Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

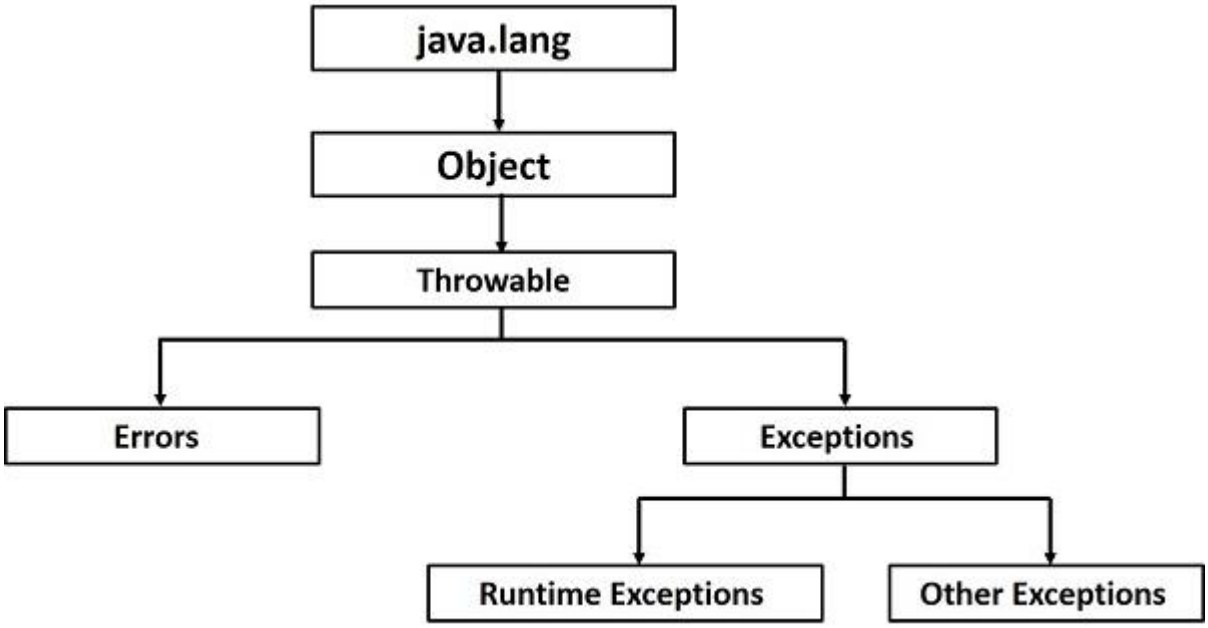
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.

For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example ,ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable.

Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
1. public class JavaExceptionExample{
2.   public static void main(String args[]){
3.    try{
4.       //code that may raise exception
5.       int data=100/0;
6.    }catch(ArithmeticException e){System.out.println(e);}
7.    //rest code of the program
```

```
8.     System.out.println("rest of the code...");
9.    }
10.}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class

exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Java Catch Multiple Exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Syntax

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

Example 1

Let's see a simple example of java multi-catch block.

```
1. public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
```

```
5.          try{
6.              int a[]=new int[5];
7.
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.   }
24.}
```

Test it Now

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

throw Instance i.e.,

```
1. throw new exception_class("error message");
```

```
1. public class TestThrow1 {
2.     //function to check if person is eligible to vote or not
3.     public static void validate(int age) {
4.         if(age<18) {
5.             //throw Arithmetic exception if not eligible to vote
6.             throw new ArithmeticException("Person is not eligible to vote");
7.         }
8.         else {
9.             System.out.println("Person is eligible to vote!!");
10.        }
11.    }
12.    //main method
13.    public static void main(String args[]){
14.        //calling the function
15.        validate(13);
16.        System.out.println("rest of the code...");
17.  }
18.}
```

Java throws keyword

The **Java throws keyword** is used to declare an exception.

It gives an information to the programmer that there may occur an exception.

So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions.

If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
```

```
3. }
4. public class TestThrows {
5.     //defining a method
6.     public static int divideNum(int m, int n) throws ArithmeticException
       {
7.         int div = m / n;
8.         return div;
9.     }
10.    //main method
11.    public static void main(String[] args) {
12.        TestThrows obj = new TestThrows();
13.        try {
14.            System.out.println(obj.divideNum(45, 0));
15.        }
16.        catch (ArithmeticException e){
17.            System.out.println("\nNumber cannot be divided by 0");
18.        }
19.
20.        System.out.println("Rest of the code..");
21.    }
22.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Example

```
public class ExcepTest {

  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Throwable class.

Exceptions Methods

Following is the list of important methods available in the Throwable class.

Commonly used exception methods of Throwable class

- **public String getMessage():** returns the message string about the exception.
- **public Throwable getCause():** returns the cause of the exception. It will return null if the cause is unknown or nonexistent.
- **public String toString():** returns a short description of the exception.
- **public void printStackTrace(PrintStream s):** prints the short description of the exception (using toString()) + a stack trace for this exception on the error output stream(System.err).

```
class ArithmaticTest {
  public void division(int num1, int num2) {
    try {

      System.out.println(num1/num2);

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

      System.out.println("getCause(): " + e.getCause());

      System.out.println("toString(): " + e.toString());
      System.out.println("printStackTrace(): ");
      e.printStackTrace();
    }
  }
}
public class Test {
  public static void main(String args[]) {

    ArithmaticTest test = new ArithmaticTest();
        test.division(20, 0);
  }
}
```

- Output
- getMessage(): / by zero
- getCause(): null
- toString(): java.lang.ArithmeticException: / by zero
- printStackTrace():
- java.lang.ArithmeticException: / by zero
- at ArithmaticTest.division(Test.java:5)
- at Test.main(Test.java:27)

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below −

```
class MyException extends Exception {
}
```

- You just need to extend the predefined **Exception** class to create your own Exception.
- These are considered to be checked exceptions.
- An exception class is like any other class,containing useful fields and methods.

TestCustomException1.java

```
1. // class representing custom exception
2. class InvalidAgeException  extends Exception
3. {
4.     public InvalidAgeException (String str)
```

```
5.      {
6.          // calling the constructor of parent Exception
7.          super(str);
8.      }
9.  }
10.
11.// class that uses custom exception InvalidAgeException
12.public class TestCustomException1
13.{
14.
15.    // method to check the age
16.    static void validate (int age) throws InvalidAgeException{
17.       if(age < 18){
18.
19.        // throw an object of user defined exception
20.        throw new InvalidAgeException("age is not valid to vote");
21.    }
22.       else {
23.        System.out.println("welcome to vote");
24.        }
25.     }
26.
27.    // main method
28.    public static void main(String args[])
29.    {
30.        try
31.        {
32.            // calling the method
33.            validate(13);
34.        }
35.        catch (InvalidAgeException ex)
36.        {
37.            System.out.println("Caught the exception");
38.
39.            // printing the message from InvalidAgeException object
40.            System.out.println("Exception occured: " + ex);
41.        }
```

```
42.
43.         System.out.println("rest of the code...");
44.     }
45.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

File Handling:

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.

In Java, the concept Stream is used in order to perform I/O operations on a file.

Stream Classes in Java

Streams in Java represent an ordered sequence of data. Java performs input and output operations in the terms of streams.
Modern versions of Java platform define two types of I/O streams:

- Byte streams
- Character streams

Let's understand first the meaning of byte streams and character streams one by one.

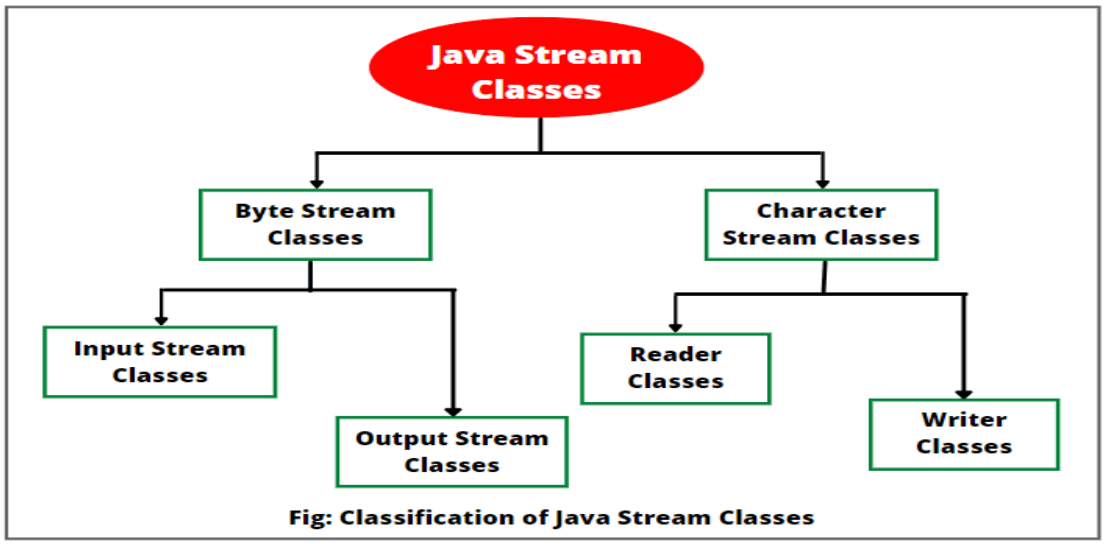


Fig: Classification of Java Stream Classes

Byte Streams in Java

- **Byte streams in Java** are designed to provide a convenient way for handling the input and output of bytes (i.e., units of 8-bits data).
- We use them for reading or writing to binary data I/O.
- Byte streams that are used for reading are called **input streams** and for writing are called **output streams**.
- They are represented by the abstract classes of InputStream and OutputStream in Java.

There are two kinds of byte stream classes in Java. They are as follows:

- InputStream classes
- OutputStream classes

InputStream Classes in Java

- InputStream class is an abstract class.
- It is the superclass of all classes representing an input stream of bytes.
  Since InputStream class is an abstract class, we cannot create an object of this class.

- We must use subclasses of this class to create an object.
- The several subclasses of Java InputStream class can be used for performing several input functions.
- They are listed with a brief description in the below table:

InputStream Subclass	Description
BufferedInputStream	It adds buffering abilities to an input stream. In simple words, it buffered input stream.
ByteArrayInputStream	Input stream that reads data from a byte array.
DataInputStream	It reads bytes from the input stream and converts them into appropriate primitive-type val strings.
FileInputStream	This input stream reads bytes from a file.

InputStream Methods in Java

1.**int read()**:

The read() method reads the next byte of data from the input stream.

2. int read(byte[ ] b): This method reads the number of bytes from the input stream and stores them into the array of bytes b.
It returns the total number of bytes read as an int. If the end of stream is reached, returns -1.

3. int read(byte[ ] b, int n, int m): It reads up to m bytes of data from the input stream starting from nth byte into an array b.
It returns the total number of bytes read as an int. Returns -1 at the end of stream because of no more data.

---

4. int available(): The available method returns an estimate of the number of bytes that can be read from the input stream.

5. void close(): The close() method closes the input stream and releases any system resources associated with it.
6. long skip(long n): This method skips over n bytes of data from this input stream. It returns the actual number of bytes skipped.
7. void reset(): The reset() method is used to go back to the beginning of the stream.
.

OutputStream Classes in Java

- OutputStream class is an abstract class. It is the root class for writing binary data.
- It is a superclass of all classes that represents an output stream of bytes.
- Since like InputStream, OutputStream is an abstract class, therefore, we cannot create object of it.
- The several subclasses of OutputStream class in Java can be used for performing several output functions.
- They are listed with a brief description in the below table:

OutputStream Subclass	Description
BufferedOutputStream	It adds buffering abilities to an output stream. In simple words, it buffered output stre
ByteArrayOutputStream	Output stream that writes data to a byte array.
DataOutputStream	It converts primitive-type values or strings into bytes and outputs bytes to the stream.
FileOutputStream	It writes byte stream into a file.

OutputStream Methods in Java

---

1. void write(int b): The write() method writes the specified byte to the output stream. It accepts an int value as an input parameter

2. void write(byte[ ] b): This method writes all the specified bytes in the array b to the output stream.
3. void write(byte[ ] b, int n, int m): It writes m bytes from array b starting from nth byte to the output stream.
4. void close(): It closes the output stream and releases any system resources associated with this stream.
5. void flush(): It flushes the output stream and forces any buffered output bytes to be written out.

Character Stream Classes in Java

- **CharacterStream classes in Java** are used to read and write 16-bit Unicode characters.
- In other words, character stream classes are mainly used to read characters from the source and write them to the destination.
- They can perform operations based on characters, char arrays, and Strings.

Note: Unicode is a 2-byte, 16-bit character set having 65,536 (i.e. 2^16) different possible characters. Only about 40,000 characters are used in practice, the rest are reserved for future purposes. It can handle most of the world's living languages.

Types of CharacterStream classes in Java

Like byte stream classes, character stream classes also contain two kinds of classes. They are named as:

- Reader Stream classes
- Writer Stream classes

Let's understand in detail reader stream classes and writer stream classes one by one.

Reader Stream Classes

- Reader stream classes are used for reading characters from files.
- Reader is an abstract superclass for all other subclasses such as BufferedReader, StringReader, CharArrayReader, etc
- The classes belonging to Reader stream classes are very similar functionality to Input Stream classes.
- The only difference is that Input Stream uses bytes, whereas Reader Stream classes use characters.

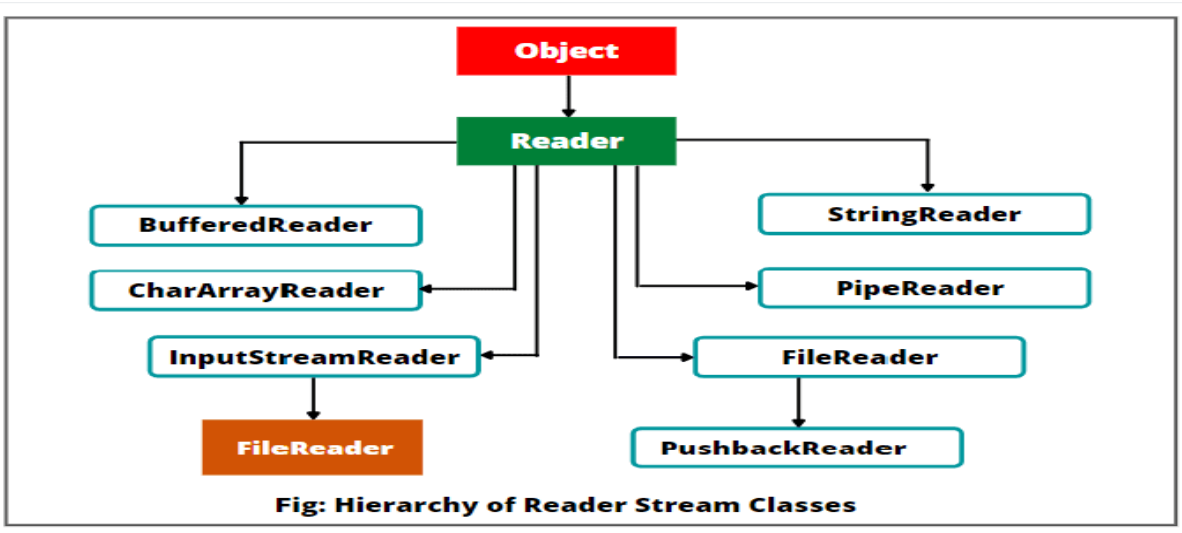


Fig: Hierarchy of Reader Stream Classes

Reader Subclasses

1. BufferedReader: This class is used to read characters from the buffered input character stream.

2. CharArrayReader: This class is used to read characters from the char array or character array.
3. FileReader: This class is used to read characters (or contents) from a file.
4. InputStreamReader: This class is used to translate (or convert) bytes to characters.

Reader Class Methods in Java

---

1. int read(): This method returns an integer representation of the next character present in the invoking input stream

2. int read(char buffer[ ]): This method is used to read up to buffer.length characters from the specified buffer. It returns the actual number of characters successfully read. It returns -1 when the end of the input is encountered.
3. void close(): This method is used to close the input stream. If the program attempts to read the input further, it generates IOException.

Writer Stream Classes

- Writer stream classes are used to write characters to a file. In other words, They are used to perform all output operations on files.
- Writer stream classes are similar to output stream classes with only one difference that output stream classes use bytes to write while writer stream classes use characters to write.
- Since the Writer class is an abstract class, it cannot be instantiated.

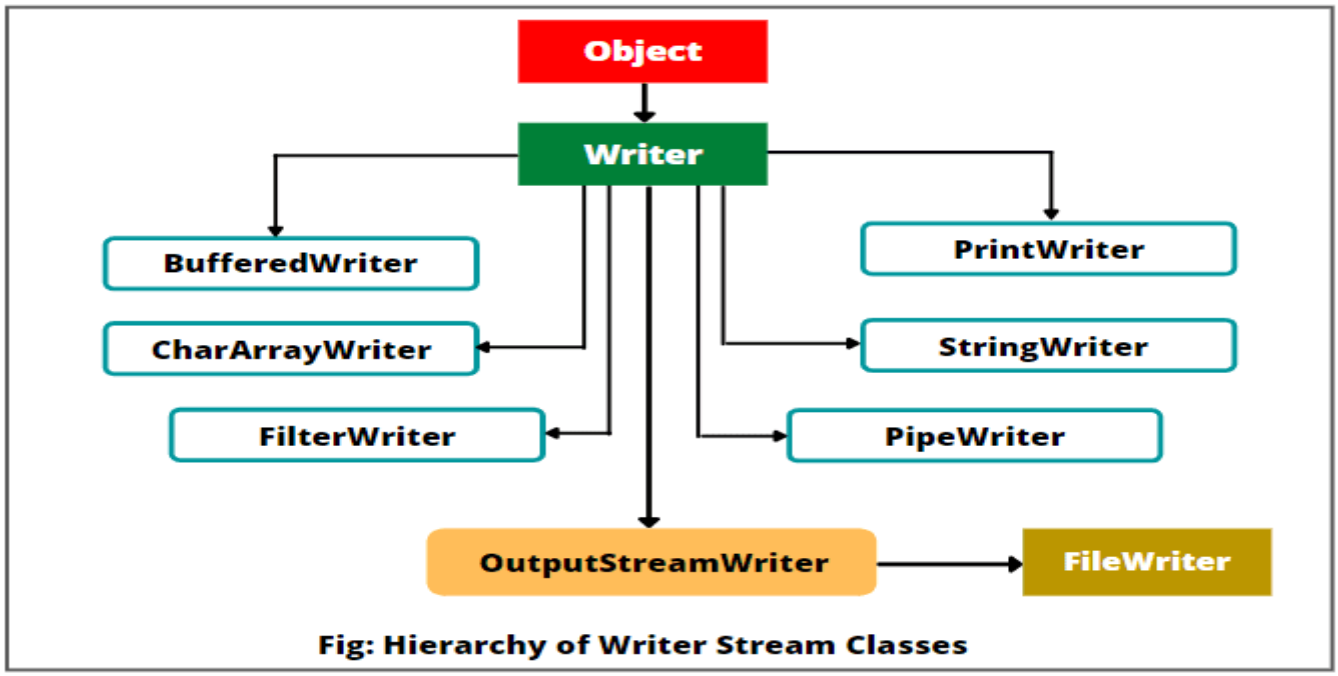


Fig: Hierarchy of Writer Stream Classes

Writer Subclasses

Let's understand the subclasses of writer class in a brief description.

1. BufferedWriter: This class is used to write characters to the buffered output character stream.
2. FileWriter: This output stream class writes characters to the file.
3. CharArrayWriter: This output stream class writes the characters to the character array.
4. OutpuStreamWriter: This output stream class translates or converts from bytes to characters.

Writer Class Methods in Java

---

1. void write(int ch): This method is used to write a single character to the invoking output stream.
2. void write(char buffer[ ]): This method is used to write a complete array of characters to the invoking output stream.

3. void write(String str): This method is used to write str to the invoking output stream.

---

4. void close (): This method is used to close the output stream. It will produce an IOException if an attempt is made to write to the output stream after closing the stream.
5. void flush (): This method flushes the output stream and writes the waiting buffered characters.
6. Writer append(char ch): The append() method appends character ch to the end of the invoking output stream. It returns a reference to the invoking output stream.

Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.

File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Useful Methods

Modifier and Type	Method	Description
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.

String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent di
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Create a File

CreateFile.java

```
1. // Importing File class
2. import java.io.File;
3. // Importing the IOException class for handling errors
4. import java.io.IOException;
5.  class CreateFile {
6.             public static void main(String args[]) {
7.             try {
8.                     // Creating an object of a file
```

```
9.                  File f0 = new File("D:FileOperationExample.txt");
10.                 if (f0.createNewFile()) {
11.                         System.out.println("File " + f0.getName() + " is created successf
    ully.");
12.                 } else {
13.                         System.out.println("File is already exist in the directory.");
14.                 }
15.               } catch (IOException exception) {
16.                       System.out.println("An unexpected error is occurred.");
17.                       exception.printStackTrace();
18.               }
19.       }
20. }
```

WriteToFile.java

```
1.  // Importing the FileWriter class
2.  import java.io.FileWriter;
3.
4.  // Importing the IOException class for handling errors
5.  import java.io.IOException;
6.
7.  class WriteToFile {
8.     public static void main(String[] args) {
9.
10.    try {
11.       FileWriter fwrite = new FileWriter("D:FileOperationExample.txt");
12.       // writing the content into the FileOperationExample.txt file
13.       fwrite.write("A named location used to store related information is referred to as
     a File.");
14.
15.       // Closing the stream
16.       fwrite.close();
17.       System.out.println("Content is successfully wrote to the file.");
18.    } catch (IOException e) {
19.       System.out.println("Unexpected error occurred");
```

```
20.         e.printStackTrace();
21.         }
22.     }
23. }
```

ReadFromFile.java

```
1.  // Importing the File class
2.  import java.io.File;
3.  // Importing FileNotFoundException class for handling errors
4.  import java.io.FileNotFoundException;
5.  // Importing the Scanner class for reading text files
6.  import java.util.Scanner;
7.
8.  class ReadFromFile {
9.      public static void main(String[] args) {
10.         try {
11.             // Create f1 object of the file to read data
12.             File f1 = new File("D:FileOperationExample.txt");
13.             Scanner dataReader = new Scanner(f1);
14.             while (dataReader.hasNextLine()) {
15.                 String fileData = dataReader.nextLine();
16.                 System.out.println(fileData);
17.             }
18.             dataReader.close();
19.         } catch (FileNotFoundException exception) {
20.             System.out.println("Unexcpected error occurred!");
21.             exception.printStackTrace();
22.         }
23.     }
24. }
```

DeleteFile.java

```
1.  // Importing the File class
2.  import java.io.File;
3.  class DeleteFile {
4.    public static void main(String[] args) {
```

```
5.      File f0 = new File("D:FileOperationExample.txt");
6.      if (f0.delete()) {
7.        System.out.println(f0.getName()+ " file is deleted successfully.");
8.      } else {
9.        System.out.println("Unexpected error found in deletion of the file.");
10.     }
11.   }
12. }
```