Java notes

14-Aug-2024

# Why we use Integer instead of int in java.

In Java, int is a primitive data type which is used for storing integer values efficiently. However, Integer is a wrapper class that provides several advantages over int:

1. **Object Creation**: Integer allows you to create objects, which means you can use it in places where objects are required, such as generic data structures.
2. **Null Assignment**: An Integer can be assigned null, but int cannot, which can be useful to represent the absence of a value.
3. [**Utility Methods**: Integer class provides utility methods like parseInt(), toBinaryString(), and others for converting and manipulating numeric values1](https://www.geeksforgeeks.org/difference-between-an-integer-and-int-in-java/).
4. **Automatic Boxing and Unboxing**: Java automatically converts between int and Integer when needed, which is known as boxing and unboxing.

// Using Integer to utilize utility methods

Integer num = Integer.valueOf("123");

int reversed = Integer.reverse(num);

// Automatic boxing (int to Integer)

int primitive = 10;

Integer wrapper = primitive; // automatic boxing

// Automatic unboxing (Integer to int)

Integer wrappedInt = new Integer(20);

int unwrapped = wrappedInt; // automatic unboxing

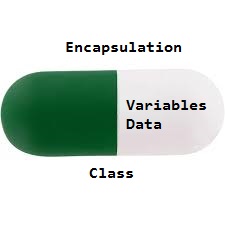
## Utility methods of wrapper class in java

In Java, wrapper classes provide a range of utility methods that are essential for working with primitive types. Here’s a list of some common utility methods provided by the wrapper classes:

1. **valueOf()**:
   * Used to create a wrapper object from a given primitive or String.
   * There are three types of valueOf() methods:
     + Wrapper valueOf(String s): Creates a wrapper object for the given String.
     + Wrapper valueOf(String s, int radix): Creates a wrapper object for the given String with the specified radix.
     + Wrapper valueOf(primitive p): Creates a wrapper object for the given primitive type.
2. **xxxValue()**:
   * These methods return the value of the wrapped object as a primitive type.
   * Each numeric wrapper class has methods like byteValue(), shortValue(), intValue(), longValue(), floatValue(), and doubleValue().
3. **parseXxx()**:
   * Converts a String to a primitive type.
   * Each wrapper class, except for Character, has a parseXxx() method like parseInt(), parseDouble(), parseBoolean(), etc.
4. **toString()**:
   * Returns a string representation of the wrapper object’s value.
5. **compareTo()**:
   * Compares two wrapper objects numerically.
6. **equals()**:
   * Compares the wrapper object to another object for equality.

# Encapsulation

* Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.



* In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

## Inner Class In java

In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is purely object-oriented so bringing it closer to the real world. Now geeks you must be wondering why they were introduced.

Lecture : <https://youtu.be/UVOztdkD7WE?si=eePGEjbmV-4eQLE7>

**There are certain advantages associated with inner classes are as follows:**

* Making code clean and readable.
* Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.
* Optimizing the code module.

*We do use them often as we go advance in java object-oriented programming where we want certain operations to be performed, granting access to limited classes and many more which will be clear as we do discuss and implement all types of inner classes in Java.*

## Types of Inner Classes

There are basically four types of inner classes in java.

1. Nested Inner Class
2. Method Local Inner Classes
3. Static Nested Classes
4. Anonymous Inner Classes

Let us discuss each of the above following types sequentially in-depth alongside a clean java program which is very crucial at every step as it becomes quite tricky as we adhere forwards.

**Type 1:**Nested Inner Class

It can access any private instance variable of the outer class. Like any other instance variable, we can have access modifier private, protected, public, and default modifier. Like class, an interface can also be nested and can have access specifiers.

**Type 4:**Anonymous Inner Classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

* As a subclass of the specified type
* As an implementer of the specified interface

Lecture : <https://youtu.be/LNpUj80JLGI?si=AcwxhnuBw-q49FmK>

# What is Abstract Class in Java?

Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the “abstract” keyword in its class definition.

Syntax :

abstract class Shape

{

int color;

// An abstract function

abstract void draw();

}

In Java, the following some important observations about abstract classes are as follows:

An instance of an abstract class can not be created.

Constructors are allowed.

We can have an abstract class without any abstract method.

There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”

We can define static methods in an abstract class

We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract

If a class contains at least one abstract method then compulsory should declare a class as abstract

If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method

# enum in Java

In Java,Enumerations or Java Enum serve the purpose of representing a group of named constants in a programming language. Java Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command-line flags, etc.

## What is Enumeration or Enum in Java?

A Java enumeration is a[class](https://www.geeksforgeeks.org/object-oriented-programming-in-python-set-1-class-and-its-members/) type. Although we don’t need to instantiate an enum using **new,**it has the same capabilities as other classes. This fact makes Java enumeration a very powerful tool. Just like classes, you can give them[constructors,](https://www.geeksforgeeks.org/constructors-c/) add instance variables and methods, and even implement interfaces.

#### Enum Example:

*The 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).*

One thing to keep in mind is that, unlike classes, enumerations neither inherit other classes nor can get extended(i.e become superclass).  We can also add variables, methods, and constructors to it. The main objective of an enum is to define our own data types(Enumerated Data Types).

# TypeCasting

Typecasting in Java is the process of converting one data type to another data type using the casting operator. When you assign a value from one primitive data type to another type, this is known as type casting. To enable the use of a variable in a specific manner, this method requires explicitly instructing the Java compiler to treat a variable of one data type as a variable of another data type.

**Syntax:**

<datatype> variableName = (<datatype>) value;

## Types of Type Casting

There are two types of Type Casting in java:

* Widening Type Casting
* Narrow Type Casting

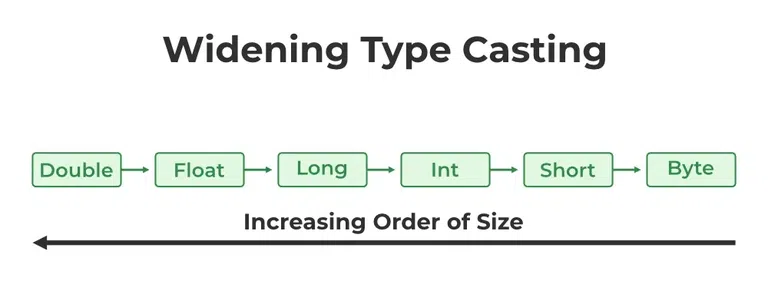
### Widening Type Casting

A lower data type is transformed into a higher one by a process known as widening type casting. Implicit type casting and casting down are some names for it. It occurs naturally. Since there is no chance of data loss, it is secure. Widening Type casting occurs when:

* The target type must be larger than the source type.
* Both data types must be compatible with each other.

**Syntax:**

larger\_data\_type variable\_name = smaller\_data\_type\_variable;



# File Handling in Java

# In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.

## **Why File Handling is Required?**

* File Handling is an integral part of any programming language as file handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.
* In simple words, file handling means reading and writing data to a file.

// Importing File Class

import java.io.File;

class FileHandlingClass {

public static void main(String[] args)

{

// File name specified

File obj = new File("myfile.txt");

System.out.println("File Created!");

}

}

Reading a file through Scanner in io

Code:

File file1 = new File("textFile.txt");  
  
try {  
 Scanner sc = new Scanner(file1);  
 while(sc.hasNextLine()){  
 String line = sc.nextLine();  
 System.*out*.println(line);  
 }  
 sc.close();  
} catch (FileNotFoundException e) {  
 throw new RuntimeException(e);  
}

# Exception

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

## What are Java Exceptions?

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

## **Major reasons why an exception Occurs**

* Invalid user input
* Device failure
* Loss of network connection
* Physical limitations (out-of-disk memory)
* Code errors
* Out of bound
* Null reference
* Type mismatch
* Opening an unavailable file
* Database errors
* Arithmetic errors

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

### Difference between Error and Exception

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

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

## Exception Hierarchy

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

n Java, exceptions can be broadly categorized into two main types: **checked exceptions** and **unchecked exceptions**. Let’s explore each of them:

1. **Checked Exceptions**:
   * Checked exceptions are verified by the Java compiler at **compile time**.
   * They represent errors that occur outside the control of the program.
   * If a method throws a checked exception, the method must either handle the exception using a try-catch block or specify the exception using the throws keyword in its method signature.
   * Common checked exceptions include:
     + IOException: Represents input/output errors (e.g., file not found, file read/write issues).
     + SQLException: Occurs when there are problems with database operations.
     + ParseException: Related to parsing errors (e.g., date parsing).
   * Example of handling a checked exception using throws:

**Java**

private static void checkedExceptionWithThrows() throws FileNotFoundException {

File file = new File("not\_existing\_file.txt");

FileInputStream stream = new FileInputStream(file);

}

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

1. **Unchecked Exceptions**:
   * Unchecked exceptions are not verified by the compiler at compile time.
   * They reflect errors within the program logic itself.
   * We don’t need to declare unchecked exceptions in a method using the throws keyword.
   * Common unchecked exceptions include:
     + ArithmeticException: Occurs during arithmetic operations (e.g., division by zero).
     + NullPointerException: Happens when trying to access a null reference.
     + ArrayIndexOutOfBoundsException: Occurs when accessing an array with an invalid index.
   * Example of an unchecked exception:

**Java**

private static void divideByZero() {

int numerator = 1;

int denominator = 0;

int result = numerator / denominator; // Throws ArithmeticException

}

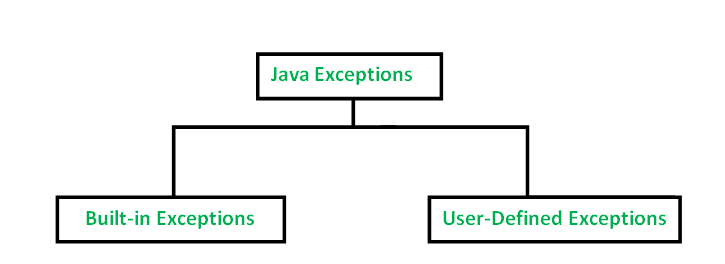
AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

1. **When to Use Checked vs. Unchecked Exceptions**:
   * Use checked exceptions when the client can reasonably recover from the exception.
   * Use unchecked exceptions when the client cannot do anything to recover from the exception.
   * Following this guideline helps separate error-handling code from regular code.

Remember that handling exceptions appropriately is crucial for robust and reliable Java programs! 😊🚀

## Types of Exception in Java with Examples

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



## [**Built-in Exceptions**](https://www.geeksforgeeks.org/built-exceptions-java-examples/)**:**

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:**It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:**This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:**This Exception is raised when a file is not accessible or does not open.
5. **IOException:**It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:**It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:**It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:**It is thrown when accessing a method that is not found.
9. **NullPointerException:**This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:**This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:**This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:**It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :**This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :**This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

### **Examples of Built-in Exception**

**A. Arithmetic exception**

* Java

|  |
| --- |
| // Java program to demonstrate ArithmeticException  **class** ArithmeticException\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {  **int** a = 30, b = 0;  **int** c = a/b;  // cannot divide by zero              System.out.println ("Result = " + c);          }  **catch**(ArithmeticException e) {              System.out.println ("Can't divide a number by 0");          }      }  } |

**Output**

Can't divide a number by 0

**B. NullPointer Exception**

* Java

|  |
| --- |
| //Java program to demonstrate NullPointerException  **class** NullPointer\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              String a = **null**; //null value              System.out.println(a.charAt(0));          } **catch**(NullPointerException e) {              System.out.println("NullPointerException..");          }      }  } |

**Output**

NullPointerException..

**C. StringIndexOutOfBound Exception**

* Java

|  |
| --- |
| // Java program to demonstrate StringIndexOutOfBoundsException  **class** StringIndexOutOfBound\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              String a = "This is like chipping "; // length is 22  **char** c = a.charAt(24); // accessing 25th element              System.out.println(c);          }  **catch**(StringIndexOutOfBoundsException e) {              System.out.println("StringIndexOutOfBoundsException");          }      }  } |

**Output**

StringIndexOutOfBoundsException

**D. FileNotFound Exception**

* Java

|  |
| --- |
| //Java program to demonstrate FileNotFoundException  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **class** File\_notFound\_Demo {    **public** **static** **void** main(String args[])  {  **try** {                // Following file does not exist              File file = **new** File("[E://file.txt](file:///E:\file.txt)");                FileReader fr = **new** FileReader(file);          } **catch** (FileNotFoundException e) {             System.out.println("File does not exist");          }      }  } |

**Output:**

File does not exist

**E. NumberFormat Exception**

* Java

|  |
| --- |
| // Java program to demonstrate NumberFormatException  **class**  NumberFormat\_Demo  {  **public** **static** **void** main(String args[])      {  **try** {              // "akki" is not a number  **int** num = Integer.parseInt ("akki") ;                System.out.println(num);          } **catch**(NumberFormatException e) {              System.out.println("Number format exception");          }      }  } |

**Output**

Number format exception

**F. ArrayIndexOutOfBounds Exception**

* Java

|  |
| --- |
| // Java program to demonstrate ArrayIndexOutOfBoundException  **class** ArrayIndexOutOfBound\_Demo  {  **public** **static** **void** main(String args[])      {  **try**{  **int** a[] = **new** **int**[5];              a[6] = 9; // accessing 7th element in an array of                        // size 5          }  **catch**(ArrayIndexOutOfBoundsException e){              System.out.println ("Array Index is Out Of Bounds");          }      }  } |

**Output**

Array Index is Out Of Bounds

**G. IO Exception**

* Java

|  |
| --- |
| // Java program to demonstrate IOException  **class** IOException\_Demo {    **public** **static** **void** main(String[] args)      {            // Create a new scanner with the specified String          // Object          Scanner scan = **new** Scanner("Hello Geek!");            // Print the line          System.out.println("" + scan.nextLine());            // Check if there is an IO exception          System.out.println("Exception Output: "                             + scan.ioException());            scan.close();      }  } |

**Output:**

Hello Geek!

Exception Output: null

**H. NoSuchMethod Exception**

* Java

|  |
| --- |
| // Java program to demonstrate NoSuchElementException  **public** **class** NoSuchElementException\_Demo {    **public** **static** **void** main(String[] args)      {            Set exampleleSet = **new** HashSet();            Hashtable exampleTable = **new** Hashtable();            exampleleSet.iterator().next();            //accessing Set            exampleTable.elements().nextElement();            //accessing Hashtable              // This throws a NoSuchElementException as there are          // no elements in Set and HashTable and we are          // trying to access elements      }  } |

**I. IllegalArgumentException:**This program, checks whether the person is eligible for voting or not. If the age is greater than or equal to 18 then it will not throw any error. If the age is less than 18 then it will throw an error with the error statement.

Also, we can specify “throw new IllegalArgumentException()” without the error message. We can also specify Integer.toString(variable\_name) inside the IllegalArgumentException() and It will print the argument name which is not satisfied the given condition.

* Java

|  |
| --- |
| /\*package whatever //do not write package name here \*/    **import** java.io.\*;    **class** GFG {  **public** **static** **void** print(**int** a)      {  **if**(a>=18){            System.out.println("Eligible for Voting");            }  **else**{    **throw** **new** IllegalArgumentException("Not Eligible for Voting");              }        }  **public** **static** **void** main(String[] args) {           GFG.print(14);      }  } |

**Output :**

Exception in thread "main" java.lang.IllegalArgumentException: Not Eligible for Voting

at GFG.print(File.java:13)

at GFG.main(File.java:19)

**J. IllegalStateException:**This program, displays the addition of numbers only for Positive integers. If both the numbers are positive then only it will call the print method to print the result otherwise it will throw the IllegalStateException with an error statement. Here, the method is not accessible for non-positive integers.

Also, we can specify the “throw new IllegalStateException()” without the error statement.

* Java

|  |
| --- |
| /\*package whatever //do not write package name here \*/    **import** java.io.\*;    **class** GFG {  **public** **static** **void**  print(**int** a,**int** b)       {           System.out.println("Addition of Positive Integers :"+(a+b));       }    **public** **static** **void** main(String[] args) {  **int** n1=7;  **int** n2=-3;  **if**(n1>=0 && n2>=0)       {           GFG.print(n1,n2);       }  **else**       {  **throw** **new** IllegalStateException("Either one or two numbers are not Positive Integer");       }      }  } |

**Output :**

Exception in thread "main" java.lang.IllegalStateException: Either one or two numbers are not Positive Integer

at GFG.main(File.java:20)

**k. ClassNotFound Exception :**

* Java

|  |
| --- |
| // Java program to demonstrate ClassNotFoundException  **public** **class** ClassNotFoundException\_Demo  {  **public** **static** **void** main(String[] args) {  **try**{              Class.forName("Class1");   // Class1 is not defined          }  **catch**(ClassNotFoundException e){              System.out.println(e);              System.out.println("Class Not Found...");          }      }  } |

**Output**

java.lang.ClassNotFoundException: Class1

Class Not Found...

### **User-Defined Exceptions**

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

The following steps are followed for the creation of a user-defined Exception.

* The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

class MyException extends Exception

* We can write a default constructor in his own exception class.

MyException(){}

* We can also create a parameterized constructor with a string as a parameter.   
  We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

MyException(String str)

{

super(str);

}

* To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

MyException me = new MyException(“Exception details”);

throw me;

* The following program illustrates how to create your own exception class MyException.
* Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
* In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
* If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

**Example**

* Java

|  |
| --- |
| // Java program to demonstrate user defined exception    // This program throws an exception whenever balance  // amount is below Rs 1000  **class** MyException **extends** Exception  {      //store account information  **private** **static** **int** accno[] = {1001, 1002, 1003, 1004};    **private** **static** String name[] =                   {"Nish", "Shubh", "Sush", "Abhi", "Akash"};    **private** **static** **double** bal[] =           {10000.00, 12000.00, 5600.0, 999.00, 1100.55};        // default constructor      MyException() {    }        // parameterized constructor      MyException(String str) { **super**(str); }        // write main()  **public** **static** **void** main(String[] args)      {  **try**  {              // display the heading for the table              System.out.println("ACCNO" + "\t" + "CUSTOMER" +                                             "\t" + "BALANCE");                // display the actual account information  **for** (**int** i = 0; i < 5 ; i++)              {                  System.out.println(accno[i] + "\t" + name[i] +                                                 "\t" + bal[i]);                    // display own exception if balance < 1000  **if** (bal[i] < 1000)                  {                      MyException me =  **new** MyException("Balance is less than 1000");  **throw** me;                  }              }          } //end of try    **catch** (MyException e) {              e.printStackTrace();          }      }  } |

Runtime Error

MyException: Balance is less than 1000

at MyException.main(fileProperty.java:36)

**Output:**

ACCNO CUSTOMER BALANCE

1001 Nish 10000.0

1002 Shubh 12000.0

1003 Sush 5600.0

1004 Abhi 999.0

The fillInStackTrace() method in Java is a powerful tool for managing exception stack traces. Let’s dive into what it does and why it’s useful:

1. **What does**fillInStackTrace()**do?**
   * The fillInStackTrace() method belongs to the java.lang.Throwable class.
   * When you call this method, it records information about the current state of the stack frames for the current thread within the Throwable object.
   * Essentially, it captures the current execution context, including the call stack, which contains information about the methods that led to the exception being thrown.
   * By default, when an exception is thrown, the stack trace includes details from the point where the exception was constructed. However, using fillInStackTrace(), you can replace this original stack trace with one that reflects the current execution context.

# Files

Files

Syntax :

File file = new File("textFile.txt");

try {

file.createNewFile();

} catch (IOException e) {

System.out.println("unable to run a file");

throw new RuntimeException(e);

}

// writing in file.

try {

FileWriter fileWriter = new FileWriter("textFile.txt");

fileWriter.write("this is my first file h/n hello guys");

fileWriter.close();

} catch (IOException e) {

throw new RuntimeException(e);

}

System.out.println("textFile.txt");

// read file through scanner file.

File file1 = new File("textFile.txt");

try {

Scanner sc = new Scanner(file1);

while(sc.hasNextLine()){

String line = sc.nextLine();

System.out.println(line);

}

sc.close();

} catch (FileNotFoundException e) {

throw new RuntimeException(e);

}

// deleting code of a file.

File fileDel = new File("textFile.txt");

if(fileDel.delete()){

System.out.println("file is deleted");

}else{

System.out.println("some error is occur ");

}

\*/

try {

Reader fileReader = new FileReader("textFile2.txt");

try {

fileReader.ready();

char[] arr = new char[100];

// String arrStr;

fileReader.read(arr);

System.out.println(arr);

fileReader.close();

} catch (IOException e) {

throw new RuntimeException(e);

}

} catch (FileNotFoundException e) {

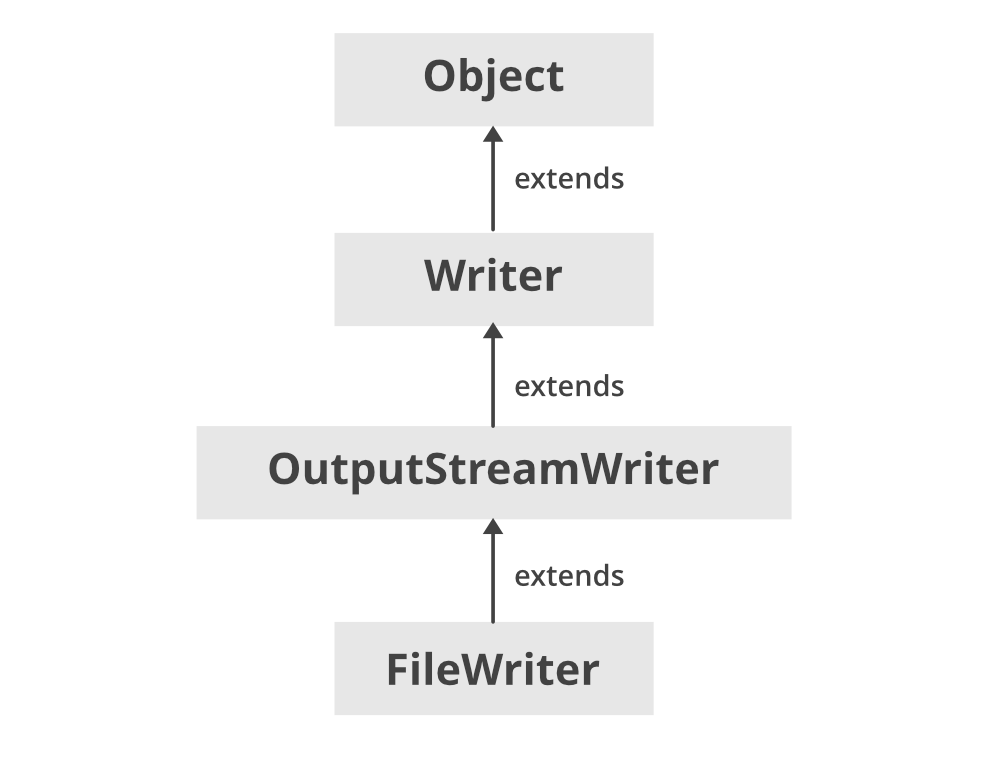
throw new RuntimeException(e);

}

## FileWriter Class in Java

**Java FileWriter** class of java.io package is used to write **data in character** form to file. Java FileWriter class is used to write character-oriented data to a file. It is a character-oriented class that is used for file handling in java.

#### Hierarchy of Java FileWriter Class



* This class inherits from[OutputStreamWriter class](https://www.geeksforgeeks.org/java-io-outputstreamwriter-class-methods/) which in turn inherits from the Writer 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](https://www.geeksforgeeks.org/fileoutputstream-in-java/).
* 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.

FileWriter extends OutputStreamWriter and [Writer](https://www.geeksforgeeks.org/java-io-writer-class-java/) classes. It implements Closeable, Flushable, Appendable, AutoCloseable interfaces.

#### Java FileWriter Class Declaration

public class FileWriter extends OutputStreamWriter

### Constructors of FileWriter Class

**1. FileWriter(File file):** It constructs a FileWriter object given a File object. It throws an **IOException** if the file exists but is a directory rather than a regular file does not exist but cannot be created, or cannot be opened for any other reason.

FileWriter fw = new FileWriter(File file);

**2. FileWriter(File file, boolean append):** It constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning. It throws an **IOException** if the file exists but is a directory rather than a regular file or does not exist but cannot be created, or cannot be opened for any other reason.

FileWriter fw = new FileWriter(File file, boolean append);

**3. FileWriter(FileDescriptor fd):**It constructs a FileWriter object associated with a file descriptor.

FileWriter fw = new FileWriter(FileDescriptor fd);

**4. FileWriter(File file, Charset charset):**It constructs the fileWriter when file and charset is given.

FileWriter fw = new FileWriter(File file, Charset charset);

**5. FileWriter(File file, Charset charset, boolean append):**It constructs the fileWriter when file and charset is given and a boolean indicating whether to append the data written or not.

FileWriter fw = new FileWriter(File file, Charset charset, boolean append);

**6. FileWriter(String fileName):** It constructs a FileWriter object given a file name.

FileWriter fw = new FileWriter(String fileName);

**7. FileWriter(String fileName, Boolean append):** It constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

FileWriter fw = new FileWriter(String fileName, Boolean append);

**8. FileWriter(String fileName, Charset charset):**It constructs a FileWriter when a fileName and charset is given.

FileWriter fw = new FileWriter(String fileName, Charset charset);

**9. FileWriter(String fileName, Charset charset, boolean append):**It constructs a fileWriter when a fileName and a charset are given and a boolean variable indicating whether to append data or not.

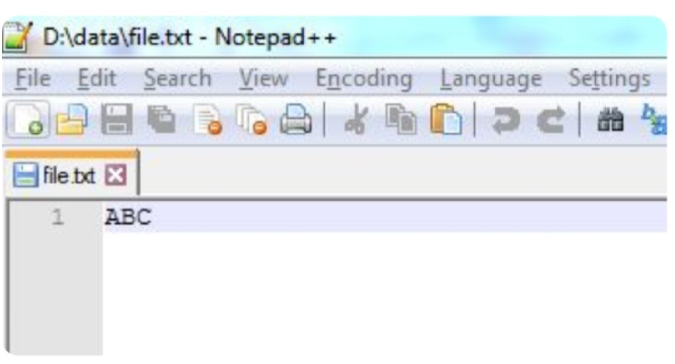
FileWriter fw = new FileWriter(String fileName, Charset charset, boolean append);

**Example:**

* Java

|  |
| --- |
| // Java program to create a text File using FileWriter    **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.\*;  **class** GFG {  **public** **static** **void** main(String[] args)  **throws** IOException      {          // initialize a string          String str = "ABC";  **try** {                // attach a file to FileWriter              FileWriter fw                  = **new** FileWriter("D:/data/file.txt");                // read each character from string and write              // into FileWriter  **for** (**int** i = 0; i < str.length(); i++)                  fw.write(str.charAt(i));                System.out.println("Successfully written");                // close the file              fw.close();          }  **catch** (Exception e) {              e.getStackTrace();          }      }  } |

**Output:**



### **Overwriting vs Appending the File**

While creating a Java FileWriter, we can decide whether we want to append the file to an existing file, or we want to overwrite any existing file. This can be decided by choosing the appropriate constructor. The constructor for **overwriting**any existing file takes only **one parameter that is a file name.**

Writer fileWriter = new FileWriter("c:\\data\\output.txt");

The constructor for **appending the file**or overwriting the file, takes**two parameters, the file name and a boolean variable**which decides whether to append or overwrite the file

Writer fileWriter = new FileWriter("c:\\data\\output.txt", true); // appends to file

Writer fileWriter = new FileWriter("c:\\data\\output.txt", false); // overwrites file

### Basic Methods of FileWriter Class

#### 1. Write()

* **write(int a):**This method writes a single character specified by int a.
* **write(String str, int pos,** **int length):**This method writes a portion of the string from position **pos** until the **length** number of characters.
* **write(char ch[], int pos, int length):**This method writes the position of characters from array ch[] from position **pos** till **length** number of characters.
* **write(char ch[]):**This method writes an array of characters specified by ch[].
* **write(String st):**This method writes a string value specified by ‘st’ into the file.
* Java

|  |
| --- |
| // Java program to write text to file    **import** java.io.FileWriter;    **public** **class** GFG {    **public** **static** **void** main(String args[])      {            String data = "Welcome to gfg";    **try** {              // Creates a FileWriter              FileWriter output                  = **new** FileWriter("output.txt");                // Writes the string to the file              output.write(data);                // Closes the writer              output.close();          }    **catch** (Exception e) {              e.getStackTrace();          }      }  } |

**Output:**

The file output.txt will contain "**Welcome to gfg**" text.

#### 2. getEncoding()

This method is used to get the type of encoding that is used for writing the data.

* Java

|  |
| --- |
| // java program to show the usage  // of getEncoding() function    **import** java.io.FileWriter;  **import** java.nio.charset.Charset;    **class** Main {  **public** **static** **void** main(String[] args)      {            String file = "output.txt";    **try** {              // Creates a FileReader with default encoding              FileWriter o1 = **new** FileWriter(file);                // Creates a FileReader specifying the encoding              FileWriter o2 = **new** FileWriter(                  file, Charset.forName("UTF11"));                // Returns the character encoding of the reader              System.out.println("Character encoding of o1: "                                 + o1.getEncoding());              System.out.println("Character encoding of o2: "                                 + o2.getEncoding());                // Closes the reader              o1.close();              o2.close();          }    **catch** (Exception e) {              e.getStackTrace();          }      }  } |

**Output:**

The character encoding of output1: Cp1253

The character encoding of output2: UTF11

In the above example, we have created 2 file writer named output1 and output2.

* **output1:**does not specify the character encoding. Hence, the getEncoding() method returns the default character encoding.
* **output2:**specifies the character encoding, **UTF11**. Hence, the getEncoding() method returns the specified character encoding.

#### 3. close() method:

After finishing writing characters to a FileWriter, we should close it. And this is done by calling the close() method.

try {

// Creates a FileReader with default encoding

FileWriter o1 = new FileWriter(file);

// Creates a FileReader specifying the encoding

FileWriter o2 = new FileWriter(file, Charset.forName("UTF11"));

// Returns the character encoding of the reader

System.out.println("Character encoding of o1: " + o1.getEncoding());

System.out.println("Character encoding of o2: " + o2.getEncoding());

**// Closes the FileWriter**

o1.close();

o2.close();

}

#### FileWriter vs FileOutputStream

* FileWriter writes streams of characters while FileOutputStream is meant for writing streams of raw bytes.
* FileWriter deals with the character of 16 bits while on the other hand, FileOutputStream deals with 8-bit bytes.
* FileWriter handles Unicode strings while FileOutputStream writes bytes to a file and it does not accept characters or strings and therefore for accepting strings, it needs to be wrapped up with OutputStreamWriter.

#### Methods of FileWriter Class

| **S. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void write(String text) | It is used to write the string into FileWriter. |
| 2. | void write(char c) | It is used to write the char into FileWriter. |
| 3. | void write(char[] c) | It is used to write a char array into FileWriter. |
| 4. | void flush() | It is used to flushes the data of FileWriter. |
| 5. | void close() | It is used to close the FileWriter. |

#### Methods of OutputStreamWriter Class

| **S. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | flush() | Flushes the stream. |
| 2. | getEncoding() | Returns the name of the character encoding being used by this stream. |
| 3. | write​(char[] cbuf, int off, int len) | Writes a portion of an array of characters. |
| 4. | write​(int c) | Writes a single character. |
| 5. | write​(String str, int off, int len) | Writes a portion of a string. |

#### Methods of Writer Class

| **S. No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | [append​(char c)](https://www.geeksforgeeks.org/writer-appendchar-method-in-java-with-examples/) | Appends the specified character to this writer. |
| 2. | append​(CharSequence csq) | Appends the specified character sequence to this writer. |
| 3. | append​(CharSequence csq, int start, int end) | Appends a subsequence of the specified character sequence to this writer. |
| 4. | [close()](https://www.geeksforgeeks.org/writer-close-method-in-java-with-examples/) | Closes the stream, flushing it first. |
| 5. | nullWriter() | Returns a new Writer which discards all characters. |
| 6. | [write​(char[] cbuf)](https://www.geeksforgeeks.org/writer-writechar-method-in-java-with-examples/) | Writes an array of characters. |
| 7. | [write​(String str)](https://www.geeksforgeeks.org/writer-writestring-method-in-java-with-examples/) | Writes a string. |

# Recursion

# Collection Framework



## Methods of List

collection framework function

List Methods

1. .add();

2. .add(index,value);

3. .remove(index);

4. .remove(Integer.ValueOf(value));

5. .clear();

6. .set(index,value); //it replace the value of the given position

7. .contains(value); //it helps to find the value is present in the list or not it returns true or false

8. .size();

9. .

## Stack methods

* 1. push(value);

## [LinkedList Methods](https://www.digitalocean.com/community/tutorials/java-linkedlist-linkedlist-java#java-linkedlist-list-methods)

In this section we will discuss some of the useful and frequently used Java LinkedList methods. The following methods are inherited from List or Collection interface:

1. int size(): to get the number of elements in the list.
2. boolean isEmpty(): to check if list is empty or not.
3. boolean contains(Object o): Returns true if this list contains the specified element.
4. Iterator iterator(): Returns an iterator over the elements in this list in proper sequence.
5. Object[] toArray(): Returns an array containing all of the elements in this list in proper sequence.
6. boolean add(E e): Appends the specified element to the end of this list.
7. boolean remove(Object o): Removes the first occurrence of the specified element from this list.
8. boolean retainAll(Collection c): Retains only the elements in this list that are contained in the specified collection.
9. void clear(): Removes all the elements from the list.
10. E get(int index): Returns the element at the specified position in the list.
11. E set(int index, E element): Replaces the element at the specified position in the list with the specified element.
12. ListIterator listIterator(): Returns a list iterator over the elements in the list.
13. List subList(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa.

## Comparators

A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class**.**

**Return 1 : means swap**

**Return -1 : not swap.**

# TreeSet

TreeSet is one of the most important implementations of the [SortedSet interface](https://www.geeksforgeeks.org/sortedset-java-examples/) in Java that uses a [Tree](https://www.geeksforgeeks.org/binary-tree-data-structure/) for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit [comparator](https://www.geeksforgeeks.org/comparator-interface-java/) is provided. This must be consistent with equals if it is to correctly implement the [Set interface](https://www.geeksforgeeks.org/set-in-java/).

It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a [NavigableSet interface](https://www.geeksforgeeks.org/navigableset-java-examples/) by inheriting [AbstractSet class](https://www.geeksforgeeks.org/abstractset-class-in-java-with-examples/).

## Hierarchy Diagram of TreeSet



It can clearly be perceived from the above image that the navigable set extends the sorted set interface. Since a set doesn’t retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

***Note:***

* *An object is said to be comparable if and only if the corresponding class implements a****Comparable interface****.*
* [*String*](https://www.geeksforgeeks.org/string-class-in-java/)*, [StringBuffer](https://www.geeksforgeeks.org/stringbuffer-class-in-java/) class and all the Wrapper classes already implements Comparable interface Hence, we DO NOT get a ClassCastException.****But if we are creating TreeSet of user defined classes or any Java classes which does not implements comparable interface we will get ClassCastException. to solve this problem we can either implement Comparable to our user defined class or we can pass Comparator object in Constructor while creating the set.***
* *For an empty tree-set, when trying to insert null as the first value, one will get NPE from JDK 7. From JDK 7 onwards, null is not at all accepted by TreeSet. However, up to JDK 6, null was accepted as the first value, but any insertion of more null values in the TreeSet resulted in NullPointerException. Hence, it was considered a bug and thus removed in JDK 7.*
* *TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.*
* *The insertion of null values into a TreeSet throws [NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/) because while insertion of null, it gets compared to the existing elements, and null cannot be compared to any value.*

## **Internal Working of TreeSet in Java**

TreeSet is basically an implementation of a self-balancing binary search tree like a [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/). Therefore operations like add, remove, and search takes O(log(N)) time. The reason is that in a self-balancing tree, it is made sure that the height of the tree is always O(log(N)) for all the operations. Therefore, this is considered as one of the most efficient data structures in order to store the huge sorted data and perform operations on it. However, operations like printing N elements in the sorted order take O(N) time.

**Now let us discuss Synchronized TreeSet prior moving ahead.**The implementation of a TreeSet is not synchronized. This means that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing some object that naturally [encapsulates](https://www.geeksforgeeks.org/encapsulation-in-java/) the set. If no such object exists, the set should be “wrapped” using the [Collections.synchronizedSortedSet](https://www.geeksforgeeks.org/collections-synchronizedsortedset-method-in-java-with-examples/) method. This is best done at the creation time, to prevent accidental unsynchronized access to the set. It can be achieved as shown below as follows:

TreeSet ts = new TreeSet();

Set syncSet = Collections.synchronziedSet(ts);

### **Constructors of TreeSet Class are as follows:**

In order to create a TreeSet, we need to create an object of the TreeSet class. The TreeSet class consists of various constructors which allow the possible creation of the TreeSet. The following are the constructors available in this class:

**1. TreeSet():** This constructor is used to build an empty TreeSet object in which elements will get stored in default natural sorting order.

**Syntax:** If we wish to create an empty TreeSet with the name ts, then, it can be created as:

TreeSet ts = new TreeSet();

**2. TreeSet(Comparator):** This constructor is used to build an empty TreeSet object in which elements will need an external specification of the sorting order.

**Syntax:**If we wish to create an empty TreeSet with the name ts with an external sorting phenomenon, then, it can be created as:

TreeSet ts = new TreeSet(Comparator comp);

**3. TreeSet(Collection):** This constructor is used to build a TreeSet object containing all the elements from the given collection in which elements will get stored in default natural sorting order. In short, this constructor is used when any conversion is needed from any Collection object to TreeSet object.

**Syntax:**If we wish to create a TreeSet with the name ts, then, it can be created as follows:

TreeSet t = new TreeSet(Collection col);

**4. TreeSet(SortedSet):** This constructor is used to build a TreeSet object containing all the elements from the given [sortedset](https://www.geeksforgeeks.org/sortedset-java-examples/) in which elements will get stored in default natural sorting order. In short, this constructor is used to convert the SortedSet object to the TreeSet object.

**Syntax:**If we wish to create a TreeSet with the name ts, then, it can be created as follows:

TreeSet t = new TreeSet(SortedSet s);

## **Methods in TreeSet Class**

**Method in TreeSet Class are depicted below**in tabular format which later on we will be implementing to showcase in the implementation part.

TreeSet implements [SortedSet](https://www.geeksforgeeks.org/sortedset-java-examples/) so it has the availability of all methods in Collection, [Set](https://www.geeksforgeeks.org/set-in-java/), and [SortedSet interfaces](https://www.geeksforgeeks.org/sortedset-java-examples/). Following are the methods in the Treeset interface. In the table below, the “?” signifies that the method works with any type of object including user-defined objects.

| **Method** | **Description** |
| --- | --- |
| [**add(Object o)**](https://www.geeksforgeeks.org/treeset-add-method-in-java/) | This method will add the specified element according to the same sorting order mentioned during the creation of the TreeSet. Duplicate entries will not get added. |
| [**addAll(Collection c)**](https://www.geeksforgeeks.org/treeset-addall-method-in-java/) | This method will add all elements of the specified Collection to the set. Elements in the Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet. |
| [**ceiling?(E e)**](https://www.geeksforgeeks.org/treeset-ceiling-method-in-java-with-examples/) | This method returns the least element in this set greater than or equal to the given element, or null if there is no such element. |
| [**clear()**](https://www.geeksforgeeks.org/treeset-clear-method-in-java/) | This method will remove all the elements. |
| [**clone()**](https://www.geeksforgeeks.org/treeset-clone-method-in-java/) | The method is used to return a shallow copy of the set, which is just a simple copied set. |
| [**Comparator comparator()**](https://www.geeksforgeeks.org/treeset-comparator-method-in-java/) | This method will return the Comparator used to sort elements in TreeSet or it will return null if the default natural sorting order is used. |
| [**contains(Object o)**](https://www.geeksforgeeks.org/treeset-contains-method-in-java/) | This method will return true if a given element is present in TreeSet else it will return false. |
| [**descendingIterator?()**](https://www.geeksforgeeks.org/treeset-descendingiterator-method-in-java-with-examples/) | This method returns an iterator over the elements in this set in descending order. |
| [**descendingSet?()**](https://www.geeksforgeeks.org/treeset-descendingset-method-in-java-with-examples/) | This method returns a reverse order view of the elements contained in this set. |
| [**first()**](https://www.geeksforgeeks.org/treeset-first-method-in-java/) | This method will return the first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException. |
| [**floor?(E e)**](https://www.geeksforgeeks.org/treeset-floor-method-in-java-with-examples/) | This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element. |
| [**headSet(Object toElement)**](https://www.geeksforgeeks.org/treeset-headset-method-in-java/) | This method will return elements of TreeSet which are less than the specified element. |
| [**higher?(E e)**](https://www.geeksforgeeks.org/treeset-higher-method-in-java-with-examples/) | This method returns the least element in this set strictly greater than the given element, or null if there is no such element. |
| [**isEmpty()**](https://www.geeksforgeeks.org/treeset-isempty-method-in-java/) | This method is used to return true if this set contains no elements or is empty and false for the opposite case. |
| [**Iterator iterator()**](https://www.geeksforgeeks.org/treeset-iterator-method-in-java/) | Returns an iterator for iterating over the elements of the set. |
| [**last()**](https://www.geeksforgeeks.org/treeset-last-method-in-java/) | This method will return the last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException. |
| [**lower?(E e)**](https://www.geeksforgeeks.org/treeset-lower-method-in-java/) | This method returns the greatest element in this set strictly less than the given element, or null if there is no such element. |
| [**pollFirst?()**](https://www.geeksforgeeks.org/treeset-pollfirst-method-in-java/) | This method retrieves and removes the first (lowest) element, or returns null if this set is empty. |
| [**pollLast?()**](https://www.geeksforgeeks.org/treeset-polllast-method-in-java-with-example/) | This method retrieves and removes the last (highest) element, or returns null if this set is empty. |
| [**remove(Object o)**](https://www.geeksforgeeks.org/treeset-remove-method-in-java/) | This method is used to return a specific element from the set. |
| [**size()**](https://www.geeksforgeeks.org/treeset-size-method-in-java/) | This method is used to return the size of the set or the number of elements present in the set. |
| **spliterator()** | This method creates a late-binding and fail-fast Spliterator over the elements in this set. |
| [**subSet(Object fromElement, Object toElement)**](https://www.geeksforgeeks.org/treeset-subset-method-in-java/) | This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive. |
| [**tailSet(Object fromElement)**](https://www.geeksforgeeks.org/treeset-tailset-method-in-java/) | This method will return elements of TreeSet which are greater than or equal to the specified element. |

## Various Operations over TreeSet in Java

Here we will be performing various operations over the TreeSet object to get familiar with the methods and concepts of TreeSet in java. Let’s see how to perform a few frequently used operations on the TreeSet. They are listed as follows:

* Adding elements
* Accessing elements
* Removing elements
* Iterating through elements

Now let us discuss each operation individually one by one later alongside grasping with the help of a clean java program.

### **Operation 1:**Adding Elements

In order to add an element to the TreeSet, we can use the [add() method](https://www.geeksforgeeks.org/treeset-add-method-in-java/). However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the TreeSet.

#### **Example:**

Java

*// Java code to Illustrate Addition of Elements to TreeSet*

*// Importing utility classes*

**import** **java.util.\***;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// Creating a Set interface with*

*// reference to TreeSet class*

*// Declaring object of string type*

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

*// Elements are added using add() method*

ts.add("Geek");

ts.add("For");

ts.add("Geeks");

*// Print all elements inside object*

System.out.println(ts);

}

}

**Output**

[For, Geek, Geeks]

### **Operation 2:**Accessing the Elements

After adding the elements, if we wish to access the elements, we can use inbuilt methods like [contains()](https://www.geeksforgeeks.org/treeset-contains-method-in-java/), [first()](https://www.geeksforgeeks.org/treeset-first-method-in-java/), [last()](https://www.geeksforgeeks.org/treeset-last-method-in-java/), etc.

#### **Example:**

Java

*// Java code to Illustrate Working of TreeSet by*

*// Accessing the Element of TreeSet*

*// Importing utility classes*

**import** **java.util.\***;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// Creating a NavigableSet object with*

*// reference to TreeSet class*

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

*// Elements are added using add() method*

ts.add("Geek");

ts.add("For");

ts.add("Geeks");

*// Printing the elements inside the TreeSet object*

System.out.println("Tree Set is " + ts);

String check = "Geeks";

*// Check if the above string exists in*

*// the treeset or not*

System.out.println("Contains " + check + " "

+ ts.contains(check));

*// Print the first element in*

*// the TreeSet*

System.out.println("First Value " + ts.first());

*// Print the last element in*

*// the TreeSet*

System.out.println("Last Value " + ts.last());

String val = "Geek";

*// Find the values just greater*

*// and smaller than the above string*

System.out.println("Higher " + ts.higher(val));

System.out.println("Lower " + ts.lower(val));

}

}

**Output**

Tree Set is [For, Geek, Geeks]

Contains Geeks true

First Value For

Last Value Geeks

Higher Geeks

Lower For

### **Operation 3:**Removing the Values

The values can be removed from the TreeSet using the [remove()](https://www.geeksforgeeks.org/treeset-remove-method-in-java/) method. There are various other methods that are used to remove the first value or the last value.

#### **Example:**

Java

*// Java Program to Illustrate Removal of Elements*

*// in a TreeSet*

*// Importing utility classes*

**import** **java.util.\***;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// Creating an object of NavigableSet*

*// with reference to TreeSet class*

*// Declaring object of string type*

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

*// Elements are added*

*// using add() method*

ts.add("Geek");

ts.add("For");

ts.add("Geeks");

ts.add("A");

ts.add("B");

ts.add("Z");

*// Print and display initial elements of TreeSet*

System.out.println("Initial TreeSet " + ts);

*// Removing a specific existing element inserted*

*// above*

ts.remove("B");

*// Printing the updated TreeSet*

System.out.println("After removing element " + ts);

*// Now removing the first element*

*// using pollFirst() method*

ts.pollFirst();

*// Again printing the updated TreeSet*

System.out.println("After removing first " + ts);

*// Removing the last element*

*// using pollLast() method*

ts.pollLast();

*// Lastly printing the elements of TreeSet remaining*

*// to figure out pollLast() method*

System.out.println("After removing last " + ts);

}

}

**Output**

Initial TreeSet [A, B, For, Geek, Geeks, Z]

After removing element [A, For, Geek, Geeks, Z]

After removing first [For, Geek, Geeks, Z]

After removing last [For, Geek, Geeks]

### **Operation 4:**[**Iterating through the TreeSet**](https://www.geeksforgeeks.org/how-to-loop-over-treeset-in-java/)

There are various ways to iterate through the TreeSet. The most famous one is to use the [enhanced for loop](https://www.geeksforgeeks.org/loops-in-java/). and geeks mostly you would be iterating the elements with this approach while practicing questions over TreeSet as this is most frequently used when it comes to tree, maps, and graphs problems.

#### **Example:**

Java

*// Java Program to Illustrate Working of TreeSet*

*// Importing utility classes*

**import** **java.util.\***;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// Creating an object of Set with reference to*

*// TreeSet class*

*// Note: You can refer above media if geek*

*// is confused in programs why we are not*

*// directly creating TreeSet object*

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

*// Adding elements in above object*

*// using add() method*

ts.add("Geek");

ts.add("For");

ts.add("Geeks");

ts.add("A");

ts.add("B");

ts.add("Z");

*// Now we will be using for each loop in order*

*// to iterate through the TreeSet*

**for** (String value : ts)

*// Printing the values inside the object*

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

System.out.println();

}

}

**Output**

A, B, For, Geek, Geeks, Z,

## Features of a TreeSet

* TreeSet implements the [SortedSet](https://www.geeksforgeeks.org/sortedset-java-examples/) interface. So, duplicate values are not allowed.
* Objects in a TreeSet are stored in a sorted and ascending order.
* TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
* If we are depending on the default natural sorting order, the objects that are being inserted into the tree should be homogeneous and comparable. TreeSet does not allow the insertion of heterogeneous objects. It will throw a [classCastException](https://www.geeksforgeeks.org/built-exceptions-java-examples/) at Runtime if we try to add heterogeneous objects.
* The TreeSet can only accept generic types which are comparable.  
  For example, the StringBuffer class implements the Comparable interface.

Java

*// Java code to illustrate What if Heterogeneous*

*// Objects are Inserted*

*// Importing all utility classes*

**import** **java.util.\***;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// Object creation*

Set<StringBuffer> ts = **new** TreeSet<>();

*// Adding elements to above object*

*// using add() method*

ts.add(**new** StringBuffer("A"));

ts.add(**new** StringBuffer("Z"));

ts.add(**new** StringBuffer("L"));

ts.add(**new** StringBuffer("B"));

ts.add(**new** StringBuffer("O"));

ts.add(**new** StringBuffer(1));

*// Note: StringBuffer implements Comparable*

*// interface*

*// Printing the elements*

System.out.println(ts);

}

}

**Output**

[, A, B, L, O, Z]

## Natural Ordering

Natural ordering is a general computing concept,, not a Java® concept. It means that for any two instances *x* and *y* of a type, one is large than the other or the two are equal in size, and that relationship doesn't change if no other data change, and there is one criterion for that comparison. [Actually, natural ordering can be applied backwards.] Example: floating‑point numbers: 1.23 is always smaller than 1.234, and -9.99 is the same as -9.99. And you won't think of any other way to order such numbers. [Yes, the [Double](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Double.html) datatype supports -0.0 and [NaN](https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Double.html" \l "NaN" \t "_blank).] If you read that link, you will find how Double interprets natural ordering in light of the IEEE754 standard.

# Stream API

**Java 8** introduces Stream, which is a new abstract layer, and some new additional packages in Java 8 called java.util.stream. A Stream is a sequence of components that can be processed sequentially. These packages include classes, interfaces, and enum to allow functional-style operations on the elements.

The stream can be used by importing java.util.stream package. Stream API is used to process collections of objects. Streams are designed to be efficient and can support improving your program’s performance by allowing you to avoid unnecessary loops and iterations. Streams can be used for filtering, collecting, printing, and converting from one data structure to another, etc.

This Java 8 Stream Tutorial will cover all the basic to advanced concepts of **Java 8 stream** like Java 8 filter and collect operations, and real-life examples of Java 8 streams.

## **Prerequisites for Java Stream**

Before proceeding to [Java 8](https://www.geeksforgeeks.org/java-8-features/), it’s recommended to have a basic knowledge of Java 8 and its important concepts such as lambda expression, Optional, method references, etc.

***Note:***

* *If we want to represent a group of objects as a single entity then we should go for*[***collection***](https://www.geeksforgeeks.org/collections-in-java-2)*.*
* *But if we want to process objects from the collection then we should go for streams.*

If we want to use the concept of streams then stream() is the method to be used. Stream is available as an interface.

#### Syntax:

Stream s = c.stream();

In the above pre-tag, ‘c’ refers to the collection. So on the collection, we are calling the ***stream() method*** and at the same time, we are storing it as the Stream object. Henceforth, this way we are getting the Stream object.

***Note:****Streams are present in java’s utility package named****java.util.stream***

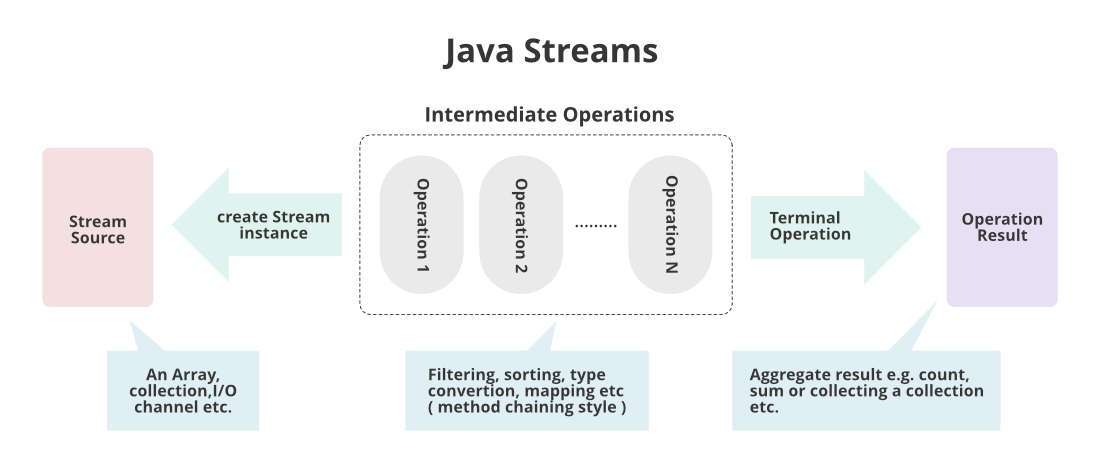
Let us now start with the basic components involved in streams. They as listed as follows:

* Sequence of Elements
* Source
* Aggregate Operations
* Pipelining
* Internal iteration

## **Features of Java Stream**

* A stream is not a data structure instead it takes input from the [Collections](https://www.geeksforgeeks.org/java-collection-tutorial/), [Arrays](https://www.geeksforgeeks.org/arrays-in-java/)**,** or [I/O channels](https://www.geeksforgeeks.org/i-o-channels-and-its-types/).
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Before moving ahead in the concept consider an example in which we are having ArrayList of integers, and we suppose we apply a filter to get only even numbers from the object inserted.



## **How does Stream Work Internally?**

In streams,

* To filter out from the objects we do have a function named ***filter()***
* To impose a condition we do have a logic of predicate which is nothing but a functional interface. Here function interface can be replaced by a random expression. Hence, we can directly impose the condition check-in our predicate.
* To collect elements we will be using ***Collectors.toList()*** to collect all the required elements.
* Lastly, we will store these elements in a List and display the outputs on the console.

#### **Example**

Java

*// Java Program to illustrate FILTER*

*// & COLLECT Operations*

**import** **java.io.\***;

**import** **java.util.\***;

**import** **java.util.stream.\***;

*// Main class*

**public** **class** **GFG** {

*// Main driver method*

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

{

*// Creating an ArrayList object of integer type*

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

*// Inserting elements to ArrayList class object*

*// Custom input integer numbers*

al.add(2);

al.add(6);

al.add(9);

al.add(4);

al.add(20);

*// First lets print the collection*

System.out.println("Printing the collection : "

+ al);

*// Printing new line for better output readability*

System.out.println();

*// Stream operations*

*// 1. Getting the stream from this collection*

*// 2. Filtering out only even elements*

*// 3. Collecting the required elements to List*

List<Integer> ls

= al.stream()

.filter(i -> i % 2 == 0)

.collect(Collectors.toList());

*// Print the collection after stream operation*

*// as stored in List object*

System.out.println(

"Printing the List after stream operation : "

+ ls);

}

}

**Output**

Printing the collection : [2, 6, 9, 4, 20]

Printing the List after stream operation : [2, 6, 4, 20]

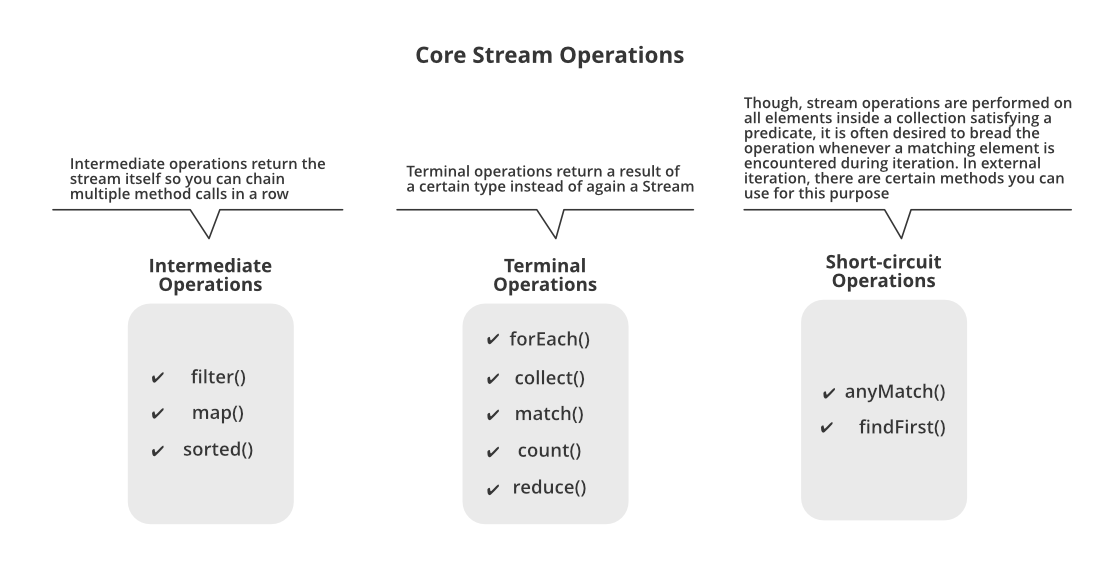
### **Explanation of the above program:**

In our collection object, we were having elements entered using the add() operation. After processing the object in which they were stored through streams we impose a condition in the predicate of streams to get only even elements, we get elements in the object as per our requirement.  Hence, streams helped us this way in processing over-processed collection objects.

## **Various Core Operations Over Streams**

There are broadly 3 types of operations that are carried over streams namely as follows as depicted from the image shown above:

1. Intermediate operations
2. Terminal operations
3. Short-circuit operations



Let us do discuss out intermediate operations here only in streams to a certain depth with the help of an example in order to figure out other operations via theoretical means.

### 1. Intermediate Operations:

Intermediate operations transform a stream into another stream. Some common intermediate operations include:

1. **filter():**Filters elements based on a specified condition.
2. **map():**Transforms each element in a stream to another value.
3. **sorted():**Sorts the elements of a stream.

All three of them are discussed below as they go hand in hand in nearly most of the scenarios and to provide better understanding by using them later by implementing in our clean Java programs below. As we already have studied in the above example of which we are trying to filter processed objects can be interpreted as filter() operation operated over streams.

### 2. Terminal Operations

Terminal Operations are the operations that on execution return a final result as an absolute value.

1. **collect():**It is used to return the result of the intermediate operations performed on the stream.
2. **forEach():**It iterates all the elements in a stream.
3. **reduce():**It is used to reduce the elements of a stream to a single value.

### 3. Short Circuit Operations

Short-circuit operations provide performance benefits by avoiding unnecessary computations when the desired result can be obtained early. They are particularly useful when working with large or infinite streams.

1. **anyMatch():**it checks the stream if it satisfies the given condition.
2. **findFirst():**it checks the element that matches a given condition and stops processing when it finds it.

***Note:****They are lazy, meaning they are not executed until a terminal operation is invoked.*

Later on from that processed filtered elements of objects, we are collecting the elements back to List using Collectors for which we have imported a specific package named ***java.util.stream***with the help of Collectors.toList() method. This is referred to as collect() operation in streams so here again we won’t be taking an example to discuss them out separately.

**Example:**

Java

*// Java program to illustrate Intermediate Operations*

*// in Streams*

*// Importing required classes*

**import** **java.io.\***;

**import** **java.util.\***;

**import** **java.util.stream.\***;

*// Main class*

**class** **Test** {

*// Main driver method*

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

{

*// Creating an integer Arraylist to store marks*

ArrayList<Integer> marks = **new** ArrayList<Integer>();

*// These are marks of the students*

*// Considering 5 students so input entries*

marks.add(30);

marks.add(78);

marks.add(26);

marks.add(96);

marks.add(79);

*// Printing the marks of the students before grace*

System.out.println(

"Marks of students before grace : " + marks);

*// Now we want to grace marks by 6*

*// using the streams to process over processing*

*// collection*

*// Using stream, we map every object and later*

*// collect to List*

*// and store them*

List<Integer> updatedMarks

= marks.stream()

.map(i -> i + 6)

.collect(Collectors.toList());

*// Printing the marks of the students after grace*

System.out.println(

"Marks of students after grace : "

+ updatedMarks);

}

}

**Output**

Marks of students before grace : [30, 78, 26, 96, 79]

Marks of students after grace : [36, 84, 32, 102, 85]

***Note:****For every object if there is urgency to do some operations be it square, double or any other than only we need to use map() function  operation else try to use filter() function operation.*

## Lazy Evaluation

Lazy Evaluation is the concept in Java Streams where computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. It is called lazy because intermediate operations are not evaluated unless a terminal operation is invoked.

Now geeks you are well aware of ‘why’ streams were introduced, but you should be wondering ‘where’ to use them. The answer is very simple as we do use them too often in our day-to-day life. Hence, the geek in simpler words we say directly lands p on wherever the concept of the collection is applicable, stream concept can be applied there.

## Java Stream: Real-life Examples

#### **Example 1:**

In general, daily world, whenever the data is fetched from the database, it is more likely we will be using collection so there stream concept is must apply to deal with processed data.

A diagram of a comparator sorted

Description automatically generated

Now we will be discussing real-time examples to interrelate streams in our life. Here we will be taking the most widely used namely as follows:

1. Streams in a Grocery store
2. Streams in mobile networking

### **1.**Streams in a Grocery store



The above pictorial image has been provided is implemented in streams which are as follows:

List<Integer> transactionsIds =   
 transactions.stream()  
 .filter(t -> t.getType() == Transaction.GROCERY)  
 .sorted(comparing(Transaction::getValue).reversed())  
 .map(Transaction::getId)  
 .collect(toList());

### **2.** Streams in mobile networking

Similarly, we can go for another widely used concept which is our dealing with our mobile numbers. Here we will not be proposing listings, simply will be demonstrating how the stream concept is invoked in mobile networking by various service providers across the globe.

*Collection can hold any number of object so let ‘mobileNumber’ be a collection and let it be holding various mobile numbers say it be holding 100+ numbers as objects. Suppose now the only carrier named ‘Airtel’ whom with which we are supposed to send a message if there is any migration between states in a country. So here streams concept is applied as if while dealing with all mobile numbers we will look out for this carrier using the filter() method operation of streams. In this way, we are able to deliver the messages without looking out for all mobile numbers and then delivering the message which senses impractical if done so as by now we are already too late to deliver. In this way these intermediate operations namely filter(), collect(), map() help out in the real world. Processing becomes super simpler which is the necessity of today’s digital world.*

Hope by now you the users come to realize the power of streams in Java as if we have to do the same task we do need to map corresponding to every object, increasing in code length, and decreasing the optimality of our code. With the usage of streams, we are able to in a single line irrespective of elements contained in the object as with the concept of streams we are dealing with the object itself.

***Note:****filter, sorted, and map, which can be connected together to form a pipeline.*

### What is a Pipeline?

A Stream Pipeline is a concept of chaining operations together Terminal Operations and Intermediate Operations. A Pipeline contains a stream source, which is further followed by zero or more intermediate operations, and a terminal operation.

## Java Stream Operations

### Method Types and Pipelines

Methods are of two types in Stream as mentioned below:

1. **Terminal Operations**
2. **Intermediate Operations**

## Terminal Operations

These are the operations that after consumed can’t further be used. There are few operations mentioned below:

### 1. **[forEach](https://www.geeksforgeeks.org/stream-foreach-method-java-examples)**

forEach performs an action for each element of the stream. Stream forEach is a terminal operation.

#### Syntax

void forEach(Consumer<? super T> action)

### 2. **[toArray](https://www.geeksforgeeks.org/stream-toarray-java-examples)**

Stream toArray() returns an array containing the elements of this stream. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used.

#### Syntax

Object[] toArray()

### 3. [**min**](https://www.geeksforgeeks.org/stream-min-method-in-java-with-examples) and [**max**](https://www.geeksforgeeks.org/stream-max-method-java-examples)

min and max return the min and max elements from the stream.

#### Syntax

Optional<**T**> min(Comparator<**?** super **T**> comparator)  
Optional<**T**> max(Comparator<**?** super **T**> comparator)

Where, Optional is a container object which may or may not contain a non-null value, and **T** is the type of object that may be compared by this comparator.

## Intermediate Operations

It returns a new stream that can be further processed. There are certain operations mentioned below:

### 1. [**filter**](https://www.geeksforgeeks.org/stream-filter-java-examples)

Stream filter returns a stream consisting of the elements of this stream that match the given predicate.

#### Syntax

Stream<T> filter(Predicate<? super T> predicate)

### 2. [**distinct**](https://www.geeksforgeeks.org/stream-distinct-java)

**distinct()** returns a stream consisting of distinct elements in a stream. distinct() is the method of **Stream** interface.

#### Syntax

Stream<T> distinct()

Where Stream is an interface and the function returns a stream consisting of distinct elements.

### 3. [**Sorted**](https://www.geeksforgeeks.org/stream-sorted-in-java)

Stream sorted() returns a stream consisting of the elements of this stream, sorted according to natural order. For ordered streams, the sort method is stable but for unordered streams, no stability is guaranteed.

#### **Syntax**

Stream<**T**> sorted()

where Stream is an interface and **T** is the type of stream elements.

### Comparison-Based Stream Operations

Comparison Based Stream Operations are the used for comparing, sorting, and ordering elements within a stream. There are certain examples of Comparison Based Stream Operations mentioned below:

1. Sorted
2. min and max
3. distinct

## Java Stream Specializations

As there are primitive data types or specializations like int, long and double. Similarly, streams have IntStream, LongStream, and DoubleStream. These are convenient for making performing transactions with numerical primitives.

### 1. Specialized Operations

Specialized streams provide additional operations as compared to the standard Stream – which are quite convenient when dealing with numbers.

### 2. Reduction Operations

Reduce Operation applies a binary operator, it takes a sequence of input elements and combines them to a single summary result. It is all done where first argument to the operator is the return value of the previous application and second argument is the current stream element.

## Parallel Streams

Parallel Streams are the type of streams that can perform operations concurrently on multiple threads. These Streams are meant to make use of multiple processors or cores available to speed us the processing speed. There are two methods to create parallel streams are mentioned below:

1. Using the parallel() method on a stream
2. Using parallelStream() on a Collection

To know more about Parallel Streams refer to [this link](https://www.geeksforgeeks.org/what-is-java-parallel-streams).

## Infinite Streams

Infinite Streams are the type of Streams that can produce unbounded(Infnite) number of elements. These Streams are useful when you need to work with the data sources that are not finite.

## Java Stream: File Operation

In this section, we see how to utilize Java stream in file I/O operation.

### 1. File Read Operation

Let’s understand file read operation through the given example

Java

*// Java Program to demonstrate*

*// File Read Operation*

**import** **java.io.IOException**;

**import** **java.nio.file.Files**;

**import** **java.nio.file.Paths**;

**import** **java.util.List**;

**import** **java.util.stream.Collectors**;

**import** **java.util.stream.Stream**;

**class** **GFG** {

*// Method to filter strings of a given length and*

*// convert them to uppercase*

**private** **static** List<String>

filterAndConvertToUpper(Stream<String> stream,

int length)

{

**return** stream.filter(s -> s.length() == length)

.map(String::toUpperCase)

.collect(Collectors.toList());

}

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

{

*// Replace with the*

*// actual file path*

String fileName = "path/to/your/file.txt";

*// Step 1: Create a Stream of lines from the*

*// file*

**try** (Stream<String> lines

= Files.lines(Paths.get(fileName))) {

List<String> filteredStrings

= filterAndConvertToUpper(lines, 5);

System.out.println(

"Filtered strings with length 5 (converted to uppercase): "

+ filteredStrings);

}

**catch** (IOException e) {

e.printStackTrace();

}

}

}

#### Input:

Geeks  
gfg  
geeks  
geeksforgeeks  
Coder  
Guys

#### Output:

Filtered strings with length 5 (converted to uppercase): [GEEKS, GEEKS, CODER]

### 2. File Write Operation

Let’s understand file write operation through the given example

Java

*// Java Program to demonstrate*

*// File Write Operation*

**import** **java.io.\***;

**import** **java.nio.file.Files**;

**import** **java.nio.file.Paths**;

**import** **java.util.stream.Stream**;

*// Driver Class*

**class** **GFG** {

*// main function*

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

{

String[] words

= { "Geeks", "for", "Geeks", "Hello", "World" };

*// Replace with the*

*// actual file path*

String fileName = "path/to/your/file.txt";

*// Step 1: Create a PrintWriter to write to the*

*// file*

**try** (PrintWriter pw

= **new** PrintWriter(Files.newBufferedWriter(

Paths.get(fileName)))) {

*// Step 2: Use Stream to write each word to the*

*// file*

Stream.of(words).forEach(pw::println);

*// Step 3: Print success message to the console*

System.out.println(

"Words written to the file successfully.");

}

**catch** (IOException e) {

*// Step 4: Handle any IO exception that occurs*

*// during the file writing process*

e.printStackTrace();

}

}

}

#### Output:

Words written to the file successfully.

## Java Streams Improvements in Java 9

*As we know, Java 8 introduced a Java stream to the world which provided a powerful way to process collections of data. However, the following version of the language also contributed to the feature. Thereafter, some improvements and features were added to the Java stream in JAVA 9. In this section, we will provide an overview of the advancements introduced by Java 9 to the Streams API like takeWhile, dropWhile,*iterate, ofNullable, etc.

## Conclusion

Java Streams offer a powerful way to handle data in Java. They allow developers to write more readable and concise code for processing collections. By using Java Streams, you can easily filter, map, and reduce data with simple and expressive methods. This not only makes your code easier to maintain but also helps improve performance by taking advantage of parallel processing. If you’re looking to make your data operations more efficient and straightforward, learning Java Streams is definitely worth the effort.

## Java 8 Stream – FAQs

### 1. How to learn streams in Java 8?

*To learn java streams effectivily you need to get you theory concepets as well as practical skills strong. for that first you need to make your concepts strong then make it in pratice.*

1. ***Grasp the fundamental concept of streams.***
   * *What are streams?*
   * *Why are they useful?*
   * *How do they work?*
2. ***Familiarize yourself with creating streams from diverse sources.***
   * *How can you create a stream from an array, a collection, or a file?*
3. ***Dive into intermediate operations.***
   * *What are intermediate operations?*
   * *How do they transform data?*
4. ***Understand terminal operations.***
   * *What are terminal operations?*
   * *How do they produce final results or trigger actions on the stream?*
5. ***Embrace the concept of lazy evaluation.***
   * *How does lazy evaluation work? How does it optimize resource usage?*
6. ***Practice chaining operations to create intricate data pipelines.***
   * *How can you chain multiple operations together to create complex data pipelines?*
7. ***Explore parallel processing.***
   * *How can you use parallel processing to improve the performance of your streams?*
8. ***Apply streams to real-world scenarios.***
   * *Find situations where streams can simplify code and improve efficiency.*
9. ***Utilize online resources, tutorials, and exercises.***
10. ***Consider in-depth tutorials.****If you want a comprehensive learning experience, consider taking an in-depth tutorial on streams.*

### 2. Why we use stream in Java 8?

*Java streams offer a range of functionalities that significantly enhance their importance.*

1. ***Efficient Data Processing:****They don’t store the data themselves, but instead act as a way to process data from various*[*DS*](https://www.geeksforgeeks.org/data-structures)*.*
2. ***Functional and Non-Destructive****: Streams follow a functional programming approach.*
3. ***Lazy Evaluation****: which means they perform computations only when needed*
4. ***Single Pass Processing****: The elements in a stream are processed only once during its lifetime*

### 3. What is the best practice of Java streams?

*Java Stream API is a powerful and flexible tool that can significantly simplify code for data processing tasks and here are some best pratices for using java streams.*

1. *Use primitive streams for better performance*
2. *Avoid nesting streams*
3. *use stream with immutable objects*
4. *Use filter() before map() to avoid unnecessary processing*
5. *Prefer method references over lambda expressions*
6. *Use distinct() to remove duplicates*
7. *Use sorted() with caution*
8. *Use lazy evaluation for better performance*

## 10 Ways to Create a Stream in Java

The [Stream API](https://www.geeksforgeeks.org/stream-in-java/), introduced in Java 8, it is used to process collections of objects. Stream is a sequence of objects, that supports many different methods which can be pipe lined to produce the desired result.

The features of Java stream are –

* A stream is not a data structure alternatively it takes input from the Collections, Arrays or I/O channels.
* A Streams does not change the original data structure, they only provide the result as the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipe lined. Terminal operation mark the end of the stream and return the result.

## **Different way to create Streams:**

1. **Using**[Collection](https://www.geeksforgeeks.org/collections-in-java-2/)

**Approach:**

* 1. Get the collection
  2. Construct a Sequential Stream from the collection using Collection.stream() method
  3. Print the Stream

Below is the implementation of the above approach:

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(List<T> list)      {            // Create stream object with the List          Stream<T> stream = list.stream();            // Iterate list first to last element          Iterator<T> it = stream.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Create ArrayList of String          List<String> list = **new** ArrayList<>();            // Add element in list          list.add("Geeks");          list.add("for");          list.add("Geeks");            // Get the Stream from the List          getStream(list);      }  } |

**Output:**

Geeks for Geeks

1. **Create a stream from specified values**

[Stream.of(T…t)](https://www.geeksforgeeks.org/stream-oft-values-java-examples/) method can be used to create a stream with the specified t values, where t are the elements. This method returns a sequential Stream containing the t elements.

Below is the implementation of the above approach:

**Program:**

|  |
| --- |
| // Java program to create Stream from values    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** **void** getStream()      {            // Create a stream from specified values          Stream<Integer> stream              = Stream.of(1, 2,                          3, 4,                          5, 6,                          7, 8,                          9);            // Displaying the sequential ordered stream          stream.forEach(p -> System.out.print(p + " "));      }    **public** **static** **void** main(String[] args)      {            // Get the Stream from the values          getStream();      }  } |

**Output:**

1 2 3 4 5 6 7 8 9

1. **Create stream from an array:**

The [Stream.of() and Arrays.stream()](https://www.geeksforgeeks.org/difference-between-stream-of-and-arrays-stream-method-in-java/) are two commonly used methods for creating a sequential stream from a specified array. Both these methods returns a Stream when called with a non-primitive type T.  
Integer array

* 1. **Create stream using**[Arrays.stream()](https://www.geeksforgeeks.org/arrays-stream-method-in-java/)

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(T[] arr)      {            // Create stream from an array          // using Arrays.stream()          Stream<T> streamOfArray              = Arrays.stream(arr);            // Iterate list first to last element          Iterator<T> it              = streamOfArray.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the array          String[] arr              = **new** String[] { "a", "b", "c" };            // Get the Stream from the Array          getStream(arr);      }  } |

**Output:**

a b c

* 1. **Create stream using**[Stream.of()](https://www.geeksforgeeks.org/stream-oft-t-java-examples/)  
     A non interfering action to be perform on elements as they are consumed from the stream and returns also a new stream.

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(T[] arr)      {            // Create stream from an array          // using Stream.of()          Stream<T> streamOfArray = Stream.of(arr);            // Iterate list first to last element          Iterator<T> it = streamOfArray.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the array          String[] arr              = **new** String[] { "a", "b", "c" };            // Get the Stream from the Array          getStream(arr);      }  } |

**Output:**

a b c

1. **Create an empty stream using**[Stream.empty()](https://www.geeksforgeeks.org/stream-empty-java-examples/)

The empty() method is used upon creation to avoid returning null for streams with no element.

**Program:**

|  |
| --- |
| // Java program to create empty Stream    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** **void** getStream()      {            // Create stream from an array using Stream.empty()          Stream<String> streamOfArray              = Stream.empty();            // Iterate list first to last element          Iterator<String> it              = streamOfArray.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the empty Stream          getStream();      }  } |

**Output:**

1. **Create a Stream using**[Stream.builder()](https://www.geeksforgeeks.org/stream-builder-java-examples/)

The builder() method is used when the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream.

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream()      {            // Create stream using Stream builder()          Stream.Builder<String> builder              = Stream.builder();            // Adding elements in the stream of Strings          Stream<String> stream = builder.add("a")                                      .add("b")                                      .add("c")                                      .build();            // Iterate list first to last element          Iterator<String> it = stream.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the Stream using Builder          getStream();      }  } |

**Output:**

a b c

1. **Create an infinite Stream using Stream.iterate()**

The iterate() method returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed. In below example, First element of the resulting stream is a first parameter of the iterate method. For creating every following element the function is applied to the previous element. In the example below the second element will be 4.

**Program:**

|  |
| --- |
| // Java program to create infinite Stream  // using Stream.iterate() method    **import** java.util.\*;  **import** java.util.stream.Stream;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void**      getStream(**int** seedValue, **int** limitTerms)      {            // Create infinite stream          // using Stream.iterate() method          Stream.iterate(seedValue,                         (Integer n) -> n \* n)              .limit(limitTerms)              .forEach(System.out::println);      }    **public** **static** **void** main(String[] args)      {            // Get the seed value  **int** seedValue = 2;            // Get the limit for number of terms  **int** limitTerms = 5;            // Get the Stream from the function          getStream(seedValue, limitTerms);      }  } |

**Output:**

2

4

16

256

65536

1. **Create an infinite Stream using**[Stream.generate()](https://www.geeksforgeeks.org/stream-generate-method-java/)**method**

The generate() method accepts a Supplier for generating elements and the resulting stream is infinite. So to restrict it, specify the desired size or the generate() method will work until it reaches the memory limit.

**Program:**

|  |
| --- |
| // Java program to create infinite Stream  // using Stream.generate() method    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(**int** limitTerms)      {            // Create infinite stream          // using Stream.generate() method          Stream.generate(Math::random)              .limit(limitTerms)              .forEach(System.out::println);      }    **public** **static** **void** main(String[] args)      {            // Get the limit for number of terms  **int** limitTerms = 5;            // Get the Stream from the function          getStream(limitTerms);      }  } |

**Output:**

0.2293502475696314

0.5650334795948209

0.3418138293253522

0.36831074763500116

0.4864408670097241

1. **Create stream from a**[Pattern](https://www.geeksforgeeks.org/regular-expressions-in-java/)**using**[Predicate](https://www.geeksforgeeks.org/java-8-predicate-with-examples/)

In java 8, the Predicate asPredicate() method of Pattern creates a predicate boolean-valued function that is used for pattern matching.

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.\*;  **import** java.util.regex.Pattern;    **class** GFG {        // Function convert a List into Stream  **private** **static** **void**      getStream(List<String> list, Pattern p)      {            list.stream()              .filter(p.asPredicate())              .forEach(System.out::println);      }    **public** **static** **void** main(String[] args)      {            // Create ArrayList of String          // that is backed by the specified array          List<String> list              = Arrays                    .asList("Geeks",                            "For",                            "Geek",                            "GeeksForGeeks",                            "A Computer Portal");            // Get the pattern          Pattern p = Pattern.compile("^G");            // Get the Stream from the List matching Pattern          getStream(list, p);      }  } |

**Output:**

Geeks

Geek

GeeksForGeeks

1. **Create stream from**[Iterator](https://www.geeksforgeeks.org/iterators-in-java/)

Iterators, in Java, are used in Collection Framework to retrieve elements one by one. Spliterator is the key to create the sequential stream. Hence in this method also, Spliterator is used. But in this method, the source of Spliterator is set to an Iterable created from the Iterator. So first the Iterable is created from the Iterator. Then the Spliterator is passed to the stream() method directly as Iterable.spliterator().

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(Iterator<T> itr)      {            // Convert the iterator into a Spliterator          Spliterator<T> spitr              = Spliterators                    .spliteratorUnknownSize(itr,                                            Spliterator.NONNULL);            // Convert spliterator into a sequential stream          Stream<T> stream              = StreamSupport.stream(spitr, **false**);            // Iterate list first to last element          Iterator<T> it = stream.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the Iterator          Iterator<String> iterator = Arrays                                          .asList("a", "b", "c")                                          .iterator();            // Get the Stream from the Iterator          getStream(iterator);      }  } |

**Output:**

a b c

1. **Create stream from**[Iterable](https://www.geeksforgeeks.org/tag/java-iterable/)

Iterable interface is designed keeping in mind and does not provide any stream() method on its own. Simply it can be passed into StreamSupport.stream() method, and get a Stream from the given Iterable object. It is easier to turn an Iterable into a Stream. Iterable has a default method spliterator(), which can be used to get a Spliterator instance, which can be in turn then converted to a Stream.

**Note:** The Iterable is not a instance of Collection, this method internally calls StreamSupport.stream() to get a sequential Stream from Spliterator else it simply calls Collection.stream() method.

**Program:**

|  |
| --- |
| // Java program to create Stream from Collections    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Function convert a List into Stream  **private** **static** <T> **void** getStream(Iterable<T> iterable)      {            // Convert the iterator into a Stream          Stream<T> stream              = StreamSupport                    .stream(iterable.spliterator(),  **false**);            // Iterate list first to last element          Iterator<T> it = stream.iterator();            // Iterate stream object  **while** (it.hasNext()) {              System.out.print(it.next() + " ");          }      }    **public** **static** **void** main(String[] args)      {            // Get the Iterable          Iterable<String> iterable              = Arrays.asList("a", "b", "c");            // Get the Stream from the Iterable          getStream(iterable);      }  } |

**Output:**

a b c

# Program to convert a Map to a Stream in Java

A [Stream](https://www.geeksforgeeks.org/stream-in-java/) is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

Below are various method to convert Map to Stream in Java:

1. **Converting complete Map<Key, Value> into Stream**: This can be done with the help of Map.entrySet() method which returns a Set view of the mappings contained in this map. In Java 8, this returned set can be easily converted into a Stream of key-value pairs using Set.stream() method.

**Algorithm**:

* 1. Get the Map<Key, Value>.
  2. Convert Map<Key, Value> into Set<Key> using Map.entrySet() method.
  3. Convert the obtained Set into Stream using Set.stream()
  4. Return/Print the Stream of Map.

**Program:**

|  |
| --- |
| // Java Program to convert  // Map<Key, Value> into Stream    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Generic function to convert List of      // String to List of Integer  **public** **static** <K, V> Stream<Map.Entry<K, V> >      convertMapToStream(Map<K, V> map)      {            // Return the obtained Stream  **return** map                // Convert the Map to Set              .entrySet()                // Convert the Set to Stream              .stream();      }    **public** **static** **void** main(String args[])      {            // Create a Map          Map<Integer, String> map = **new** HashMap<>();            // Add entries to the Map          map.put(1, "Geeks");          map.put(2, "forGeeks");          map.put(3, "A computer Portal");            // Print the Map          System.out.println("Map: " + map);            // Convert the Map to Stream          Stream<Map.Entry<Integer, String> > stream =                                     convertMapToStream(map);            // Print the TreeMap          System.out.println("Stream: "                        + Arrays.toString(stream.toArray()));      }  } |

**Output:**

Map: {1=Geeks, 2=forGeeks, 3=A computer Portal}

Stream: [1=Geeks, 2=forGeeks, 3=A computer Portal]

1. **Converting only the Keyset of the Map<Key, Value> into Stream**: This can be done with the help of Map.keySet() method which returns a Set view of the keys contained in this map. In Java 8, this returned set can be easily converted into a Stream of key-value pairs using Set.stream() method.

**Algorithm**:

* 1. Get the Map<Key, Value>.
  2. Convert Map<Key, Value> into Set<Key> using Map.keySet() method.
  3. Convert the obtained Set into Stream using Set.stream()
  4. Return/Print the Stream of Map.

**Program:**

|  |
| --- |
| // Java Program to convert  // Map<Key, Value> into Stream    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Generic function to convert List of      // String to List of Integer  **public** **static** <K, V> Stream<K>      convertMapToStream(Map<K, V> map)      {            // Return the obtained Stream  **return** map                // Convert the Map to Set<Key>              .keySet()                // Convert the Set to Stream              .stream();      }    **public** **static** **void** main(String args[])      {            // Create a Map          Map<Integer, String> map = **new** HashMap<>();            // Add entries to the Map          map.put(1, "Geeks");          map.put(2, "forGeeks");          map.put(3, "A computer Portal");            // Print the Map          System.out.println("Map: " + map);            // Convert the Map to Stream          Stream<Integer> stream = convertMapToStream(map);            // Print the TreeMap          System.out.println("Stream: "                      + Arrays.toString(stream.toArray()));      }  } |

**Output:**

Map: {1=Geeks, 2=forGeeks, 3=A computer Portal}

Stream: [1, 2, 3]

1. **Converting only the Value of the Map<Key, Value> into Stream**: This can be done with the help of Map.values() method which returns a Set view of the values contained in this map. In Java 8, this returned set can be easily converted into a Stream of key-value pairs using Set.stream() method.

**Algorithm**:

* 1. Get the Map<Key, Value>.
  2. Convert Map<Key, Value> into Set<Value> using Map.values() method.
  3. Convert the obtained Set into Stream using Set.stream()
  4. Return/Print the Stream of Map.

**Program:**

|  |
| --- |
| // Java Program to convert  // Map<Key, Value> into Stream    **import** java.util.\*;  **import** java.util.stream.\*;    **class** GFG {        // Generic function to convert List of      // String to List of Integer  **public** **static** <K, V> Stream<V>      convertMapToStream(Map<K, V> map)      {            // Return the obtained Stream  **return** map                // Convert the Map to Set<Value>              .values()                // Convert the Set to Stream              .stream();      }    **public** **static** **void** main(String args[])      {            // Create a Map          Map<Integer, String> map = **new** HashMap<>();            // Add entries to the Map          map.put(1, "Geeks");          map.put(2, "forGeeks");          map.put(3, "A computer Portal");            // Print the Map          System.out.println("Map: " + map);            // Convert the Map to Stream          Stream<String> stream = convertMapToStream(map);            // Print the TreeMap          System.out.println("Stream: "                       + Arrays.toString(stream.toArray()));      }  } |

**Output:**

Map: {1=Geeks, 2=forGeeks, 3=A computer Portal}

Stream: [Geeks, forGeeks, A computer Portal]

# JDBC

## Introduction to JDBC (Java Database Connectivity)

**JDBC**stands for **Java Database Connectivity. JDBC**is a **Java API** to connect and execute the query with the database. It is a specification from Sun Microsystems that provides a standard abstraction(API or Protocol) for Java applications to communicate with various databases. It provides the language with Java database connectivity standards. It is used to write programs required to access databases. JDBC, along with the database driver, can access databases and spreadsheets. The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

## **Definition of JDBC(Java Database Connectivity)**

JDBC is an API(Application programming interface) used in Java programming to interact with databases. The [classes](https://www.geeksforgeeks.org/classes-objects-java) and [interfaces](https://www.geeksforgeeks.org/interfaces-in-java) of JDBC allow the application to send requests made by users to the specified database. The current version of JDBC is JDBC 4.3, released on 21st September 2017.

### **Purpose of JDBC**

Enterprise applications created using the JAVA EE technology need to interact with databases to store application-specific information. So, interacting with a database requires efficient database connectivity, which can be achieved by using the [ODBC](https://www.geeksforgeeks.org/difference-odbc-jdbc)(Open database connectivity) driver. This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql, and SQL server database.

### **Components of JDBC**

There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

**1. JDBC API:** It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA(write once run anywhere) capabilities. The **java.sql**package contains interfaces and classes of JDBC API.

java.sql: This package provides APIs for data access and data process in a relational database, included in   
 Java Standard Edition (java SE)  
javax.sql: This package extends the functionality of java package by providing datasource interface for   
 establishing connection pooling, statement pooling with a data source, included in   
 Java Enterprise Edition (java EE)

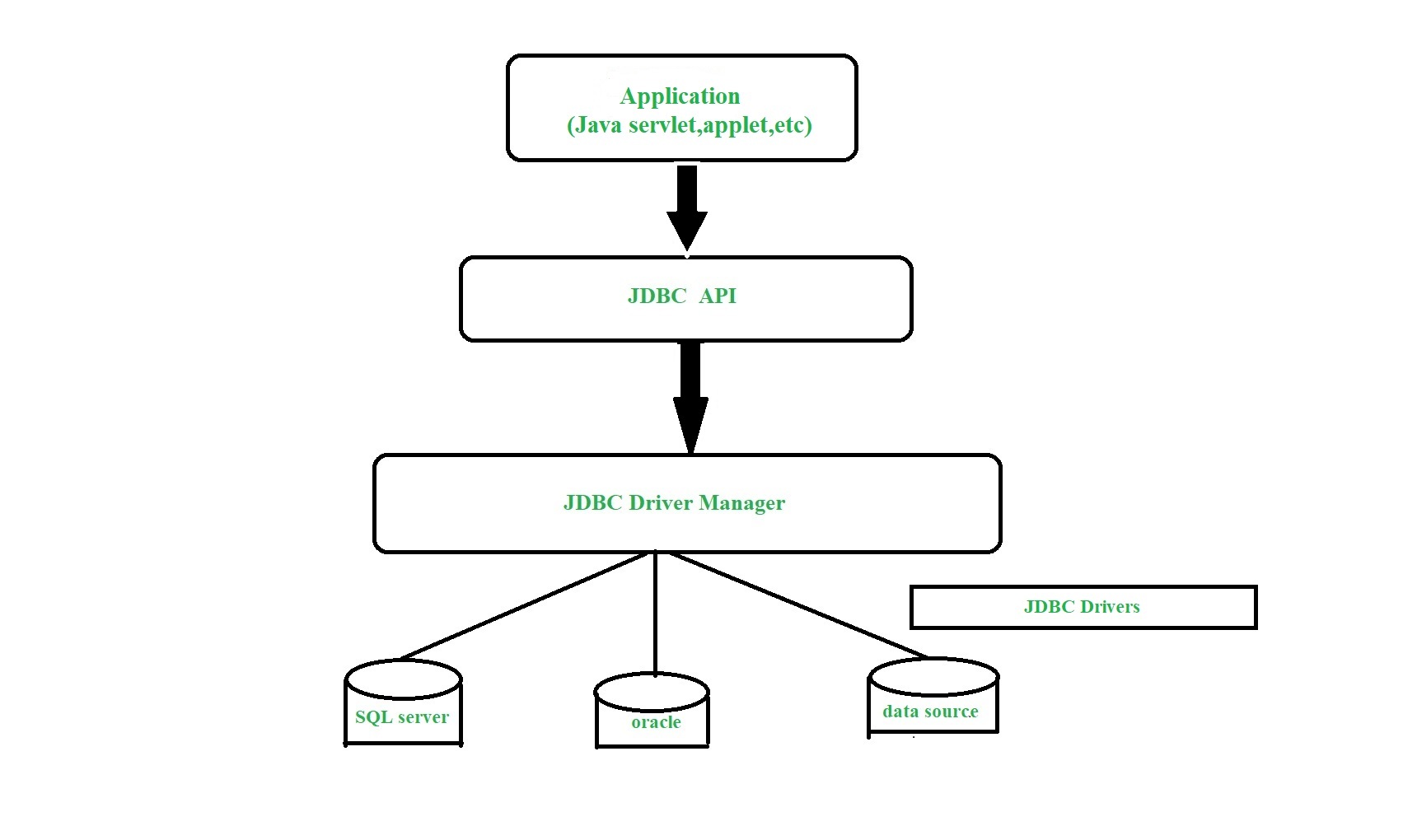
It also provides a standard to connect a database to a client application.

**2.**[**JDBC Driver manager**](https://www.geeksforgeeks.org/jdbc-drivers)**:**It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.

**3. JDBC Test suite:** It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.

**4. JDBC-ODBC Bridge Drivers**: It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the **sun.jdbc.odbc**package which includes a native library to access ODBC characteristics.

### Architecture of JDBC



**Description:**

1. **Application:** It is a java applet or a servlet that communicates with a data source.
2. **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important interfaces defined in JDBC API are as follows: Driver interface , ResultSet Interface , RowSet Interface , PreparedStatement interface, Connection inteface, and cClasses defined in JDBC API are as follows: DriverManager class, Types class, Blob class, clob class.
3. **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

### Types of JDBC Architecture(2-tier and 3-tier)

The JDBC architecture consists of[two-tier and three-tier processing models](https://www.geeksforgeeks.org/dbms-architecture-2-level-3-level) to access a database. They are as described below:

1. **Two-tier model:** A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.   
   The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the user’s machine acts as a client, and the machine has the data source running acts as the server.
2. **Three-tier model:** In this, the user’s queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.   
   This type of model is found very useful by management information system directors.

### What is API?

Before jumping into JDBC Drivers, let us know more about API.

API stands for **Application Programming Interface**. It is essentially a set of rules and protocols which transfers data between different software applications and allow different software applications to communicate with each other. Through an API one application can request information or perform a function from another application without having direct access to it’s underlying code or the application data.

JDBC API uses JDBC Drivers to connect with the database.

## **JDBC Drivers**

[JDBC drivers](https://www.geeksforgeeks.org/jdbc-drivers) are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver (partially java driver)
3. Type-3 driver or Network Protocol driver (fully java driver)
4. Type-4 driver or Thin driver (fully java driver)

### **Interfaces of JDBC API**

A list of popular interfaces of JDBC API is given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface
* DatabaseMetaData interface
* RowSet interface

### **Classes of JDBC API**

A list of popular classes of JDBC API is given below:

* DriverManager class
* Blob class
* Clob class
* Types class

## Types of Statements in JDBC

The **Statement**interface in JDBC is used to create SQL statements in Java and execute queries with the database. There are different types of statements used in JDBC:

* Create Statement
* Prepared Statement
* Callable Statement

### **1.**  **Create a Statement:**

A Statement object is used for general-purpose access to databases and is useful for executing static SQL statements at runtime.

**Syntax:**

Statement statement = connection.createStatement();

**Implementation:**Once the Statement object is created, there are three ways to execute it.

* ***boolean execute(String SQL):*** If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute [SQL DDL](https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/) statements or for dynamic SQL.
* **int executeUpdate(String SQL):** Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.
* ***ResultSet executeQuery(String SQL):*** Returns a ResultSet object. Used similarly as SELECT is used in SQL.

**Example:**

Java

*// Java Program illustrating Create Statement in JDBC*

*// Importing Database(SQL) classes*

**import** **java.sql.\***;

*// Class*

**class** **GFG** {

*// Main driver method*

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

{

*// Try block to check if any exceptions occur*

**try** {

*// Step 2: Loading and registering drivers*

*// Loading driver using forName() method*

Class.forName("com.mysql.cj.jdbc.Driver");

*// Registering driver using DriverManager*

Connection con = DriverManager.getConnection(

"jdbc:mysql:///world", "root", "12345");

*// Step 3: Create a statement*

Statement statement = con.createStatement();

String sql = "select \* from people";

*// Step 4: Execute the query*

ResultSet result = statement.executeQuery(sql);

*// Step 5: Process the results*

*// Condition check using hasNext() method which*

*// holds true till there is single element*

*// remaining in List*

**while** (result.next()) {

*// Print name an age*

System.out.println(

"Name: " + result.getString("name"));

System.out.println(

"Age:" + result.getString("age"));

}

}

*// Catching database exceptions if any*

**catch** (SQLException e) {

*// Print the exception*

System.out.println(e);

}

*// Catching generic ClassNotFoundException if any*

**catch** (ClassNotFoundException e) {

*// Print and display the line number*

*// where exception occurred*

e.printStackTrace();

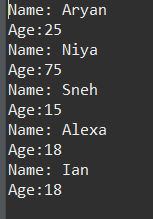
}

}

}

#### **Output:**

Name and age are as shown for random inputs.



### **2. Prepared Statement:**

A PreparedStatement represents a precompiled SQL statement that can be executed multiple times. It accepts parameterized SQL queries, with ? as placeholders for parameters, which can be set dynamically.

**Illustration:**

Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

INSERT INTO people VALUES ("Ayan",25);  
INSERT INTO people VALUES("Kriya",32);

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

String query = "INSERT INTO people(name, age)VALUES(?, ?)";  
PreparedStatement pstmt = con.prepareStatement(query);  
pstmt.setString(1,"Ayan");  
ptstmt.setInt(2,25);  
// where pstmt is an object name

**Implementation:**Once the PreparedStatement object is created, there are three ways to execute it:

* ***execute():*** This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.
* [***executeQuery()***](https://www.geeksforgeeks.org/establishing-jdbc-connection-in-java/)***:*** Returns a ResultSet from the current prepared statement.
* [***executeUpdate()***](https://www.geeksforgeeks.org/how-to-insert-records-to-a-table-using-jdbc-connection/)***:*** Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in thecurrent Prepared Statement.

**Example:**

Java

*// Java Program illustrating Prepared Statement in JDBC*

*// Step 1: Importing DB(SQL here) classes*

**import** **java.sql.\***;

*// Importing Scanner class to*

*// take input from the user*

**import** **java.util.Scanner**;

*// Main class*

**class** **GFG** {

*// Main driver method*

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

{

*// try block to check for exceptions*

**try** {

*// Step 2: Establish a connection*

*// Step 3: Load and register drivers*

*// Loading drivers using forName() method*

Class.forName("com.mysql.cj.jdbc.Driver");

*// Scanner class to take input from user*

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

*// Display message for ease for user*

System.out.println(

"What age do you want to search?? ");

*// Reading age an primitive datatype from user*

*// using nextInt() method*

int age = sc.nextInt();

*// Registering drivers using DriverManager*

Connection con = DriverManager.getConnection(

"jdbc:mysql:///world", "root", "12345");

*// Step 4: Create a statement*

PreparedStatement ps = con.prepareStatement(

"select name from world.people where age = ?");

*// Step 5: Execute the query*

ps.setInt(1, age);

ResultSet result = ps.executeQuery();

*// Step 6: Process the results*

*// Condition check using next() method*

*// to check for element*

**while** (result.next()) {

*// Print and display elements(Names)*

System.out.println("Name : "

+ result.getString(1));

}

*// Step 7: Closing the connections*

*// (Optional but it is recommended to do so)*

}

*// Catch block to handle database exceptions*

**catch** (SQLException e) {

*// Display the DB exception if any*

System.out.println(e);

}

*// Catch block to handle class exceptions*

**catch** (ClassNotFoundException e) {

*// Print the line number where exception occurred*

*// using printStackTrace() method if any*

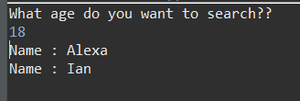
e.printStackTrace();

}

}

}

#### **Output:**



### **3. Callable Statement:**

A CallableStatement is used to execute stored procedures in the database. Stored procedures are precompiled SQL statements that can be called with parameters. They are useful for executing complex operations that involve multiple SQL statements.

**Syntax:** To create a CallableStatement,

CallableStatement cstmt = con.prepareCall("{call ProcedureName(?, ?)}");

* {call ProcedureName(?, ?)}: Calls a stored procedure named ProcedureName with placeholders ? for input parameters.

**Methods to Execute:**

* **execute():** Executes the stored procedure and returns a boolean indicating whether the result is a ResultSet (true) or an update count (false).
* **executeQuery():** Executes a stored procedure that returns a ResultSet.
* **executeUpdate():** Executes a stored procedure that performs an update and returns the number of rows affected.

**Example:**

Java

*// Java Program illustrating Callable Statement in JDBC*

*// Importing DB(SQL) classes*

**import** **java.sql.\***;

**public** **class** **GFG** {

*// Main driver method*

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

*// Try block to check if any exceptions occur*

**try** {

*// Step 1: Load and register the driver*

Class.forName("com.mysql.cj.jdbc.Driver");

*// Step 2: Establish a connection*

Connection con = DriverManager.getConnection("jdbc:mysql:///world", "root", "12345");

*// Step 3: Create a CallableStatement*

CallableStatement cs = con.prepareCall("{call GetPeopleInfo()}");

*// Step 4: Execute the stored procedure*

ResultSet result = cs.executeQuery();

*// Step 5: Process the results*

**while** (result.next()) {

*// Print and display elements (Name and Age)*

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

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

}

*// Step 6: Close resources*

result.close();

cs.close();

con.close();

}

*// Catch block for SQL exceptions*

**catch** (SQLException e) {

e.printStackTrace();

}

*// Catch block for ClassNotFoundException*

**catch** (ClassNotFoundException e) {

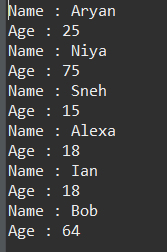
e.printStackTrace();

}

}

}

#### **Output:**



#### Explanation of the Program:

* This Java code demonstrates how to use a CallableStatement in JDBC to execute a stored procedure.
* It connects to a MySQL database and prepares a CallableStatement to call a stored procedure named peopleinfo with two parameters.
* After executing the procedure, it runs a SELECT query to retrieve and display all records from the people table.
* Exception handling is included to manage potential SQL and class loading errors.