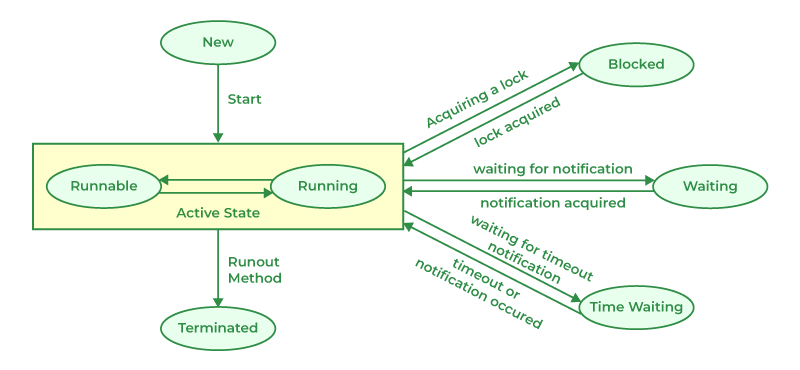
* We create a class that implements Runnable.
* We override the run() method.
* We pass an instance of our class to a Thread object and call **start()**.

Thread Life Cycle

A [thread](https://www.geeksforgeeks.org/multithreading-in-java/) in Java can exist in any one of the following states at any given time. A thread lies only in one of the shown states at any instant:

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

The diagram below represents various states of a thread at any instant.



**Life Cycle of a Thread**

There are multiple states of the thread in a lifecycle as mentioned below:

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 get a small amount of time to run. After running for a while, a thread pauses and gives up the CPU so that other threads can run.
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.

Synchronization and Inter-thread Communication

**synchronization** is crucial for ensuring that multiple threads operate safely on shared resources. Without **Synchronization**, data inconsistency or corruption can occur when multiple threads try to access and modify shared variables simultaneously. In Java, it is a mechanism that ensures that only one thread can access a resource at any given time. This process helps prevent issues such as data inconsistency and [race conditions](https://www.geeksforgeeks.org/race-condition-vulnerability/) when multiple threads interact with shared resources.

**Types of Synchronization**

There are two synchronizations in Java mentioned below:

1. Process Synchronization
2. Thread Synchronization

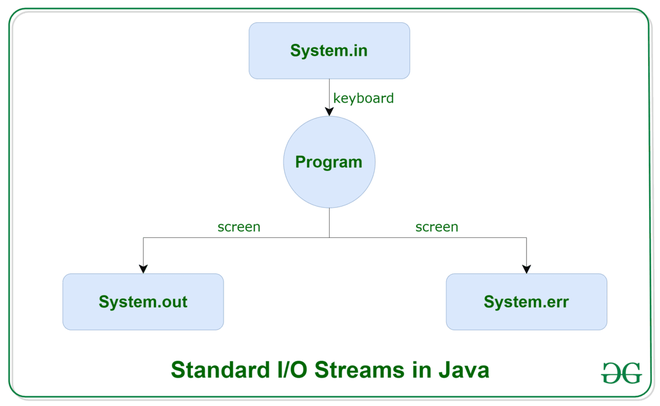
**Inter-thread communication**

**Inter-thread** communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

File Handling

Introduction to File I/O in Java

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc. to fully execute the I/O operations.



**System.in**

This is the **standard input stream(**[**System.in**](https://www.geeksforgeeks.org/java-lang-system-class-java/)) that is used to read characters from the keyboard or any other standard input device.

**System.out**

This is the **standard output stream([System.out](https://www.geeksforgeeks.org/java-lang-system-class-java/)**) that is used to produce the result of a program on an output device like the computer screen. Here is a list of the various print functions that we use to output statements:

**–**[**print()**](https://www.geeksforgeeks.org/difference-between-print-and-println-in-java/)**:** This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.

**Syntax:**

*System.out.print(parameter);*

File Reader and File Writer Classes

**FileWriter**  
FileWriter is useful to create a file writing characters into it.

* This class inherits from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
* FileWriter creates the output file if it is not present already.

**FileReader**

FileReader is useful to read data in the form of characters from a ‘text’ file.

* This class inherited from the InputStreamReader Class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.   
The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object. 

public final void writeObject(Object obj)  
 throws IOException

Collections Framework

Introduction to Collections Framework

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

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

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

List, Set, Map, and Queue Interfaces

**Difference between List,** **Set**,**and Map in Java**

| [List](https://www.geeksforgeeks.org/list-interface-java-examples/) | [Set](https://www.geeksforgeeks.org/set-in-java/) | [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) |
| --- | --- | --- |
| The list interface allows duplicate elements | Set does not allow duplicate elements. | The map does not allow duplicate elements |
| The list maintains insertion order. | Set do not maintain any insertion order. | The map also does not maintain any insertion order. |
| We can add any number of null values. | But in set almost only one null value. | The map allows a single null key at most and any number of null values. |
| List implementation classes are [Array List](https://www.geeksforgeeks.org/arraylist-in-java/), [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/). | Set implementation classes are [HashSet](https://www.geeksforgeeks.org/hashset-in-java/), [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/), and [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/). | Map implementation classes are [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/), [HashTable](https://www.geeksforgeeks.org/hashtable-in-java/), [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/), [ConcurrentHashMap](https://www.geeksforgeeks.org/concurrenthashmap-in-java/), and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/). |
| The list provides get() method to get the element at a specified index. | Set does not provide get method to get the elements at a specified index | The map does not  provide get method to get the elements at a specified index |
| If you need to access the elements frequently by using the index then we can use the list | If you want to create a collection of unique elements then we can use set | If you want to store the data in the form of key/value pair then we can use the map. |
| To traverse the list elements by using Listlterator. | Iterator can be used traverse the set elements | Through keyset, value, and entry set. |

Iterators and List Iterators

| **Iterator** | **ListIterator** |
| --- | --- |
| Can traverse elements present in Collection only in the forward direction. | Can traverse elements present in Collection both in forward and backward directions. |
| Helps to traverse Map, List and Set. | Can only traverse List and not the other two. |
| Indexes cannot be obtained by using Iterator. | It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List. |
| Cannot modify or replace elements present in Collection | We can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time. |
| Certain methods of Iterator are next(), remove() and hasNext(). | Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e). |

Java Input/Output (I/O)

Streams in Java

**Difference between InputStream and OutputStream**

| **InputStream** | **OutputStream** |
| --- | --- |
| 1. It is an abstract class that describes Stream Input. | 1. It is an abstract class that describes Stream Output. |
| 2. InputStream Read data from the source once at a time. | 2. OutputStream Write Data to the destination once at a time. |
| 3. InputStream consist of method which performs:   * Read next byte of data from the input stream and return -1 at the end of the file: **public abstract int read()throws IOException** * Close current InputStream: **public int available()throws IOException** * Returns an estimate of the number of bytes that can be read from the current input stream: **public void close()throws IOException** | 3. Output Stream consists of methods which perform:   * Write a byte to current Outputstream : **public void write(int)throws IOException** * Write array of byte to current output stream :**public void write(byte[])throws IOException** * Flushes the current OutputStream: **public void flush()throws IOException** * Close  current Output Stream. : **public void close()throws IOException** |
| 4. Types of InputStream are:   * FileInputStream * ByteArrayInputStream * FilterInputStream * ObjectInputStream   In these types the most important and mostly used type is FileInputStream. | 4. Types of OutputStream are:   * FileOutputStream * ByteArrayOutputStream * FilterOutputStream * ObjectOutputStream   In these types the most important and mostly used type is FileOutput Stream. |

Reading and Writing Data Using Streams

**FileWriter**  
FileWriter is useful to create a file writing characters into it.

* This class inherits from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
* FileWriter creates the output file if it is not present already.

**FileReader**

FileReader is useful to read data in the form of characters from a ‘text’ file.

* This class inherited from the InputStreamReader Class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.