

UNIT - 3

EXCEPTION HANDLING AND I/O

3.1 EXCEPTIONS

An exception is an unexpected event, which may occur during the execution of a program (at run time), to disrupt the normal flow of the program's instructions. This leads to the abnormal termination of the program, which is not always recommended.

Therefore, these exceptions are needed to be handled. The exception handling in java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

An exception may occur due to the following reasons. They are.

- Invalid data as input.
- Network connection may be disturbed in the middle of communications
- JVM may run out of memory.
- File cannot be found/opened.

These exceptions are caused by user error, programmer error, and physical resources.

Based on these, the exceptions can be classified into three categories.

- **Checked exceptions** – A checked exception is an exception that occurs at the compile time, also called as compile time (static time) exceptions. These exceptions cannot be ignored at the time of compilation. So, the programmer should handle these exceptions.
- **Unchecked exceptions** – An unchecked exception is an exception that occurs at run time, also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- **Errors** – Errors are not exceptions, but problems may arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

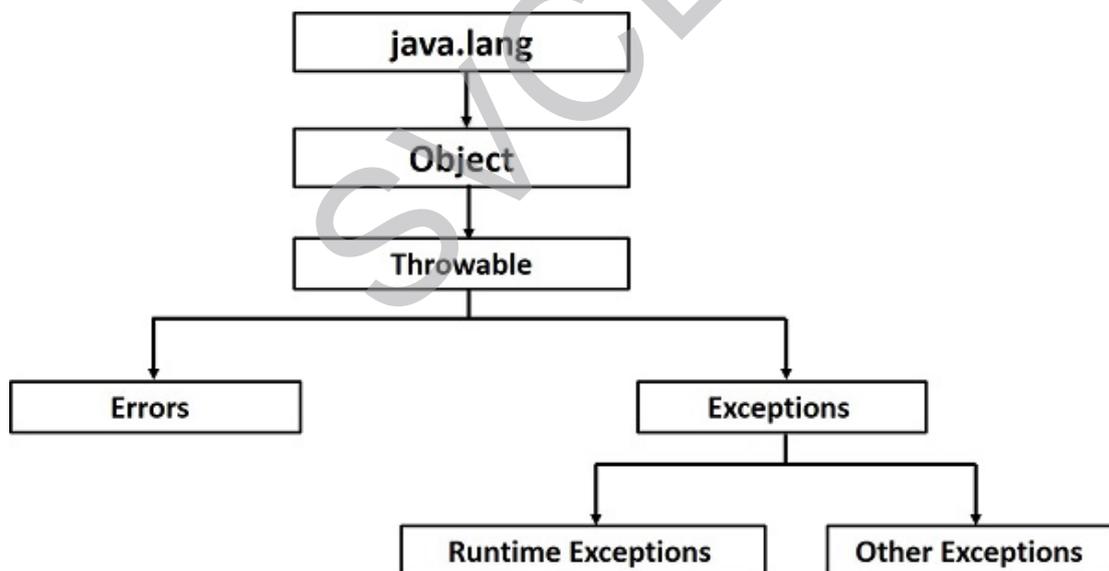
- **Error:** An Error indicates serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

3.2 EXCEPTION HIERARCHY

The `java.lang.Exception` class is the base class for all exception classes. All exception and errors types are sub classes of class `Throwable`, which is base class of 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` are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). `StackOverflowError` is an example of such an error.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The `Exception` class has two main subclasses: `IOException` class and `RuntimeException` Class.



Exceptions Methods

Method	Description
public String getMessage()	Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
public Throwable getCause()	Returns the cause of the exception as represented by a Throwable object.
public String toString()	Returns the name of the class concatenated with the result of getMessage().
public void printStackTrace()	Prints the result of toString() along with the stack trace to System.err, the error output stream.
public StackTraceElement [] getStackTrace()	Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
public Throwable fillInStackTrace()	Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Exception handling in java uses the following Keywords

1. try
2. catch
3. finally
4. throw
5. throws

The try/catch block is used as follows:

```
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
```

```

// exception handler for ExceptionType
}
// optional
finally {
// block of code to be executed after try block ends
}

```

Throwing and catching exceptions

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. The program code that may generate an exception should be placed inside the try/catch block. The syntax for try/catch is depicted as below–

Syntax

```

try {
// Protected code
} catch (ExceptionName e1) {
// Catch block
}

```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception that might be tried to catch. If an exception occurs, then the catch block (or blocks) which follow the try block is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block similar to an argument that is passed into a method parameter.

To illustrate the try-catch blocks the following program is developed.

```

class Exception_example {
public static void main(String args[])
{
int a,b;
try { // monitor a block of code.
a = 0;
b = 10 / a; //raises the arithmetic exception

```

```
System.out.println("Try block.");
}
catch (ArithmeticException e)
{ // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After try/catch block.");
}
}
```

Output:

Division by zero.

After try/catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this multiple exceptions, two or more catch clauses can be specified. Here, each catch block catches different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
class MultiCatch_Example {
    public static void main(String args[]) {
        try {
            int a,b;
            a = args.length;
            System.out.println("a = " + a);
            b = 10 / a; //may cause division-by-zero error
            int arr[] = { 10,20 };
            c[5]=100;
        }
        catch(ArithmeticException e)
        {
```

```

System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}

```

Here is the output generated by the execution of the program in both ways:

```

C:\>java MultiCatch_Example
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch_Example arg1
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:5
After try/catch blocks.

```

While the multiple catch statements is used, it is important to remember that exception subclasses must come before their superclasses. A catch statement which uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. And also, in Java, unreachable code is an error. For example, consider the following program:

```

class MultiCatch_Example {
    public static void main(String args[]) {
    try {
    int a,b;
    a = args.length;
    System.out.println("a = " + a);
    b = 10 / a; //may cause division-by-zero error
    int arr[] = { 10,20 };

```

```
c[5]=100;
}
catch(Exception e) {
System.out.println("Generic Exception catch.");
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

The exceptions such as `ArithmeticException`, and `ArrayIndexOutOfBoundsException` are the subclasses of `Exception` class. The catch statement after the base class catch statement is raising the unreachable code exception.

Nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
}
```

```
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

The following program is an example for Nested try statements.

```
class Nestedtry_Example{
public static void main(String args[]){
try{
try{
System.out.println("division");
int a,b;
a=0;
b =10/a;
}
catch(ArithmeticException e)
{
System.out.println(e);
}
}
try
{
int a[]=new int[5];
a[6]=3;
}
catch(ArrayIndexOutOfBoundsException e)
{
```

```

        System.out.println(e);
    }
    System.out.println("other statement");
}
catch(Exception e)
{
    System.out.println("handed");}
    System.out.println("normal flow.");
}
}

```

Throw keyword

The Java throw keyword is used to explicitly throw an exception. The general form of throw is shown below:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways to obtain a Throwable object:

1. using a parameter in a catch clause
2. creating one with the new operator.

The following program explains the use of throw keyword.

```

public class TestThrow1 {
    static void validate(int age){
        try{
            if(age<18)
                throw new ArithmeticException("not valid");
            else
                System.out.println("welcome to vote");
        }
        Catch(ArithmeticException e)
        {

```

```

        System.out.println("Caught inside ArithmeticExceptions.");
        throw e; // rethrow the exception
    }
}

public static void main(String args[]){
try{
validate(13);
}
Catch(ArithmeticException e)
{
    System.out.println("ReCaught ArithmeticExceptions.");
}
}
}

```

The flow of execution stops immediately after the throw statement and any subsequent statements that are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must be declared using the throws keyword. The throws keyword appears at the end of a method's signature.

The difference between throws and throw keywords is that, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a Remote Exception –

Example

```

import java.io.*;

public class throw_Example1 {
public void function(int a) throws RemoteException {
// Method implementation
throw new RemoteException();
}
}

```

```

    } // Remainder of class definition
}

```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an ArithmeticException –

```

import java.io.*;
public class throw_Example2 {
public void function(int a) throws RemoteException,ArithmeticException {
// Method implementation
}
// Remainder of class definition
}

```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of the occurrence of an Exception. A finally block appears at the end of the catch blocks that follows the below syntax.

Syntax

```

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

```

Example

```

public class Finally_Example {
public static void main(String args[]) {
try {

```

```

int a,b;
    a=0;
    b=10/a;
} catch (ArithmeticException e) {
    System.out.println("Exception thrown :"+ e);
}finally {
    System.out.println("The finally block is executed");
}
}
}

```

Points to remember:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

3.3 BUILT-IN EXCEPTIONS

Built-in exceptions are the exceptions which 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.

Exceptions	Description
Arithmetic Exception	It is thrown when an exceptional condition has occurred in an arithmetic operation.
Array Index Out Of Bound Exception	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.
ClassNotFoundException	This Exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	This Exception is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.
NoSuchMethodException	It is thrown when accessing a method which is not found.
NullPointerException	This exception is raised when referring to the members of a null object. Null represents nothing.
NumberFormatException	This exception is raised when a method could not convert a string into a numeric format.
RuntimeException	This represents any exception which occurs during runtime.
StringIndexOutOfBoundsException	It is thrown by String class methods to indicate that an index is either negative than the size of the string

The following Java program explains NumberFormatException

```

class NumberFormat_Example
{
    public static void main(String args[])
    {
        try {
            int num = Integer.parseInt ("hello");
            System.out.println(num);
        }
        catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
}

```

The following Java program explains StackOverflowError exception.

```

class Example {
    public static void main(String[] args)
    {
        fun1();
    }
}

```

```

public static void fun1()
{
    fun2();
}
public static void fun2()
{
    fun1();
}
}

```

Output:

```

Exception in thread "main" java.lang.StackOverflowError
at Example.fun2(File.java:14)
at Example.fun1(File.java:10)

```

3.4 USER DEFINED EXCEPTION IN JAVA

Java allows the user to create their own exception class which is derived from built-in class Exception. The Exception class inherits all the methods from the class Throwable. The Throwable class is the superclass of all errors and exceptions in the Java language. It contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error.

- The Exception class is defined in java.lang package.
- User defined exception class must inherit Exception class.
- The user defined exception can be thrown using throw keyword.

Syntax:

```

class User_defined_name extends Exception{
    .....
}

```

Some of the methods defined by Throwable are shown in below table.

Methods	Description
Throwable fillInStackTrace()	Fills in the execution stack trace and returns a Throwable object.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.

void printStackTrace()	Displays the stack trace.
String toString()	Returns a String object containing a description of the Exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.

Two commonly used constructors of Exception class are:

- Exception() - Constructs a new exception with null as its detail message.
- Exception(String message) - Constructs a new exception with the specified detail message.

Example:

```
//creating a user-defined exception class derived from Exception class
public class MyException extends Exception
{
public String toString(){           // overriding toString() method
    return "User-Defined Exception";
}
public static void main(String args[]){
    MyException obj= new MyException();
    try
    {
        throw new MyException();    // customized exception is raised
    }

    catch(MyException e)           /*Printing object e makes a call to toString() method which
                                   returns String error message*/
    {
        System.out.println("Exception handled - "+ e);
    }
}
}
```

Sample Output:

Exception handled - User-Defined Exception

In the above example, a custom defined exception class MyException is created by inheriting it from Exception class. The toString() method is overridden to display the customized method on catch. The MyException is raised using the throw keyword.

Example:

Program to create user defined exception that test for odd numbers.

```
import java.util.Scanner;
class OddNumberException extends Exception
{
    OddNumberException()    //default constructor
    {
        super("Odd number exception");
    }
    OddNumberException(String msg) //parameterized constructor
    {
        super(msg);
    }
}
public class UserdefinedExceptionDemo{
    public static void main(String[] args)
    {
        int num;
        Scanner Sc = new Scanner(System.in); // create Scanner object to read input
        System.out.println("Enter a number : ");
        num = Integer.parseInt(Sc.nextLine());
        try
        {
            if(num%2 != 0) // test for odd number
                throw(new OddNumberException()); // raise the exception if number is odd
            else
                System.out.println(num + " is an even number");
        }
    }
}
```

```
        catch(OddNumberException Ex)
        {
            System.out.print("\n\tError : " + Ex.getMessage());
        }
    }
}
```

Sample Output1:

Enter a number : 11

Error : Odd number exception

Sample Output2:

10 is an even number

Odd Number Exception class is derived from the Exception class. To implement user defined exception we need to throw an exception object explicitly. In the above example, If the value of num variable is odd, then the throw keyword will raise the user defined exception and the catch block will get execute.

3.5 CHAINED EXCEPTIONS

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Throwable constructors that supports chained exceptions are:

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Throwable methods that supports chained exceptions are:

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

Example:

```
import java.io.IOException;
public class ChainedException
{
    public static void divide(int a, int b)
    {
        if(b==0)
        {
            ArithmeticException ae = new ArithmeticException("top layer");
            ae.initCause( new IOException("cause") );
            throw ae;
        }
        else
        {
            System.out.println(a/b);
        }
    }
    public static void main(String[] args)
    {
        try {
            divide(5, 0);
        }
        catch(ArithmeticException ae) {
            System.out.println( "caught : " +ae);
            System.out.println("actual cause: "+ae.getCause());
        }
    }
}
```

Sample Output:

```
caught : java.lang.ArithmeticException: top layer
actual cause: java.io.IOException: cause
```

In this example, the top-level exception is `ArithmeticException`. To it is added a cause exception, `IOException`. When the exception is thrown out of `divide()`, it is caught by `main()`. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling `getCause()`.

3.6 STACK TRACE ELEMENT

The `StackTraceElement` class element represents a single stack frame which is a stack trace when an exception occurs. Extracting stack trace from an exception could provide useful information such as class name, method name, file name, and the source-code line number. The `getStackTrace()` method of the `Throwable` class returns an array of `StackTraceElement`s.

StackTraceElement class constructor

`StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber)`

This creates a stack trace element representing the specified execution point.

Stack Trace Element class methods

Method	Description
<code>boolean equals(Object obj)</code>	Returns true if the invoking <code>StackTraceElement</code> is the same as the one passed in <code>obj</code> . Otherwise, it returns false.
<code>String getClassName()</code>	Returns the class name of the execution point
<code>String getFileName()</code>	Returns the filename of the execution point
<code>int getLineNumber()</code>	Returns the source-code line number of the execution point
<code>String getMethodName()</code>	Returns the method name of the execution point
<code>String toString()</code>	Returns the String equivalent of the invoking sequence

Example:

```
public class StackTraceEx {
    public static void main(String[] args) {
        try {
            throw new RuntimeException("go"); //raising an runtime exception
        }
        catch(Exception e){
            System.out.println("Printing stack trace:");
            //create array of stack trace elements
            final StackTraceElement[] stackTrace = e.getStackTrace();
        }
    }
}
```

```

for (StackTraceElement s : stackTrace) {
    System.out.println("\tat " + s.getClassName() + "." + s.getMethodName()
        + "(" + s.getFileName() + ":" + s.getLineNumber() + ")");
    }
}
}
}
}
}

```

Sample Output:

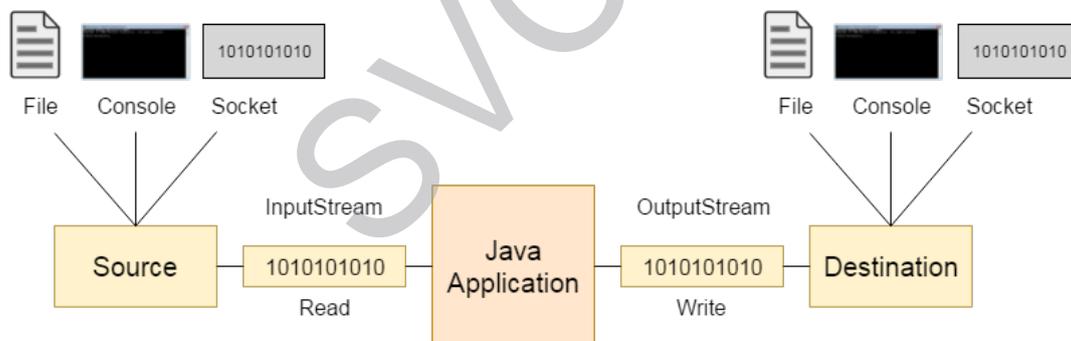
Printing stack trace:

```
at StackTraceEx.main(StackTraceEx.java:5)
```

3.7 INPUT/OUTPUT BASICS

Java I/O (Input and Output) is used to process the input and produce the output. Java uses the concept of stream to make I/O operation fast. All the classes required for input and output operations are declared in java.io package.

A stream can be defined as a sequence of data. The Input Stream is used to read data from a source and the OutputStream is used for writing data to a destination.

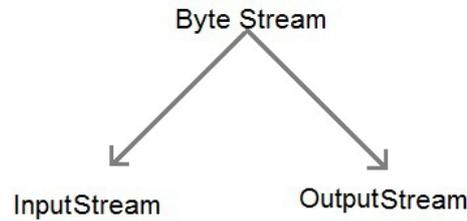


Java defines two types of streams. They are,

1. **Byte Stream** : It is used for handling input and output of 8 bit bytes. The frequently used classes are FileInputStream and FileOutputStream.
2. **Character Stream** : It is used for handling input and output of characters. Character stream uses 16 bit Unicode. The frequently used classes are FileReader and FileWriter.

Byte Stream Classes

The byte stream classes are topped by two abstract classes InputStream and OutputStream.



InputStream class

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

- The Input Stream class is the superclass for all byte-oriented input stream classes.
- All the methods of this class throw an IOException.
- Being an abstract class, the InputStream class cannot be instantiated hence, its subclasses are used

Some of the Input Stream classes are listed below

Class	Description
Buffered Input Stream	Contains methods to read bytes from the buffer (memory area)
Byte Array Input Stream	Contains methods to read bytes from a byte array
Data Input Stream	Contains methods to read Java primitive data types
File Input Stream	Contains methods to read bytes from a file
Filter Input Stream	Contains methods to read bytes from other input streams which it uses as its basic source of data
Object Input Stream	Contains methods to read objects
Piped Input Stream	Contains methods to read from a piped output stream. A piped input stream must be connected to a piped output stream
Sequence Input Stream	Contains methods to concatenate multiple input streams and then read from the combined stream

Some of the useful methods of `InputStream` are listed below.

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

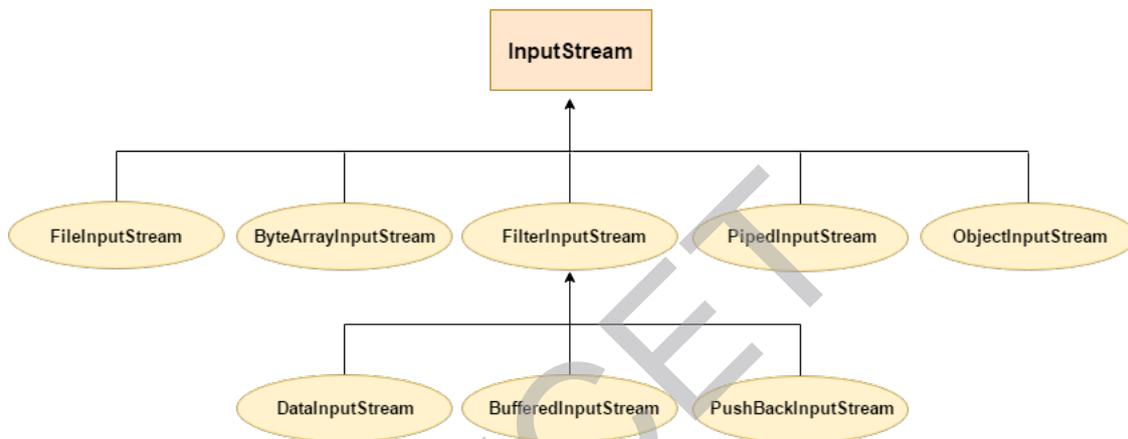


Fig. `InputStream` class Hierarchy

OutputStream class

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

Class	Description
Buffered Output Stream	Contains methods to write bytes into the buffer
Byte Array Output Stream	Contains methods to write bytes into a byte array
Data Output Stream	Contains methods to write Java primitive data types
File Output Stream	Contains methods to write bytes to a file
Filter Output Stream	Contains methods to write to other output streams
Object Output Stream	Contains methods to write objects
Piped Output Stream	Contains methods to write to a piped output stream
Print Stream	Contains methods to print Java primitive data types

Some of the useful methods of OutputStream class are listed below.

Method	Description
public void write(int) throws IO Exception	Write a byte to the current output stream.
public void write(byte[]) throws IO Exception	Write an array of byte to the current output stream.
public void flush() throws IO Exception	Flushes the current output stream.
public void close() throws IO Exception	close the current output stream.

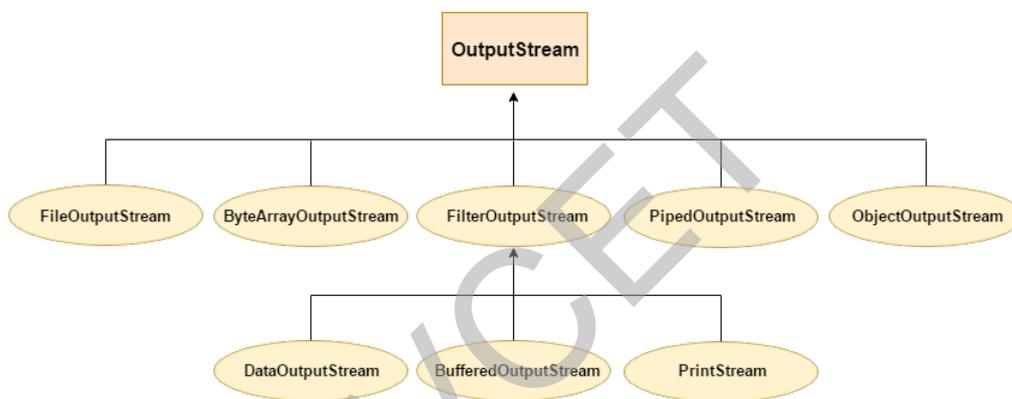
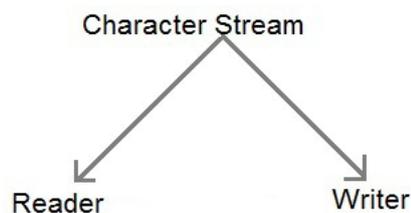


Fig. OutputStream class Hierarchy

Character Stream Classes

The character stream classes are also topped by two abstract classes Reader and Writer.



Some important Character stream reader classes are listed below.

Reader classes are used to read 16-bit unicode characters from the input stream.

- The Reader class is the superclass for all character-oriented input stream classes.
- All the methods of this class throw an IO Exception.
- Being an abstract class, the Reader class cannot be instantiated hence its subclasses are used.

Reader class	Description
BufferedReader	Contains methods to read characters from the buffer
FileReader	Contains methods to read from a file
InputStreamReader	Contains methods to convert bytes to characters
Reader	Abstract class that describes character stream input

The Reader class defines various methods to perform reading operations on data of an input stream. Some of these methods are listed below.

Method	Description
int read()	returns the integral representation of the next available character of input. It returns -1 when end of file is encountered
int read (char buffer [])	attempts to read buffer.length characters into the buffer and returns the total number of characters successfully read. It returns -1 when end of file is encountered
int read (char buffer [], int loc, int nChars)	attempts to read 'nChars' characters into the buffer starting at buffer[loc] and returns the total number of characters successfully read. It returns -1 when end of file is encountered
long skip (long nChars)	skips 'nChars' characters of the input stream and returns the number of actually skipped characters
void close ()	closes the input source. If an attempt is made to read even after closing the stream then it generates IOException

Some important Character stream writer classes are listed below.

Writer classes are used to write 16-bit Unicode characters onto an outputstream.

- The Writer class is the superclass for all character-oriented output stream classes .
- All the methods of this class throw an IOException.
- Being an abstract class, the Writer class cannot be instantiated hence, its subclasses are used.

Writer class	Description
BufferedWriter	Contains methods to write characters to a buffer
FileWriter	Contains methods to write to a file
OutputStreamWriter	Contains methods to convert from bytes to character
PrintWriter	Output stream that contains print() and println()
Writer	Abstract class that describes character stream output

The Writer class defines various methods to perform writing operations on output stream. Some of these methods are listed below.

Method	Description
void write ()	writes data to the output stream
void write (int i)	Writes a single character to the output stream
void write (char buffer [])	writes an array of characters to the output stream
void write(char buffer [],int loc, int nChars)	writes 'n' characters from the buffer starting at buffer [loc] to the output stream
void close ()	closes the output stream. If an attempt is made to perform writing operation even after closing the stream then it generates IOException
void flush ()	flushes the output stream and writes the waiting buffered output characters

Predefined Streams

Java provides the following three standard streams –

- Standard Input – refers to the standard InputStream which is the keyboard by default. This is used to feed the data to user's program and represented as **System.in**.
- Standard Output – refers to the standard OutputStream by default, this is console and represented as **System.out**.
- Standard Error – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

The System class is defined in java.lang package. It contains three predefined stream variables: in, out, err. These are declared as public and static within the system.

3.8 READING CONSOLE INPUT

Reading characters

The read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into char type.

Syntax:

```
int read() throws IOException
```

Example:

Read character from keyboard

```
import java.io.*;
class Main
{
    public static void main( String args[]) throws IOException
```

```
{
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    char c;
    System.out.println("Enter characters, @ to quit");
    do{
        c = (char)br.read();    //Reading character
        System.out.println(c);
    }while(c!='@');
}
```

Sample Output:

Enter characters, @ to quit

abcd23@

a

b

c

d

2

3

@

Example:

Read string from keyboard

The readLine() function with BufferedReader class's object is used to read string from keyboard.

Syntax:

String readLine() throws IOException

Example :

```
import java.io.*;
public class Main{
    public static void main(String args[])throws Exception{
```

```
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
}
}
```

Sample Output :

```
Enter your name
Priya
Welcome Priya
```

3.9 WRITING CONSOLE OUTPUT

- Console output is most easily accomplished with `print()` and `println()`. These methods are defined by the class `PrintStream` (which is the type of object referenced by `System.out`).
- Since `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`.
- So, `write()` can be used to write to the console.

Syntax:

```
void write(int byteval)
```

This method writes to the stream the byte specified by `byteval`.

The following java program uses `write()` to output the character "A" followed by a new-line to the screen:

```
// Demonstrate System.out.write().
class WriteDemo
{
public static void main(String args[])
{
int b;
b = 'A';
```

```
System.out.write(b);
System.out.write('\n');
}
}
```

3.10 THE PRINTWRITER CLASS

- Although using `System.out` to write to the console is acceptable, its use is recommended mostly for debugging purposes or for sample programs.
- For real-world programs, the recommended method of writing to the console when using Java is through a `PrintWriter` stream.
- `PrintWriter` is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize our program.
- `PrintWriter` defines several constructors.

Syntax:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Here,

- `output Stream` is an object of type `OutputStream`
- `flushOnNewline` controls whether Java flushes the output stream every time a `println()` method is called.
- If `flushOnNewline` is true, flushing automatically takes place. If false, flushing is not automatic.
- `PrintWriter` supports the `print()` and `println()` methods for all types including `Object`.
- Thus, we can use these methods in the same way as they have been used with `System.out`.
- If an argument is not a simple type, the `PrintWriter` methods call the object's `toString()` method and then print the result.
- To write to the console by using a `PrintWriter`, specify `System.out` for the output stream and flush the stream after each newline.

For example, the following code creates a `PrintWriter` that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a `PrintWriter` to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo
{
    public static void main(String args[])
    {
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is a string");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Sample Output:

```
This is a string
-7
4.5E-7
```

3.11 READING AND WRITING FILES

In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

Two of the most often-used stream classes are `FileInputStream` and `FileOutputStream`, which create byte streams linked to files.

File Input Stream

This stream is used for reading data from the files. Objects can be created using the keyword `new` and there are several types of constructors available.

The two constructors which can be used to create a `FileInputStream` object:

- i) Following constructor takes a file name as a string to create an input stream object to read the file:

```
InputStream f = new FileInputStream("filename");
```

- ii) Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
```

```
InputStream f = new FileInputStream(f);
```

Methods to read to stream or to do other operations on the stream.

Method	Description
public void close() throws IOException {}	<ul style="list-style-type: none"> Closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
protected void finalize()throws IOException {}	<ul style="list-style-type: none"> Cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
public int read(int r)throws IOException {}	<ul style="list-style-type: none"> Reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
public int read(byte[] r) throws IOException {}	<ul style="list-style-type: none"> Reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
public int available() throws IOException {}	<ul style="list-style-type: none"> Gives the number of bytes that can be read from this file input stream. Returns an int.

File Output Stream

FileOutputStream is used to create a file and write data into it.

The stream would create a file, if it doesn't already exist, before opening it for output.

The two constructors which can be used to create a FileOutputStream object:

- i) Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("filename");
```

- ii) Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
```

```
OutputStream f = new FileOutputStream(f);
```

Methods to write to stream or to do other operations on the stream

Method	Description
public void close() throws IOException {}	<ul style="list-style-type: none"> • Closes the file output stream. • Releases any system resources associated with the file. • Throws an IOException.
protected void finalize()throws IOException {}	<ul style="list-style-type: none"> • Cleans up the connection to the file. • Ensures that the close method of this file output stream is called when there are no more references to this stream. • Throws an IOException.
public void write(int w)throws IOException {}	<ul style="list-style-type: none"> • Writes the specified byte to the output stream.
public void write(byte[] w)	<ul style="list-style-type: none"> • Writes w.length bytes from the mentioned byte array to the OutputStream.

Following code demonstrates the use of InputStream and OutputStream.

```
import java.io.*;
public class FileStreamTest
{
    public static void main(String args[])
    {
        try
        {
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x = 0; x < bWrite.length ; x++)
            {
```

```
        os.write( bWrite[x] ); // writes the bytes
    }
    os.close();
    InputStream is = new FileInputStream("test.txt");
    int size = is.available();
    for(int i = 0; i < size; i++)
    {
        System.out.print((char)is.read() + " ");
    }
        is.close();
    }
catch (IOException e)
{
    System.out.print("Exception");
}
}
```

The above code creates a file named test.txt and writes given numbers in binary format. The same will be displayed as output on the stdout screen.