

# UNIT-2 NOTES

## OF

## OOPS

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.

**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
- Opening an unavailable file

**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.

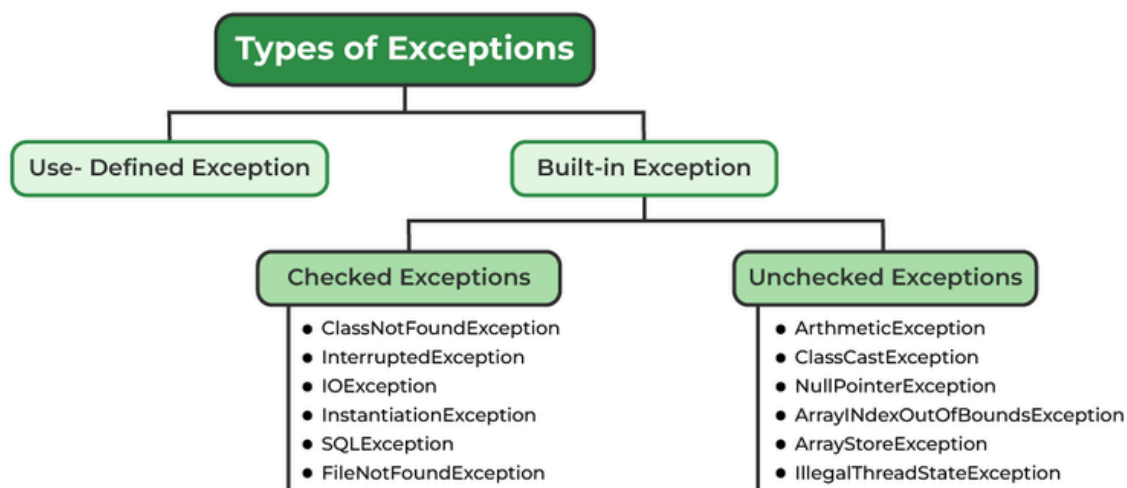
### **Difference between Error and Exception**

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.

### **Types of Exceptions**

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



**Exceptions can be categorized in two ways:**

### 1. Built-in Exceptions

- Checked Exception
- Unchecked Exception

### 2. User-Defined Exceptions

Let us discuss the above-defined listed exception that is as follows:

#### 1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

**Note:** For checked vs unchecked exception, see [Checked vs Unchecked Exceptions](#)

#### 2. User-Defined Exceptions:

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

## 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){
7.             System.out.println(e);
8.         }
9.         //rest code of the program
10.        System.out.println("rest of the code..");
11.    }
12. }
```

#### 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 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.

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

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

#### MultipleCatchBlock1.java

```
1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.         }
9.         catch(ArithmeticException e)
10.            {
11.                System.out.println("Arithmetic Exception occurs");
12.            }
13.        catch(ArrayIndexOutOfBoundsException e)
14.            {
15.                System.out.println("ArrayIndexOutOfBoundsException occurs");
16.            }
17.        catch(Exception e)
18.            {
19.                System.out.println("Parent Exception occurs");
20.            }
21.        System.out.println("rest of the code");
22.    }
23. }
```

## Java finally block

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

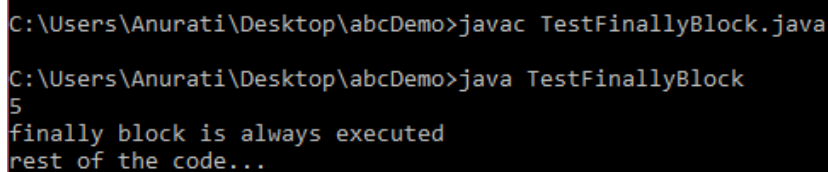
The finally block follows the try-catch block.

- o finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- o The important statements to be printed can be placed in the finally block.

### TestFinallyBlock.java

```
1. class TestFinallyBlock {
2.     public static void main(String args[]){
3.         try{
4.             //below code do not throw any exception
5.             int data=25/5;
6.             System.out.println(data);
7.         }
8.         //catch won't be executed
9.         catch(NullPointerException e){
10.            System.out.println(e);
11.        }
12.        //executed regardless of exception occurred or not
13.        finally {
14.            System.out.println("finally block is always executed");
15.        }
16.
17.        System.out.println("rest of phe code...");
18.    }
19. }
```

### Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

## When an exception occurs but not handled by the catch block

Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

### TestFinallyBlock1.java

```
1. public class TestFinallyBlock1{
2.     public static void main(String args[]){
3.
4.         try {
5.
6.             System.out.println("Inside the try block");
7.
8.             //below code throws divide by zero exception
9.             int data=25/0;
10.            System.out.println(data);
11.        }
12.        //cannot handle Arithmetic type exception
13.        //can only accept Null Pointer type exception
```

```

14.     catch(NullPointerException e){
15.         System.out.println(e);
16.     }
17.
18.     //executes regardless of exception occurred or not
19.     finally {
20.         System.out.println("finally block is always executed");
21.     }
22.     System.out.println ("rest of the code...");
23. }
24. }

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestFinallyBlock1.main(TestFinallyBlock1.java:9)

```

## 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. We will discuss custom exceptions later in this section.

```

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. }

```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
vote
    at TestThrow1.validate(TestThrow1.java:8)
    at TestThrow1.main(TestThrow1.java:18)
```

### Exceptions as Control Flow

Using exceptions as control flow is generally considered a bad practice. While exceptions are a mechanism to stop unexpected behavior in software development, abusing them in expected behaviors can lead to negative consequences.

This practice can reduce the performance of the code and make it less readable

### Reasons to Avoid Using Exceptions as Control Flow

1. **Performance Impact:** Using exceptions as control flow can reduce the performance of the code as a response per unit time
2. **Readability:** It makes the code less readable and hides the programmer's intention, which is considered a bad practice

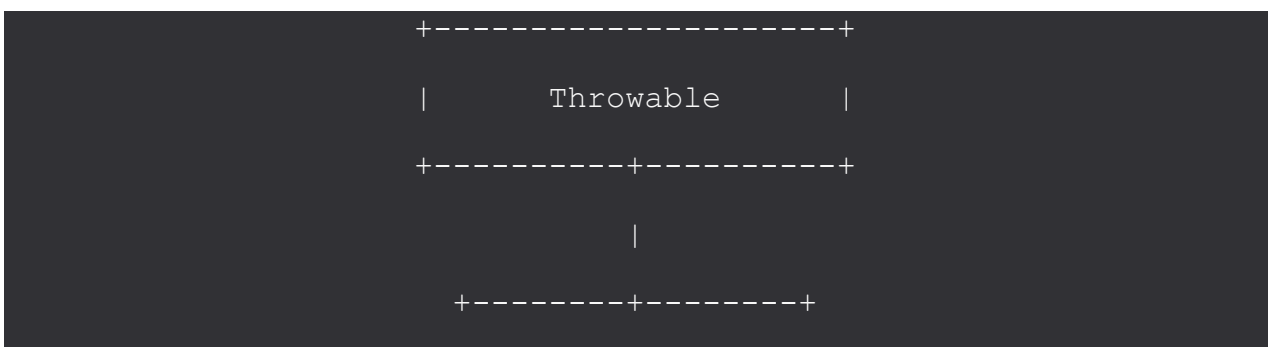
### Alternatives to Using Exceptions for Control Flow

Instead of using exceptions for expected behaviours, it is recommended to use control flow statements to handle the logic.

If the behaviours are expected, using control flow statements is generally considered better than throwing an exception and handling it by yourself

### JVM Reaction to Exceptions,

Certainly! Here is a diagram illustrating the Java exception hierarchy and the various types of exceptions:





```

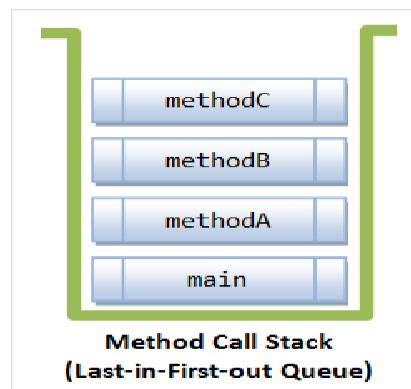
|      Error      |
+-----+
|      Exception  |
+-----+-----+
|
+-----+-----+
| CompileTimeException|
|
or
| Checked Exception|
+-----+-----+
| RuntimeException |
|
or
|Unchecked Exception|
+-----+

```

In this hierarchy:

- **Throwable** is the base class for all exceptions and errors in Java.
- **Error** is used by the Java runtime system (JVM) to indicate errors related to the runtime environment itself.
- **Exception** is used for exceptional conditions that user programs should catch. It has two main branches:
  - o **Unchecked Exception:** These are unchecked exceptions that occur at runtime and are not checked by the compiler.
  - o **Checked Exception:** These are exceptions that occur at compile time and are checked by the compiler.

This diagram provides a clear overview of the Java exception hierarchy and the different types of exceptions.



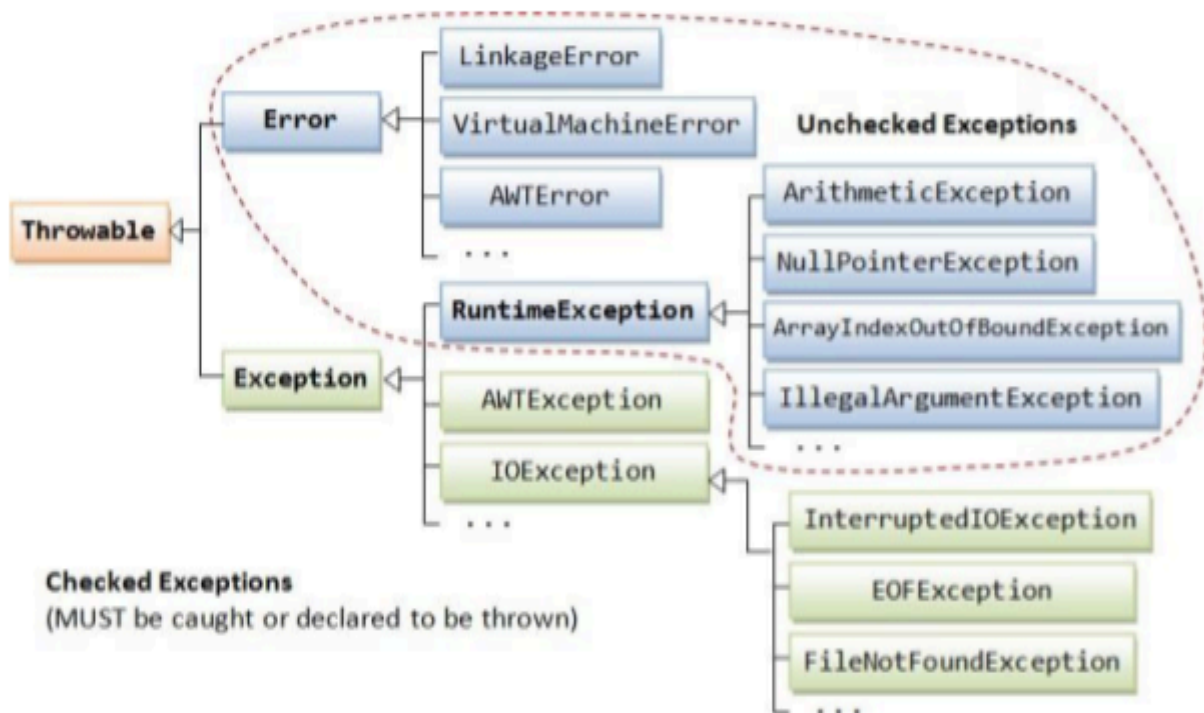
### 1.3 Exception & Call Stack

When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception). The Exception object contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an *exception handler* to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

## Exception Classes - Throwable, Error, Exception & RuntimeException

The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is

.....`java.lang.Throwable`, together with its two subclasses `java.lang.Exception` and `java.lang.Error`.....



The `Error` class describes internal system errors (e.g., `VirtualMachineError`, `LinkageError`) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.

The `Exception` class describes the error caused by your program (e.g., `FileNotFoundException`, `IOException`). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

## Use of try

Certainly! Here's a diagram illustrating the use of the try...catch block in Java:

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            // Code that may throw an exception  
  
            int[] myNumbers = {1, 2, 3};  
  
            System.out.println(myNumbers[10]); // This line  
may throw an exception  
  
        } catch (Exception e) {  
            // Code to handle the exception  
  
            System.out.println("Something went wrong.");  
  
        } finally {  
            // Code that always executes, regardless of the  
result  
  
            System.out.println("The 'try catch' is  
finished.");  
  
        }  
    }  
}
```

In this diagram:

- The `try` block contains the code that may throw an exception.
- The `catch` block is used to catch and handle the exception. It contains the code that executes when an exception is thrown.
- The `finally` block contains the code that always executes, regardless of whether an exception was thrown or caught.
-

## 1.7 try-catch-finally

---

The syntax of try-catch-finally is:

```
try {  
    // main logic, uses methods that may throw Exceptions  
    .....  
} catch (Exception1 ex) {  
    // error handler for Exception1  
    .....  
} catch (Exception2 ex) {  
    // error handler for Exception1  
    .....  
} finally {           // finally is optional  
    // clean up codes, always executed regardless of exceptions  
    .....  
}
```

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks *sequentially*. If a catch-block catches that exception class or catches a *superclass* of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a `System.exit()` is called in the catch block).

### Example 1

---

```

import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class TryCatchFinally
{
    public static void main(String[] args)
    {
        try
        { // main logic
            System.out.println("Start of the main logic");
            System.out.println("Try opening a file ...");
            Scanner in = new Scanner(new File("test.in"));

            System.out.println("File Found, processing the file ...");
            System.out.println("End of the main logic");

        } catch (FileNotFoundException ex) { // error handling separated from the
            // main logic
            System.out.println("File Not Found caught ...");
        } finally { // always run regardless of exception status
            System.out.println("finally-block runs regardless of the state of exception");
        }
        // after the try-catch-finally
        System.out.println("After try-catch-finally, life goes on...");
    } finally { // finally is optional
        // clean up codes, always executed regardless of exceptions
        .....
    }
}

```

## CATCH

### Catch

In Java, the `catch` keyword is used as part of the `try...catch` block to handle exceptions. Here's a summary of its usage based on the provided search results:

- The `catch` statement allows you to define a block of code to be executed if an error occurs in the `try` block. It comes in pairs with the `try` statement.

- o Syntax:

```
o try {
o     // Block of code to try
o } catch(Exception e) {
o     // Block of code to handle errors
o }
```

- o The `catch` block catches and handles the exceptions by declaring the type of exception within the parameter. It includes the code that is executed if an exception inside the `try` block occurs.
- o It is used to handle uncertain conditions of a `try` block and must be followed by the `try` block.
- o Multiple `catch` blocks can be used with a single `try` block to handle different types of exceptions.

The `catch` block is an essential part of exception handling in Java, allowing developers to gracefully handle errors and prevent the abnormal termination of the program.

### Finally

The `finally` block is used to execute important code such as releasing resources, regardless of whether an exception is thrown or not in the `try` block.

- o Syntax:

```
o try {
o     // Block of code to try
o } catch(Exception e) {
o     // Block of code to handle errors
o } finally {
o     // Block of code to execute regardless of an
o     exception
o }
```

- o The `finally` block always executes, even if an exception is thrown and caught.
- o It is often used to release resources like file handles, database connections, or network connections, to ensure that these resources are properly closed or released.
- o The `finally` block is optional, but when used, it ensures that certain code will always be executed, making it useful for cleanup or finalization tasks.

### Throw

the `throw` keyword is used to explicitly throw an exception. Here's a summary of its usage based on the provided search results:

- The `throw` statement is used to throw an exception explicitly within a method or block of code.

- o Syntax:

```
o throw new ExceptionType("Error message");
```

- o The `throw` statement is followed by the keyword `new` and the constructor of the exception type to create and throw an instance of that exception.
- o It is typically used to handle exceptional situations where the program encounters an error or an unexpected condition.
- o The exception that is thrown can be a built-in exception class provided by Java, or it can be a custom exception class created by the programmer.

- 

### throws in Exception Handling,

`throws` keyword is used in method declarations to specify which exceptions are not handled within the method but are instead propagated to the calling code to be handled. Here's an overview of its usage:

#### Usage of the **throws** Keyword:

- **Syntax:**

```
• returnType methodName(parameters) throws ExceptionType1,  
  ExceptionType2, ... {  
•    // Method body  
• }
```

- The `throws` keyword is used in the method signature to indicate that the method may throw one or more exceptions of the specified types.
- When a method defines a `throws` clause, it informs the calling code that it is not handling the specified exceptions internally, and it's the responsibility of the calling code to handle or propagate these exceptions.
- The calling code must either handle the exceptions using a `try-catch` block or declare the exceptions to be thrown further up the call stack.
- Example:

```
• public void readFile(String fileName) throws IOException  
  {  
•    // Method implementation that may throw IOException  
• }
```

In the example above, the `readFile` method declares that it may throw an `IOException`. This means that any code calling the `readFile` method must either handle the `IOException` or declare it to be thrown further up the call stack

## Types of Exception in Java

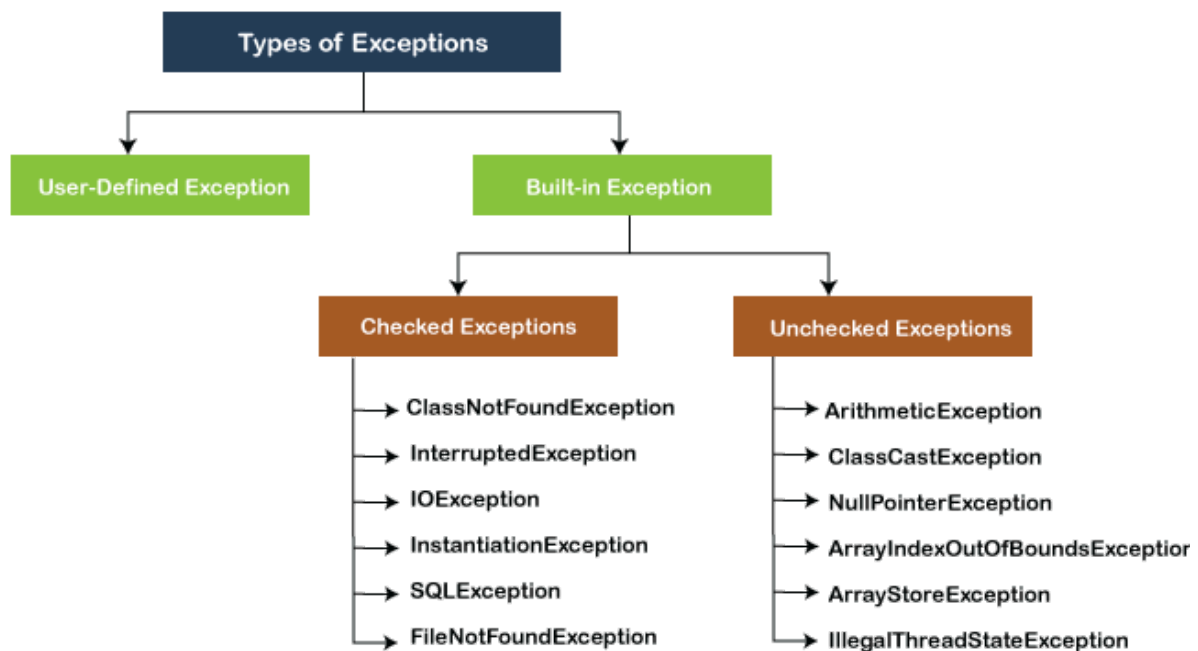
In Java, **exception** is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. Bugs or errors that we don't want and restrict our program's normal execution of code are referred to as **exceptions**.



In this section, we will focus on the **types of exceptions in Java** and the differences between the two.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
  - o Checked Exception
  - o Unchecked Exception
2. User-Defined Exceptions



## Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

### Checked Exception

**Checked** exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

#### CheckedExceptionExample.java

```
1. import java.io.*;
2. class CheckedExceptionExample {
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
6.         int m;
```

```

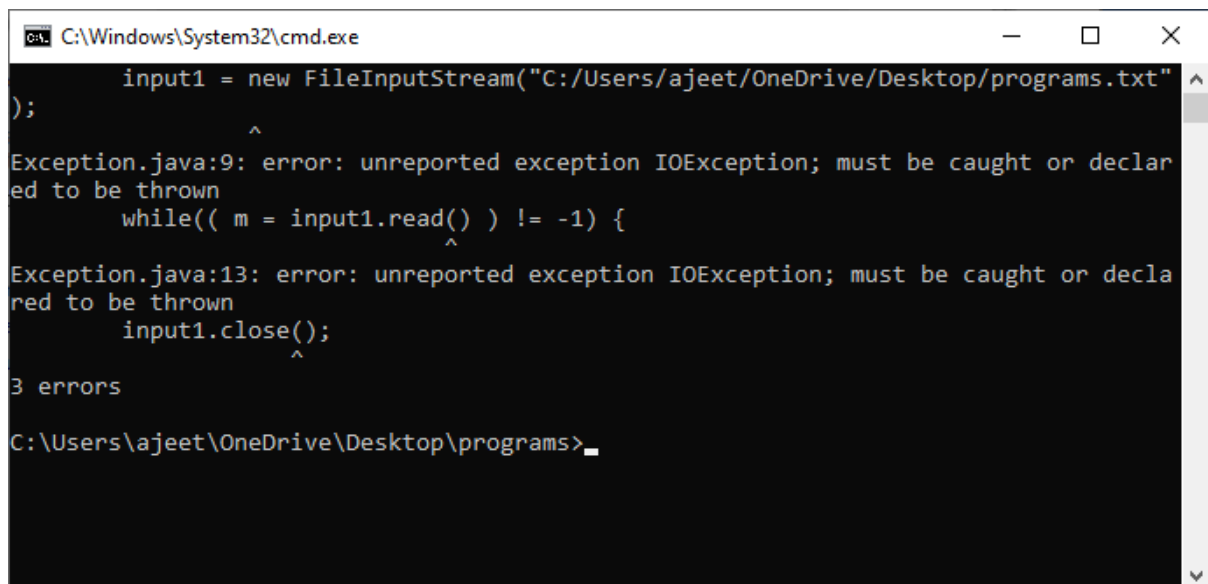
7.     while(( m = file_data.read() ) != -1) {
8.         System.out.print((char)m);
9.     }
10.    file_data.close();
11. }
12. }

```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the **IOException**.

Output:



```

C:\Windows\System32\cmd.exe
input1 = new FileInputStream("C:/Users/ajet/OneDrive/Desktop/programs.txt"
);
^
Exception.java:9: error: unreported exception IOException; must be caught or declar
ed to be thrown
    while(( m = input1.read() ) != -1) {
        ^
Exception.java:13: error: unreported exception IOException; must be caught or decla
red to be thrown
    input1.close();
        ^
3 errors
C:\Users\ajet\OneDrive\Desktop\programs>_

```

### How to resolve the error?

There are basically two ways through which we can solve these errors.

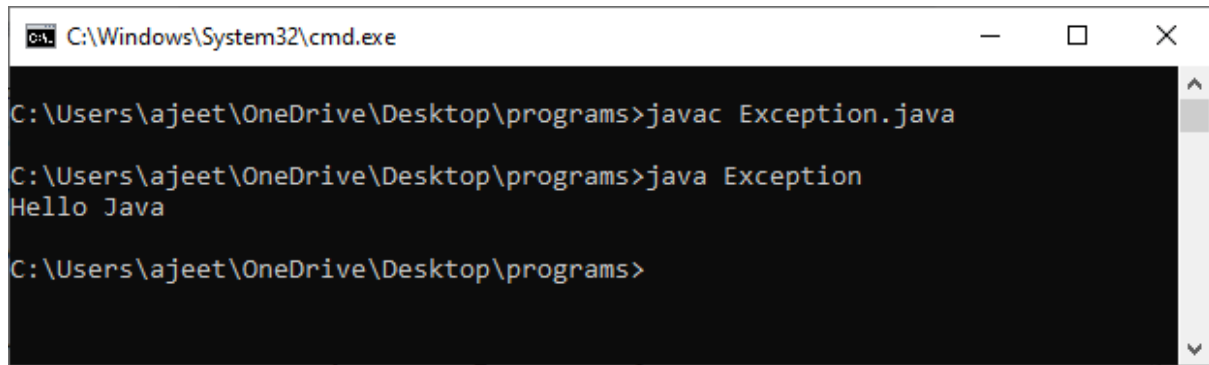
1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **throws**. We only declare the **IOException**, not **FileNotFoundException**, because of the child-parent relationship. The **IOException** class is the parent class of **FileNotFoundException**, so this exception will automatically cover by **IOException**. We will declare the exception in the following way:

```

1. class Exception{
2.     public static void main(String args[]) throws IOException {
3.         ...
4.         ...
5.     }

```

If we compile and run the code, the errors will disappear, and we will see the data of the file.



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
Hello Java

C:\Users\ajeet\OneDrive\Desktop\programs>
```

2) We can also handle these exception using **try-catch** However, the way which we have used above is not correct. We have to give a meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

### Exception.java

```
1. import java.io.*;
2. class Exception{
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         try{
6.             file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
7.         }catch(FileNotFoundException fnfe){
8.             System.out.println("File Not Found!");
9.         }
10.        int m;
11.        try{
12.            while(( m = file_data.read() ) != -1) {
13.                System.out.print((char)m);
14.            }
15.            file_data.close();
16.        }catch(IOException ioe){
17.            System.out.println("I/O error occurred: "+ioe);
18.        }
19.    }
20. }
```

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.

```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
File Not Found!
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.io.FileInputStream.read()" because "<local1>" is null
    at Exception.main(Exception.java:14)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

## Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

*Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.*

### UncheckedExceptionExample1.java

```
1. class UncheckedExceptionExample1 {
2.     public static void main(String args[])
3.     {
4.         int positive = 35;
5.         int zero = 0;
6.         int result = positive/zero;
7.         //Give Unchecked Exception here.
8.         System.out.println(result);
9.     }
10. }
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a unchecked exception.

Output:

```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at UncheckedException1.main(UncheckedException1.java:7)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

### UncheckedException1.java

```
1. class UncheckedException1 {
```

```

2. public static void main(String args[])
3. {
4.     int num[] = {10,20,30,40,50,60};
5.     System.out.println(num[7]);
6. }
7. }

```

Output:

```

C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index
7 out of bounds for length 6
    at UncheckedException1.main(UncheckedException1.java:5)

C:\Users\ajeet\OneDrive\Desktop\programs>_

```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the `ArrayIndexOutOfBoundsException` at runtime.

## User-defined Exception

In [Java](#), we already have some built-in exception classes like [ArrayIndexOutOfBoundsException](#), [NullPointerException](#), and [ArithmeticException](#). These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the `Exception` class. We can throw our own exception on a particular condition using the `throw` keyword. For creating a user-defined exception, we should have basic knowledge of the [try-catch](#) block and [throw keyword](#).

Let's write a [Java program](#) and create user-defined exception.

### Difference between Checked and Unchecked Exception

S.No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to checked exceptions. These exceptions are not checked and handled at compile time.

Java	2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of RuntimeException class.
	3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error but the exceptions escape the notice of the compiler. These exceptions are the result of user-created errors in programming logic.
	4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.
	5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
	6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
	7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

**I/O** (Input and Output) is used *to process the input and produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

## Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) **System.out**: standard output stream

2) **System.in**: standard input stream

3) **System.err**: standard error stream

Let's see the code to print **output and an error** message to the console.

1. `System.out.println("simple message");`
2. `System.err.println("error message");`

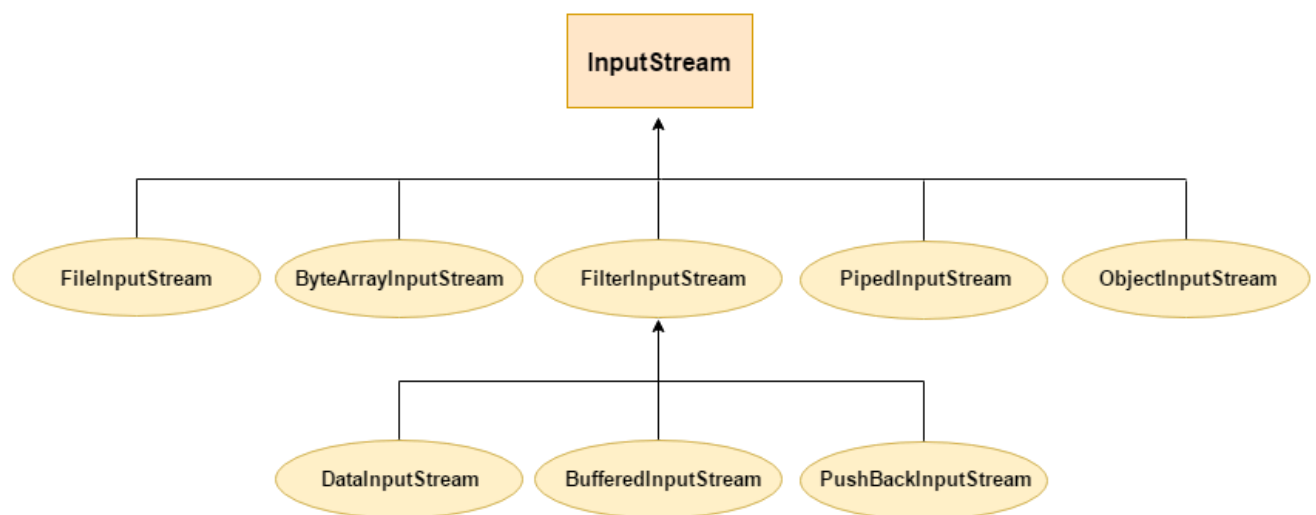
## InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Useful methods of InputStream

Method	Description
1) <code>public abstract int read()throws IOException</code>	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) <code>public int available()throws IOException</code>	returns an estimate of the number of bytes that can be read from the current input stream.
3) <code>public void close()throws IOException</code>	is used to close the current input stream.

### InputStream Hierarchy



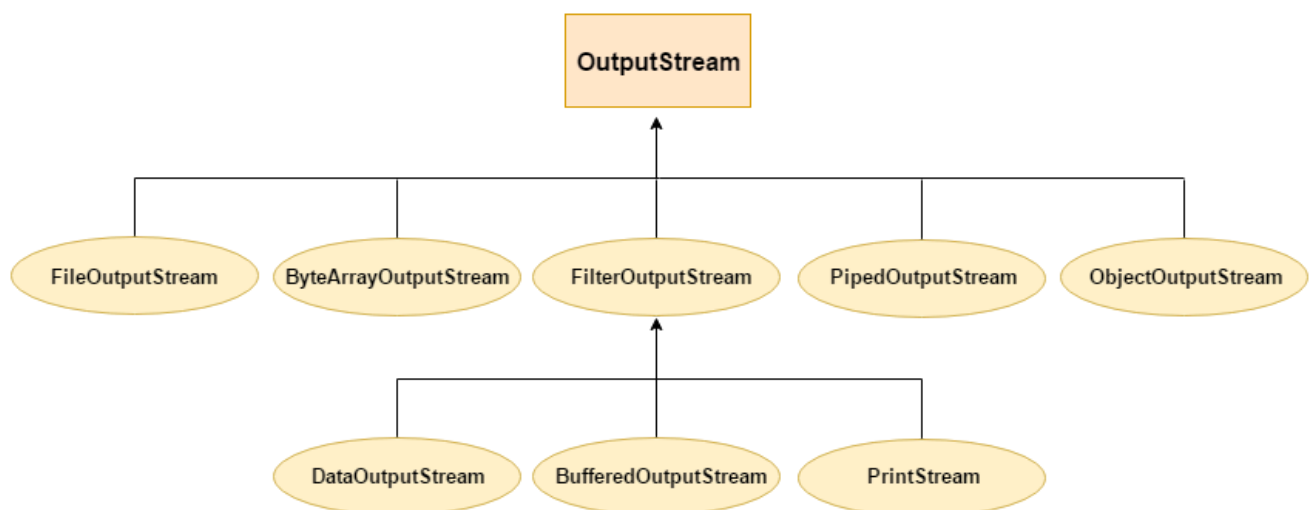
## OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

## Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

## OutputStream Hierarchy



## Java FileInputStream Class

Java `FileInputStream` class obtains input bytes from a [file](#). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use [FileReader](#) class.

## Java FileInputStream class declaration



Let's see the declaration for java.io.FileInputStream class:

1. **public class** FileInputStream **extends** InputStream
- 

## Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to <b>b.length</b> bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to <b>len</b> bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the <a href="#">FileDescriptor</a> object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the <a href="#">stream</a> .

## Java FileInputStream example 1: read single character

1. **import** java.io.FileInputStream;
2. **public class** DataStreamExample {
3.     **public static void** main(String args[]){
4.         **try**{
5.             FileInputStream fin=**new** FileInputStream("D:\\testout.txt");
6.             **int** i=fin.read();

```
7.      System.out.print((char)i);
8.
9.      fin.close();
10.     }catch(Exception e){System.out.println(e);}
11.     }
12. }
```

## Java FileOutputStream Class

Java `FileOutputStream` is an output stream used for writing data to a [file](#).

If you have to write primitive values into a file, use `FileOutputStream` class. You can write byte-oriented as well as character-oriented data through `FileOutputStream` class. But, for character-oriented data, it is preferred to use [FileWriter](#) than `FileOutputStream`.

---

### FileOutputStream class declaration

Let's see the declaration for `Java.io.FileOutputStream` class:

1. `public class` `FileOutputStream` `extends` `OutputStream`

---

### FileOutputStream class methods

Method	Description
<code>protected void finalize()</code>	It is used to clean up the connection with the file output stream.
<code>void write(byte[] ary)</code>	It is used to write <b>ary.length</b> bytes from the byte <a href="#">array</a> to the file output stream.
<code>void write(byte[] ary, int off, int len)</code>	It is used to write <b>len</b> bytes from the byte array starting at offset <b>off</b> to the file output stream.
<code>void write(int b)</code>	It is used to write the specified byte to the file output stream.
<code>FileChannel getChannel()</code>	It is used to return the file channel object associated with the file output stream.

FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

## Java FileOutputStream Example 1: write byte

```

1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.         try{
5.             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.             fout.write(65);
7.             fout.close();
8.             System.out.println("success...");
9.         }catch(Exception e){System.out.println(e);}
10.    }
11. }

```

Output:

Success...

## CharacterStream Classes in Java

The java.io package provides CharacterStream classes to overcome the limitations of ByteStream classes, which can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters. CharacterStream classes are used to work with 16-bit Unicode characters. They can perform operations on characters, char arrays and Strings.

However, the CharacterStream classes are mainly used to read characters from the source and write them to the destination. For this purpose, the CharacterStream classes are divided into two types of classes, i.e., Reader class and Writer class.

### Reader Class

[Reader class](#) is used to read the 16-bit characters from the input stream. However, it is an abstract class and can't be instantiated, but there are various subclasses that inherit the Reader class and override the methods of the Reader class. All methods of the Reader class throw an IOException. The subclasses of the Reader class are given in the following table.

SN	Class	Description
1.	<a href="#">BufferedReader</a>	This class provides methods to read characters from the buffer.

2.	<a href="#">CharArrayReader</a>	This class provides methods to read characters from the char array.
3.	<a href="#">FileReader</a>	This class provides methods to read characters from the file.
4.	<a href="#">FilterReader</a>	This class provides methods to read characters from the underlying character input stream.
5	<a href="#">InputStreamReader</a>	This class provides methods to convert bytes to characters.
6	<a href="#">PipedReader</a>	This class provides methods to read characters from the connected piped output stream.
7	<a href="#">StringReader</a>	This class provides methods to read characters from a string.

The Reader class methods are given in the following table.

SN	Method	Description
1	int read()	This method returns the integral representation of the next character present in the input. It returns -1 if the end of the input is encountered.
2	int read(char buffer[])	This method is used to read from the specified buffer. It returns the total number of characters successfully read. It returns -1 if the end of the input is encountered.
3	int read(char buffer[], int loc, int nChars)	This method is used to read the specified nChars from the buffer at the specified location. It returns the total number of characters successfully read.
4	void mark(int nchars)	This method is used to mark the current position in the input stream until nChars characters are read.
5	void reset()	This method is used to reset the input pointer to the previous set mark.
6	long skip(long nChars)	This method is used to skip the specified nChars characters from the input stream and returns the number of characters skipped.

7	boolean ready()	This method returns a boolean value true if the next request of input is ready. Otherwise, it returns false.
8	void close()	This method is used to close the input stream. However, if the program attempts to access the input, it generates IOException.

## Writer Class

Writer class is used to write 16-bit Unicode characters to the output stream. The methods of the Writer class generate IOException. Like Reader class, Writer class is also an abstract class that cannot be instantiated; therefore, the subclasses of the Writer class are used to write the characters onto the output stream. The subclasses of the Writer class are given in the below table.

SN	Class	Description
1	<a href="#"><u>BufferedWriter</u></a>	This class provides methods to write characters to the buffer.
2	<a href="#"><u>FileWriter</u></a>	This class provides methods to write characters to the file.
3	<a href="#"><u>CharArrayWriter</u></a>	This class provides methods to write the characters to the character array.
4	<a href="#"><u>OutputStreamWriter</u></a>	This class provides methods to convert from bytes to characters.
5	<a href="#"><u>PipedWriter</u></a>	This class provides methods to write the characters to the piped output stream.
6	<a href="#"><u>StringWriter</u></a>	This class provides methods to write the characters to the string.

To write the characters to the output stream, the Write class provides various methods given in the following table.

SN	Method	Description
1	void write()	This method is used to write the data to the output stream.
2	void write(int i)	This method is used to write a single character to the output stream.

3	Void write(char buffer[])	This method is used to write the array of characters to the output stream.
4	void write(char buffer [],int loc, int nChars)	This method is used to write the nChars characters to the character array from the specified location.
5	void close ()	This method is used to close the output stream. However, this generates the IOException if an attempt is made to write to the output stream after closing the stream.
6	void flush ()	This method is used to flush the output stream and writes the waiting buffered characters.