

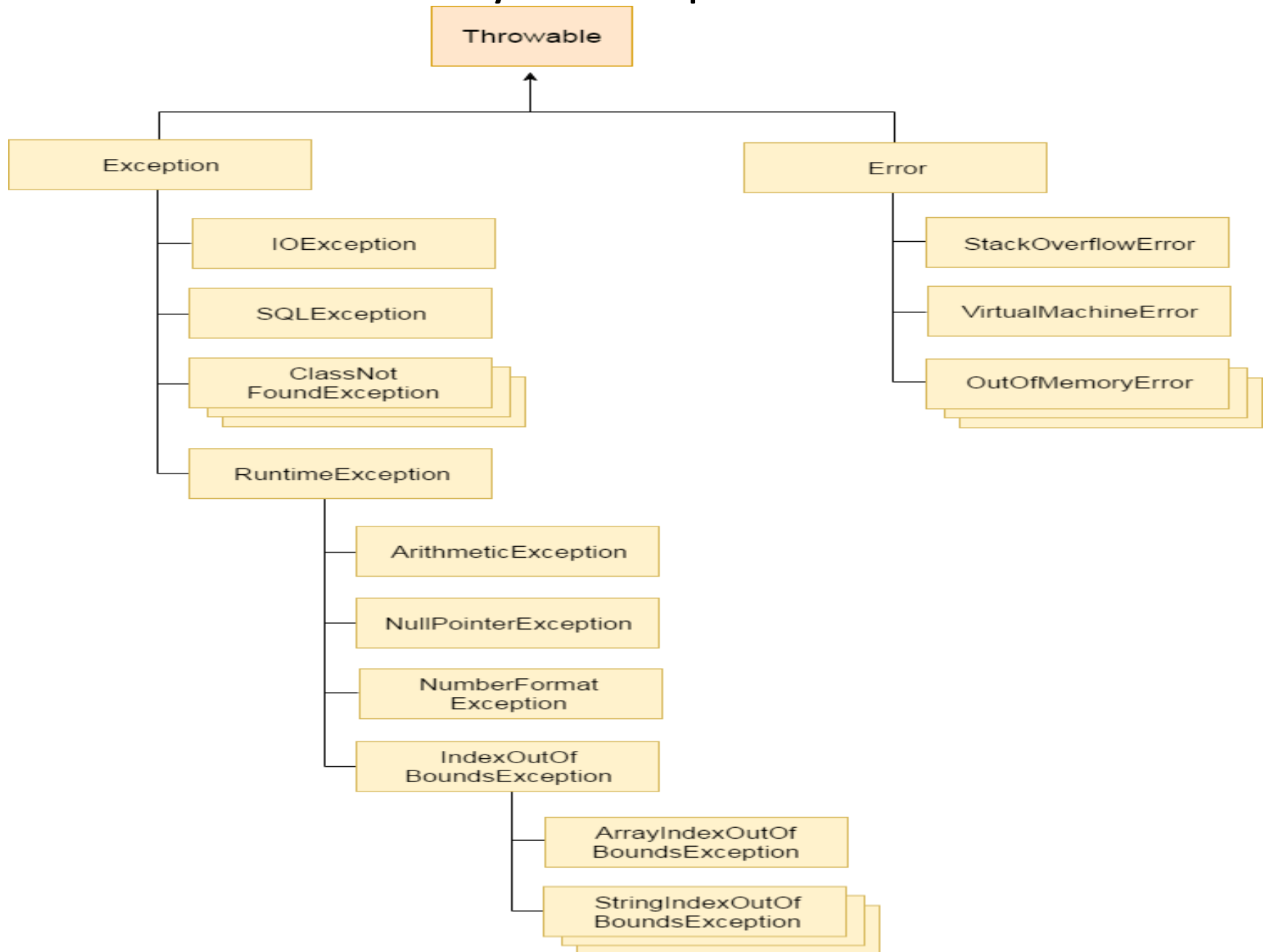
Exception Handling

Exceptions in Java

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- An exception is an abnormal condition that arises in a code sequence at run time.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- In other words, “An exception is a run-time error.”

- An Exception is a run-time error that can be handled programmatically in the application and does not result in abnormal program termination.
- Exception handling is a mechanism that facilitates programmatic handling of run-time errors.
- In java, each run-time error is represented by an object.

Hierarchy of exception classes



- At the root of the class hierarchy, there is a class named *Throwable* which represents the basic features of run-time errors.
- There are two non-abstract sub-classes of Throwable.
 - *Exception* : can be handled
 - *Error* : can't be handled
- *RuntimeException* is the sub-class of Exception.
- Each exception is a run-time error but all run-time errors are not exceptions.

Types of Exceptions: Checked and Unchecked

Checked Exception

- These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- Checked Exceptions are those, that have to be either caught or declared to be thrown in the method in which they are raised.
- Examples: FileNotFoundException, IOException, SQLException, ClassNotFoundException

Example in the next slide:

.

Checked Exception

```
import java.io.*;

public class checked {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\\\Users\\abc\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```

The program doesn't compile, because the function `main()` uses `FileReader()` and `FileReader()` throws a checked exception *FileNotFoundException*. It also uses `readLine()` and `close()` methods, and these methods also throw checked exception *IOException*

Method throwing Checked Exception

```
import java.io.*;
public class checkedcorrect {
    public static void main(String[] args) throws IOException {
        FileReader file = new FileReader("C:\\Users\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        fileInput.close();
    }
}
```

We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

Output:

<First three lines a.txt will be printed>

Java's Checked Exceptions defined in java.lang package

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

IOException

SQLException

FileNotFoundException[Sub-class of IOException]

Unchecked Exceptions

- Unchecked Exceptions are those that are not forced by the compiler either to be caught or to be thrown.
- Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error.
- Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately.
- In Java exceptions/or classes under *Error* and *RuntimeException* classes are unchecked exceptions

Unchecked Exceptions[Subclasses of RuntimeException class defined in java.lang package

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

How a default exception is thrown by JVM??

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
public class Main {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 10;  
        int z = y/x;  
    }  
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at Main.main(Main.java:5) Java Result: 1
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Key points from the previous example

- In the previous example, we haven't supplied any exception handlers of our own.
- So the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Why Exception handling?

- When the default exception handler is provided by the Java run-time system , why Exception Handling?
- Exception Handling is needed because:
 - it allows to fix the error, customize the message .
 - it prevents the program from automatically terminating

Customized Exception Handling

Java uses following keywords for handling exceptions:

- try
- catch
- throw
- throws
- finally



Keywords for Exception Handling

try

- used to execute the statements whose execution may result in an exception.

```
try {  
    Statements whose execution may cause an exception  
}
```

Note: try block is always used either with catch or finally or with both.

catch

- catch is used to define a handler.
- It contains statements that are to be executed when the exception represented by the try block is generated.
- If program executes normally, then the statements of catch block will not be executed.
- If no catch block is found in program, exception is caught by JVM and program is terminated.

Basic example of try-catch

```
public class Main
{
    public static void main(String args[])
    {
        int d, a;
        try
        { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        }
        catch (ArithmeticException e)
        { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

Output:

Division by zero

After catch statement

Key points from the last example

If try catch block is not used, then default exception handler will run, and exception will be handled by Java runtime system automatically, but where the exception arises the program will terminate there only and next part of program will not run.

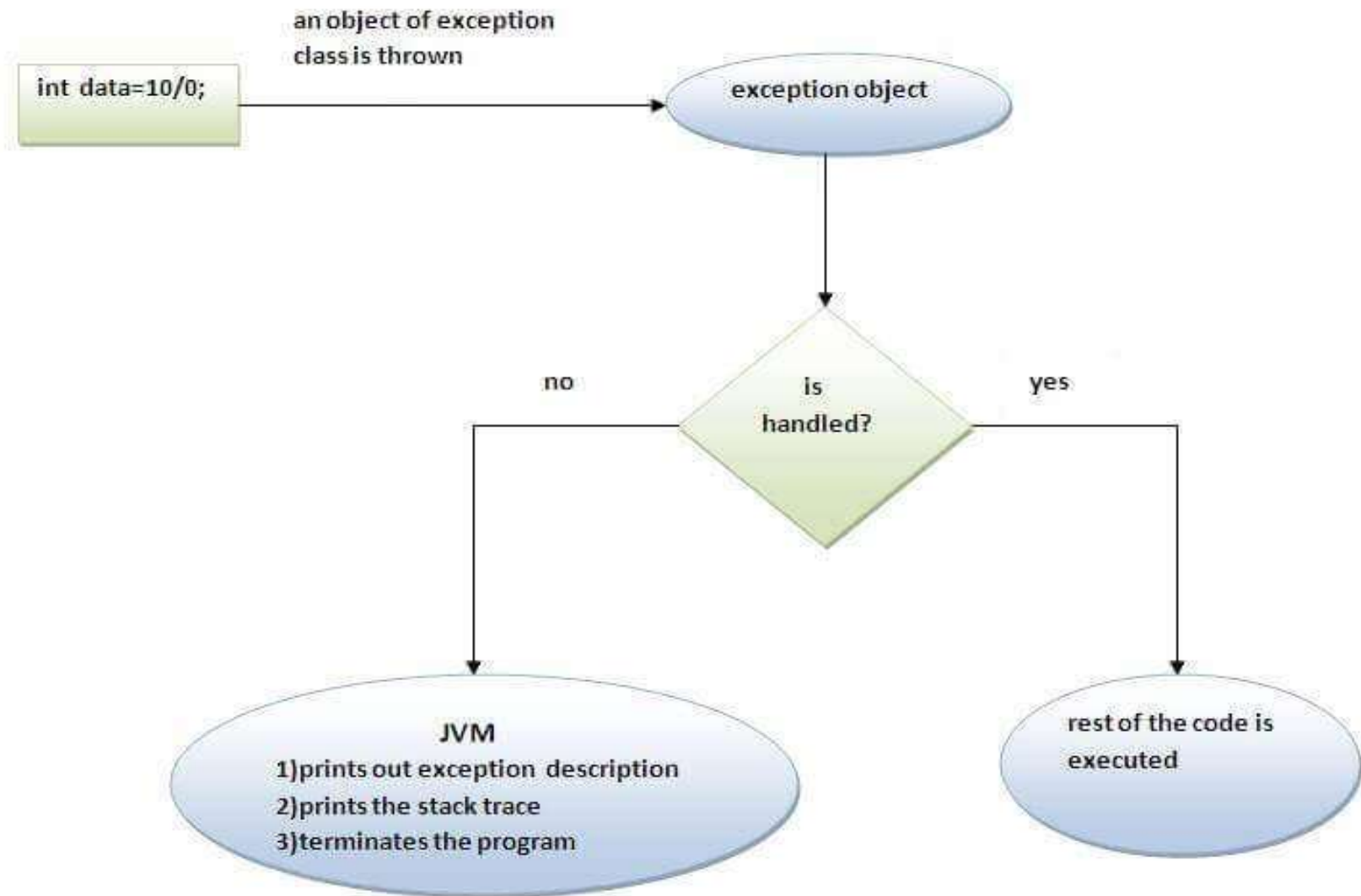
If we remove try catch from last example, then it will be like:

```
public class Main
{
    public static void main(String args[])
    {
        int d, a;
        d = 0;
        a = 42 / d;//Exception will be raised here and message after this will not be printed, as program
        will be terminated
        System.out.println("Statement after expression");
    }
}
```

Exception raised by JVM:

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Main.main(Main.java:7)

Internal working of java 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 type of situation, you can specify two or more **catch** clauses, each catching a 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.
- 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, Example: catch for `ArithmeticException` must come before catch for `Exception`.

Example

```
public class Main {  
    public static void main(String[] args) {  
        try{  
            String x="ABC";  
            int a[]=new int[5];  
            a[0]=30/0;//It will raise ArithmeticException  
            //System.out.println(a[7]);//It will raise ArrayIndexOutOfBoundsException  
            //System.out.println(Integer.parseInt(x));//It can raise NumberFormatException, but its specific catch is not  
written so catch with Exception class will work  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Multi-catch feature introduced from JDK 7 onwards

// Demonstrate the multi-catch feature.

```
public class Main {  
    public static void main(String args[]) {  
        int a=10, b=0;  
        int vals[] = { 1, 2, 3 };  
        try {  
            int result = a / b; // generate an ArithmeticException  
            //vals[10] = 19; // generate an ArrayIndexOutOfBoundsException  
            // This catch clause catches both exceptions.  
        } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception caught: " + e);  
        }  
        System.out.println("After multi-catch.");  
    }  
}
```

Output:

Exception caught: java.lang.ArithmeticException: / by zero

After multi-catch.

Example, where compilation error could come

```
public class Main {  
    public static void main(String[] args) {  
        try{  
            String x="ABC";  
            int a[]=new int[5];  
            a[0]=30/0;//It will raise ArithmeticException  
            //System.out.println(a[7]);//It will raise ArrayIndexOutOfBoundsException  
            //System.out.println(Integer.parseInt(x));//It can raise NumberFormatException, but its  
specific catch is not written so catch with Exception class will work  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Here compilation error will come, as general catch handler with super class is written first and then the subclass catches are written

Nested try statements

- The try block within a try block is known as nested try block in java.
- 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.

Syntax:

```
....  
try  
{  
    statement 1;  
    statement 2;  
    try  
    {  
        statement 1;  
        statement 2;  
    }  
    catch(Exception e)  
    {  
    }  
}  
catch(Exception e)  
{  
}  
....
```

Example

```
public class Main{
    public static void main(String args[]){
        try{
            int arr[]=new int[5];
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e){System.out.println(e+" inner");}

            try{
                int a[]=new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e+" inner");}

            System.out.println(arr[6]);
        }catch(Exception e){System.out.println(e+" outer");}
        System.out.println("After try catch");
    }
}
```

Output:

```
going to divide
java.lang.ArithmeticException: / by zero inner
java.lang.ArrayIndexOutOfBoundsException: 5 inner
java.lang.ArrayIndexOutOfBoundsException: 6 outer
After try catch
```

Defining Generalized Exception Handler

- A generalized exception handler is one that can handle the exceptions of all types.
- If a class has a generalized as well as specific exception handler, then the generalized exception handler must be the last one.

Example

```
public class Main{  
    public static void main(String args[]){  
        try {  
            String x=null;  
            //int c = 12/0;//It will raise ArithmeticException  
            //System.out.println(c);  
            System.out.println(x.length());//It will raise NullPointerException  
        }  
        catch (Throwable e) //Generalized exception handler, which can handle both  
            exceptions[one at one time]  
        {  
            System.out.println(e);  
        }  
    }  
}
```

Using throw keyword

The *throw* keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The *throw* keyword is mainly used to throw custom exceptions.

Syntax:

throw Instance

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception.

If it finds a match, control is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program.

```
public class Main
{
    static void demo()
    {
        try {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside demo.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String args[])
    {
        try {
            demo();
        }
        catch(NullPointerException e)
        {
            System.out.println("Recaught: " + e);
        }
    }
}
```

Output:

Caught inside demo.

Recaught: java.lang.NullPointerException: demo

Using throws keyword

throws

- A throws clause lists the types of exceptions that a method might throw.

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

- This is necessary for all exceptions, except those of type Error or Runtime Exception, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Example of throws

```
import java.io.*;
public class abc {
    public static void findFile() throws IOException {
        // code that may produce IOException
        File newFile=new File("test.txt");
        FileInputStream stream=new FileInputStream(newFile);
    }
    public static void main(String[] args) {
        findFile();//try catch block must be used as method throws checked exception
    }
}
```

Output:

```
C:\Codes>javac abc.java
```

```
abc.java:9: error: unreported exception IOException; must be caught or declared to be
thrown
```

```
    findFile();
```

```
        ^
```

```
1 error
```


Enclosing method call inside try-catch which throws the exception

```
import java.io.*;
public class abc {
    public static void findFile() throws IOException {
        // code that may produce IOException
        File newFile=new File("test.txt");
        FileInputStream stream=new FileInputStream(newFile);
    }
    public static void main(String[] args) {
        try{
            findFile();
        } catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Output:

java.io.FileNotFoundException: test.txt (No such file or directory)

throw vs throws

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Throw is followed by an instance.	Throws is followed by class.
3)	Throw is used within the method.	Throws is used with the method signature.
4)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Finally

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Example 1

```
public class TestFinallyBlock2{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(ArithmeticException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero
 finally block is always executed
 rest of the code...

Example 2

```
// Demonstrate finally.
public class Main {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }
    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
}
```

```
// Execute a try block normally.
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}
public static void main(String args[]) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Exception caught");
    }
    procB();
    procC();
}
}
```

Output:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Propagation of Exceptions

- If an exception is not caught and handled where it is thrown, the control is passed to the method that has invoked the method where the exception was thrown.
- The propagation continues until the exception is caught, or the control passes to the main method, which terminates the program and produces an error message.

Example

```
public class Main{
    public void first(){int data=50/0; }
    public void second(){first();}
    public void third(){try{second();}
                                catch(Exception e){
System.out.println("Exception occurred");}
    }
    public static void main(String [] args){
        Main ob = new Main();
        ob.third();
        System.out.println("Thank You");
    }
}
```

Output:

Exception occurred

Thank You

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.

- **ArithmeticException**

It is thrown when an exceptional condition has occurred in an arithmetic operation.

- **ArrayIndexOutOfBoundsException**

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

- **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 or greater than the size of the string

Few Examples

ArithmeticException

```
// Java program to demonstrate ArithmeticException  
class Main
```

```
{  
    public static void main(String args[])  
    {  
        try {  
            int a = 30, b = 0;  
            int c = a/b; // cannot divide by zero  
            System.out.println ("Result = " + c);  
        }  
        catch(ArithmeticException e) {  
            System.out.println ("Can't divide a number by 0");  
        }  
  
        System.out.println("Code will execute after the exception handled");  
    }  
}
```

Output:

Can't divide a number by 0

Code will execute after the exception handled

If we use try catch block to handle the exceptions then code will be executed even after the exception is thrown which is not possible without handling the exception. If we do not handle the exception then JVM will give built in exception message and terminate the program.

NullPointerException

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

ArrayIndexOutOfBoundsException Exception

```
public class Main {  
    public static void main(String args[]) {  
        int a[]=new int[5];  
        try {  
            System.out.println(a[7]);  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("After try catch...");  
    }  
}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 7  
After try catch...
```

StringIndexOutOfBoundsException

```
public class Main {  
    public static void main(String args[]) {  
        String x="Testing";  
        try {  
            System.out.println(x.charAt(7));  
        }  
        catch(StringIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("After try catch...");  
    }  
}
```

Output:

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7  
After try catch...
```

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined Exceptions'. Following steps are followed for the creation of user-defined Exception.

Step 1:

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

Class MyException extends Exception

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

```
MyException(){} 
```

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

```
MyException(String str) { super(str); } 
```

Step 2:

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

```
MyException me = new MyException("Exception details");  
throw me;
```

Example

```
// Java program to demonstrate user defined  
exception
```

```
class MyException extends Exception
```

```
{
```

```
private static int accno[] = {1001, 1002, 1003,  
1004,1005};
```

```
private static String name[] = {"Nish", "Shubh",  
"Sush", "Abhi", "Akash"};
```

```
private static double bal[] = {10000.00,  
12000.00, 5600.0, 999.00, 1100.55};
```

```
MyException() { }
```

```
MyException(String str) { super(str);  
}
```

Output:

```
1001  Nish  10000.0
```

```
1002  Shubh 12000.0
```

```
1003  Sush   5600.0
```

```
1004  Abhi   999.0
```

```
Main: Balance is less than 1000
```

```
at Main.main(Main.java:18)
```

```
Program is running fine after handling the  
exception.
```

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

```
{
```

```
try {
```

```
for (int i = 0; i < 5 ; i++)
```

```
{
```

```
System.out.println(accno[i] + "\t" + name[i] + "\t" + bal[i]);
```

```
if (bal[i] < 1000)
```

```
{
```

```
MyException me = new MyException("Balance is less  
than 1000");
```

```
throw me;
```

```
}
```

```
}
```

```
}
```

```
catch (MyException e) {
```

```
e.printStackTrace();
```

```
}
```

```
System.out.println("Program is running fine after handling  
the exception.");
```

```
}
```

```
}
```

Methods of Throwable class

printStackTrace():

It is a method of Java's **throwable class** which prints the throwable along with other details like the line number and class name where the exception occurred.

getMessage():

The `getMessage()` method of Java Throwable class is used to get a detailed message of the Throwable[or Exception thrown].

Example 1

```
public class Main
{
    public static void main(String[] args) {
        try
        {
            int a=12,b=0;
            System.out.println(a/b);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:8)
/ by zero
```

Example 2

```
public class Main
{
    public static void main(String[] args) {
        try
        {
            throw new ArithmeticException("Wrong denominator");
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
java.lang.ArithmeticException: Wrong denominator
java.lang.ArithmeticException: Wrong denominator
    at Main.main(Main.java:7)
Wrong denominator
```

Example 3

```
public class Main
{
    public static void main(String[] args) {
        try
        {
            throw new ArithmeticException();
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
java.lang.ArithmeticException
java.lang.ArithmeticException
    at Main.main(Main.java:7)
null
```

Q1

Predict the output of following Java program

```
public class Main {  
    public static void main(String args[]) {  
        try {  
            throw 10;  
        }  
        catch(int e) {  
            System.out.println("Got the Exception " + e);  
        }  
    }  
}
```

- A. Got the Exception 10
- B. Got the Exception 0
- C. Compiler Error
- D. Blank output

Q2:What will be the output of following code?

```
class Test extends Exception { }  
public class Main {  
    public static void main(String args[]) {  
        try {  
            throw new Test();  
        }  
        catch(Test t) {  
            System.out.println("Got the Test Exception");  
        }  
        finally {  
            System.out.println("Inside finally block ");  
        }  
    }  
}
```

- A. Got the Test Exception
 Inside finally block
- B. Got the Test Exception
- C. Inside finally block
- D. Compiler Error

Q3:What will be the output of following Java program?

```
public class Main {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 10;  
        int z = y/x;  
    }  
}
```

- A. Compiler Error
- B. Compiles and runs fine
- C. Compiles fine but throws ArithmeticException exception
- D. None of these

What will be the output of following code?[Q4]

```
public class Test
{
    public static void main (String[] args)
    {
        try
        {
            int a = 0;
            System.out.println ("a = " + a);
            int b = 20 / a;
            System.out.println ("b = " + b);
        }

        catch(ArithmeticException e)
        {
            System.out.println ("Divide by zero error");
        }

        finally
        {
            System.out.println ("inside the finally block");
        }
    }
}
```

- A. Compile error
- B. Divide by zero error
- C. a = 0
Divide by zero error
inside the finally block
- D. a = 0

What will be the output of following code?[Q5]

```
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            int a[]={1, 2, 3, 4};
            for (int i = 1; i <= 4; i++)
            {
                System.out.println ("a[" + i + "]= " + a[i] + "n");
            }
        }
        catch (Exception e)
        {
            System.out.println ("error = " + e);
        }

        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println ("ArrayIndexOutOfBoundsException");
        }
    }
}
```

- A. Compiler error
- B. Run time error
- C. ArrayIndexOutOfBoundsException
- D. Array elements are printed

Predict the output of the following program.[Q6]

```
public class Test
{   int count = 0;
    void A() throws Exception
    {
        try
        {
            count++;
            try
            {
                count++;

                try
                {
                    count++;
                    throw new Exception();
                }
                catch(Exception ex)
                {
                    count++;
                    throw new Exception();
                }
            }
            catch(Exception ex)
            {
                count++;
            }
        }
        catch(Exception ex)
        {
            count++;
        }
    }
}
```

```
void display()
{
    System.out.println(count);
}

public static void main(String[] args)
throws Exception
{
    Test obj = new Test();
    obj.A();
    obj.display();
}
```

- A. 4
- B. 5
- C. 6
- D. Compilation error

Which of these is a super class of all errors and exceptions in the Java language?[Q7]

A. RuntimeExceptions

B. Throwable

C. Catchable

D. None of the above

```
public class Test
```

```
{
```

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

```
    {
```

```
        try
```

```
        {
```

```
            System.out.printf("1");
```

```
            int sum = 9 / 0;
```

```
            System.out.printf("2");
```

```
        }
```

```
        catch(ArithmeticException e)
```

```
        {
```

```
            System.out.printf("3");
```

```
        }
```

```
        catch(Exception e)
```

```
        {
```

```
            System.out.printf("4");
```

```
        }
```

```
        finally
```

```
        {
```

```
            System.out.printf("5");
```

```
        }
```

```
    }
```

```
}
```

a) 1325

b) 1345

c) 1342

d) 135

Output??[Q8]

Output??[Q9]

```
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.printf("1");
            int data = 5 / 0;
        }
        catch(ArithmeticException e)
        {
            System.out.printf("2");
            System.exit(0);
        }
        finally
        {
            System.out.printf("3");
        }
        System.out.printf("4");
    }
}
```

- a) 12
- b) 1234
- c) 124
- d) 123


```
public class Test
```

```
{
```

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

```
    {
```

```
        try
```

```
        {
```

```
            System.out.printf("1");
```

```
            int data = 5 / 0;
```

```
        }
```

```
        catch(ArithmeticException e)
```

```
        {
```

```
            Throwable obj = new Throwable("Sample");
```

```
            try
```

```
            {
```

```
                throw obj;
```

```
            }
```

```
            catch (Throwable e1)
```

```
            {
```

```
                System.out.printf("8");
```

```
            }
```

```
        }
```

```
        finally
```

```
        {
```

```
            System.out.printf("3");
```

```
        }
```

```
        System.out.printf("4");
```

```
    }
```

```
}
```

a) Compilation error

b) Runtime error

c) 1834

d) 134

Output??[Q10]

Output??[Q11]

```
import java.io.EOFException;
import java.io.IOException;

public class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.printf("1");
            int value = 10 / 0;
            throw new IOException();
        }
        catch(EOFException e)
        {
            System.out.printf("2");
        }
        catch(ArithmeticException e)
        {
            System.out.printf("3");
        }
        catch(NullPointerException e)
        {
            System.out.printf("4");
        }
        catch(IOException e)
        {
            System.out.printf("5");
        }
        catch(Exception e)
        {
            System.out.printf("6");
        }
    }
}
```

a) 1346

b) 136726

c) 136

d) 13

Output??[Q12]

```
public class Main
{
    public static void main(String[] args) {
        try
        {
            System.out.print("Hello" + " " + 1 / 0);
        }
        catch(ArithmeticException e)
        {
            System.out.print("World");
        }
    }
}
```

- a) Hello
- b) World
- c) HelloWorld
- d) Hello World

Output??[Q13]

What will be the output of the following Java program?

```
public class exception_handling
{
    public static void main(String args[])
    {
        try
        {
            int a, b;
            b = 0;
            a = 5 / b;
            System.out.print("A");
        }
        catch(ArithmeticException e)
        {
            System.out.print("B");
        }
    }
}
```

- a) A
- b) B
- c) Compilation Error
- d) Runtime Error

Output??[Q14]

// Demonstrate the multi-catch feature.

```
public class Main {  
    public static void main(String args[]) {  
        int sum=10;  
        try  
        {  
            int i;  
            for (i = -1; i < 3 ;++i)  
                sum = (sum / i);  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.print("0 ");  
        }  
        System.out.print(sum);  
    }  
}
```

- A. 0 -10
- B. 0 10
- C. Compile time error
- D. 0

Programming in Java

Topic: Assertion



Contents

- ▶ Introduction
- ▶ Assertion
- ▶ Using Assertion
- ▶ How assertion works
- ▶ Benefit of using Assertion
- ▶ Important Points



Introduction

- ▶ An assertion is a statement in Java that enables you to test your assumptions about the program.
- ▶ Each assertion contains a boolean expression that you believe will be true when the assertion executes.
- ▶ If it is not true, the system will throw a runtime error (`java.lang.AssertionError`).



Example

For example, if we write a method that calculates the interest of any account, we might assert that the roi or amount is positive.

```
double calculateInterest(int amt, double roi, int years)
{
    //Using assertion
    assert amt>0;
    double interest = (amt*roi*years)/100;
    return interest;
}
```



Assertion

- ▶ The assertion statement has two forms.

- **assert** *Expression1*;

Here, Expression1 is a boolean expression. When the system runs the assertion, it evaluates Expression1 and if it is false throws an AssertionError with no detail message.

- **assert** *Expression1 : Expression2*;

Here, Expression1 is a boolean expression and Expression2 is an expression that has a value and it cannot be an invocation of a method that is declared void. If Expression1 fails then Expression2 result will be displayed.



Example

```
public class pqr {  
    public static void main(String[] args) {  
        // get a number in the first argument  
        int num = Integer.parseInt(args[0]);  
        //assert num <= 10; // stops if number > 10  
        assert num <= 10 : "Number is greater than 10";  
        System.out.println("All assumptions are correct");  
    }  
}
```

Output: Suppose we have passed 11, then output will be:

Exception in thread "main" java.lang.AssertionError: Number is greater than
10

at pqr.main(pqr.java:6)

On passing value less than or equal to 10, output will be:

All assumptions are correct



Using Assertion

Assertion will not work directly because assertion is disabled by default.

To enable the assertion, -ea or -enableassertions switch of java must be used.

Compile: `javac AssertionExample.java`

Run: `java -ea AssertionExample`



How Assertion Works?

In assertion a `BooleanExpression` is used and if boolean expression is false then java throws `AssertionError` and the program terminates.

Second form of assert keyword "`assert booleanExpression : errorMessage`" is more useful and provides a mechanism to pass additional data when Java assertion fails and java throws `AssertionError`.



Need of Assertion

```
class pqr{  
void remainder(int i)  
{  
if(i%3==0)  
System.out.println("Divisible by 3");  
else if(i%3==1)  
System.out.println("Remainder 1");  
else  
{  
System.out.println("Remainder 2");  
}  
}  
public static void main(String [] args)  
  
{  
new pqr().remainder(8);  
}
```

Output:

Remainder 2

What About it?

```
public class pqr
{
    public static void main(String [] args)
    {
        new pqr().remainder(-8);
    }
}
```



Need of Assertion

```
class pqr{  
void remainder(int i)  
{  
if(i%3==0)  
System.out.println("Divisible by 3");  
else if(i%3==1)  
System.out.println("Remainder 1");  
else  
{  
System.out.println("Remainder 2");  
}  
}  
public static void main(String [] args)  
{  
new pqr().remainder(-8);  
}  
}
```

Here output will be:

Remainder 2, which is not correct, it must be Remainder -2, so assertion could be used[Described in next slide]



Previous example using assertion

```
class pqr{  
    void remainder(int i)  
    {  
        if(i%3==0)  
            System.out.println("Divisible by 3");  
        else if(i%3==1)  
            System.out.println("Remainder 1");  
        else  
        {  
            assert i%3==2:"Remainder is negative, not  
            matching with any case";  
            System.out.println("Remainder 2");  
        }  
    }  
    public static void main(String [] args)  
    {  
        new pqr().remainder(-8);  
    }  
}
```

Output:

Exception in thread "main"
java.lang.AssertionError:
Remainder is negative, not
matching with any case
at
pqr.remainder(pqr.java:10)
at pqr.main(pqr.java:16)

Why Assertion?

- ▶ Using assertion helps to detect bug early in development cycle which is very cost effective.
- ▶ Assertion in Java makes debugging easier as `AssertionError` is a best kind of error while debugging because its clearly tell source and reason of error.
- ▶ Assertion is great for data validation if by any chance your function is getting incorrect data your program will fail with `AssertionError`.




Important

- ▶ Assertion is introduced in JDK 1.4 and implemented using assert keyword in java.
 - ▶ Assertion can be enabled at run time by using switch `-ea` or `-enableassertion`
 - ▶ Assertion does not replace Exception but compliments it.
 - ▶ Do not use assertion to validate arguments or parameters of public method.(It would be better to throw exceptions rather than using assertion for checking parameter values, as assertions could be disabled also, which can compromise the parameter check sometimes)
-

Q1

What will happen when you compile and run the following code with assertion enabled?

```
public class Test{  
    public static void main(String[] args){  
  
        int age = 20;  
        assert age > 20 : getMessage();  
        System.out.println("Valid");  
  
    }  
    private static void getMessage() {  
        System.out.println("Not valid");  
    }  
}
```

- A. The code will not compile
- B. The code will compile but will throw `AssertionError` when executed
- C. The code will compile and print `Not Valid`
-  D. The code will compile and print `Valid`

Q2

What will happen when you compile and run the following code with assertion enabled?

```
public class Test{  
  
    public static void main(String[] args){  
        assert false;  
        System.out.println("True");  
    }  
}
```

- A. The code will not compile due to unreachable code since false is hard coded in assert statement
 - B. The code will compile but will throw `AssertionError` when executed
 - C. The code will compile and print true
 - D. None of the above
-



Q3

Output?

```
public class Test{  
  
    public static void main(String[] args){  
  
        boolean b = false;  
        assert b = true;  
        System.out.println("Hi");  
    }  
}
```

- A. Hi
- B. The code will not compile
- C. The code will compile but will throw `AssertionError` when executed
- D. None of these



Q4(Output??)

```
public class pqr{  
    public static void main(String[] args){  
        int i = 10;  
        int j = 24;  
        int k = 34;  
        assert i + j >= k : k--;  
        System.out.println(i + j + k);  
    }  
}
```

- A. 67
 - B. 68
 - C. Compiles fine but gives AssertionError
 - D. None of these
-



Java Input Output

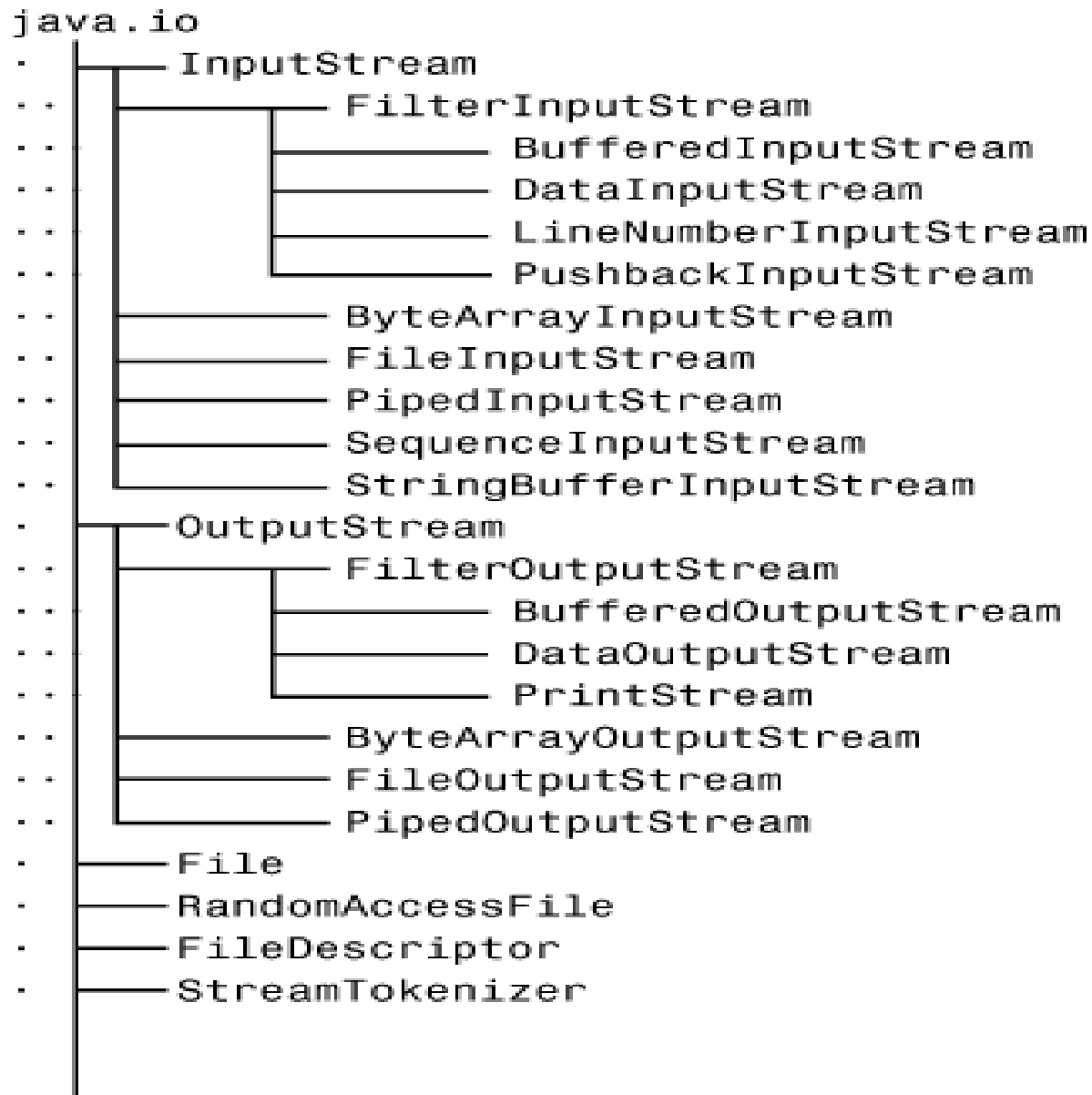
Java I/O

Java 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.

java.io



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.

```
System.out.println("simple message");
```

```
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
```

```
System.out.println((char)i);//will print the character
```

OutputStream vs InputStream

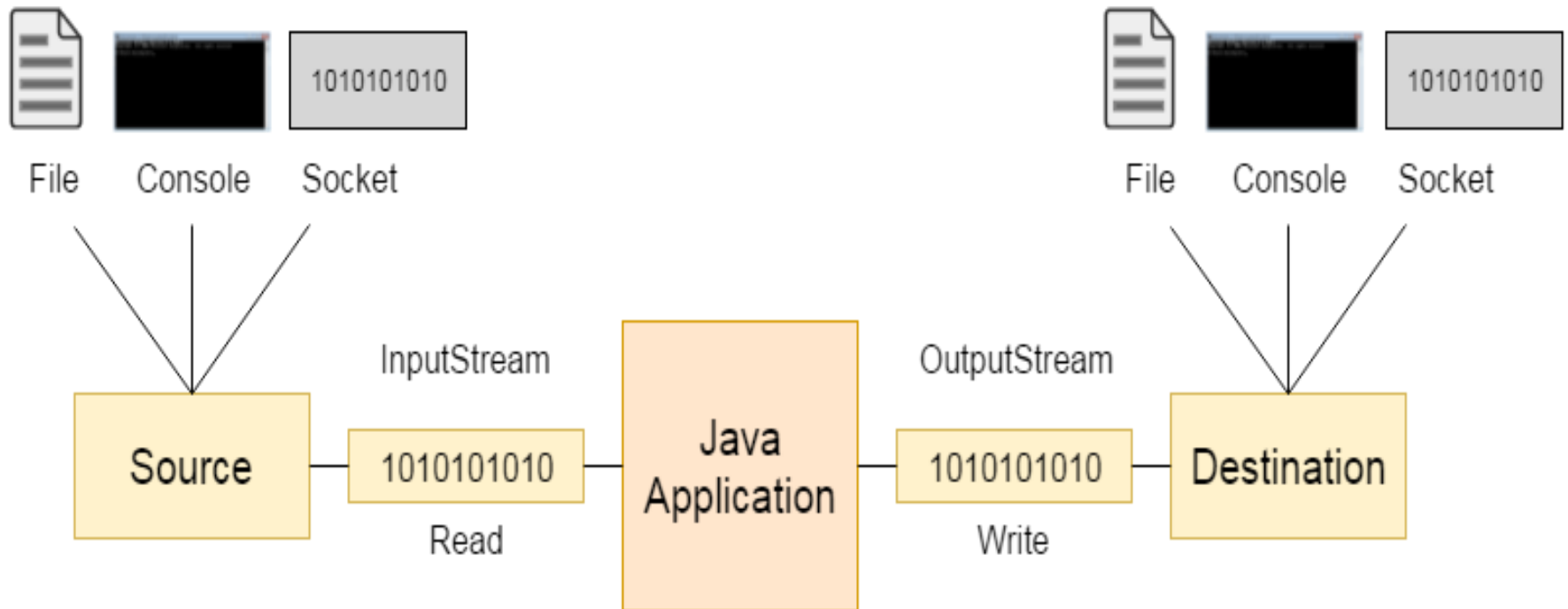
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.





File Class

- The File class provides the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.
- An absolute file name (or full name) contains a file name with its complete path and drive letter.
- For example, `c:\book\Welcome.java`
- A relative file name is in relation to the current working directory.
- The complete directory path for a relative file name is omitted.
- For example, `Welcome.java`

Method and constructor for File class

Constructor:

`File(String path_name)`

Creates a File object for the specified path name.
The path name may be a directory or a file.

Methods of File class

Methods:

- `boolean isFile()`
- `boolean isDirectory()`
- `boolean isHidden()`
- `boolean exists()`
- `boolean canRead()`
- `boolean canWrite()`
- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `long lastModified()`
- `long length()`
- `boolean delete()`
- `boolean renameTo(File f)`
- `File[] listFiles()`

Example

```
import java.io.*;

public class FileClass
{
    public static void main(String[] args) {
        File ref1=new
File("C:\\Users\\Salil\\Desktop\\Java\\UNIT-
5\\Codes\\JavaIO");

        System.out.println(ref1.isFile());//It will give
false as JavaIO is not a file

        System.out.println(ref1.isDirectory());//It will give true
as JavaIO is a folder/or directory

        File ref2=new
File("C:\\Users\\Salil\\Desktop\\Java\\UNIT-
5\\Codes\\JavaIO\\abc.txt");

        System.out.println(ref2.exists());
        System.out.println(ref2.getName());
        System.out.println(ref2.getPath());

        File ref3=new
File("C:\\Users\\Salil\\Desktop\\Java\\UNIT-
5\\Codes\\JavaIO\\abc2.txt");

        File ref4=new
File("C:\\Users\\Salil\\Desktop\\Java\\UNIT-
5\\Codes\\JavaIO\\newname.txt");

        System.out.println(ref3.isHidden());
        System.out.println(ref3.canRead());
        System.out.println(ref3.canWrite());
```

```
File ref5=new File("abc.txt");

        System.out.println(ref5.getPath());
        System.out.println(ref5.getAbsolutePath());
        System.out.println("Last modified on " + new
java.util.Date(ref5.lastModified()));

        System.out.println("Length:"+ref5.length());
        /*File ref6=new File("abc3.txt");
        if(ref6.delete())
        System.out.println("File deleted successfully");
        else
        System.out.println("File does not exists");*/
        File x[]=ref1.listFiles();
        for(File var:x)
        System.out.println(var);
        boolean flag = ref5.renameTo(ref4);
        if (flag == true) {
            System.out.println("File Successfully Renamed");
        }
        else {
            System.out.println("Operation Failed");
        }
    }
}
```

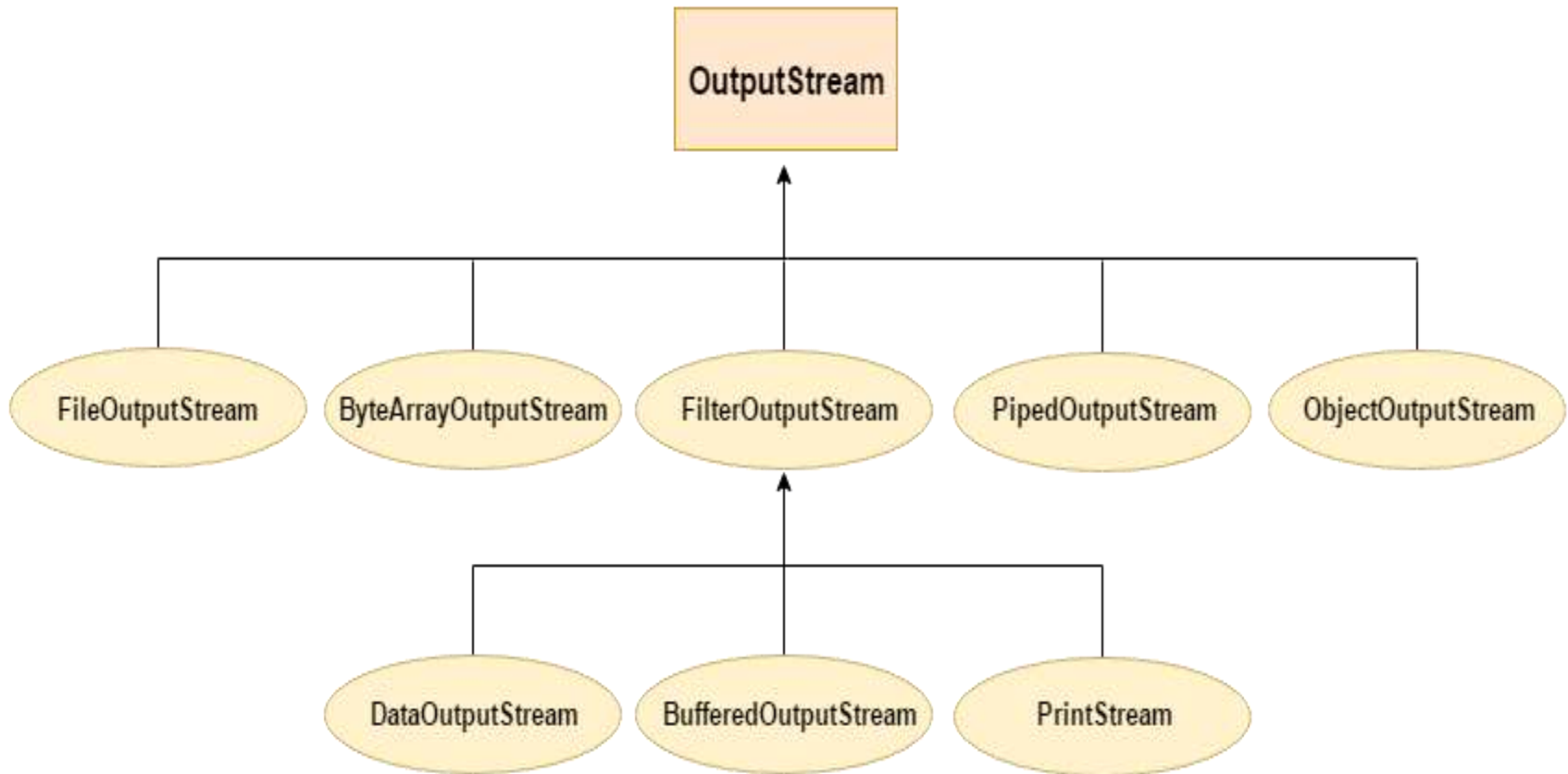
OutputStream class

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

OutputStream Hierarchy



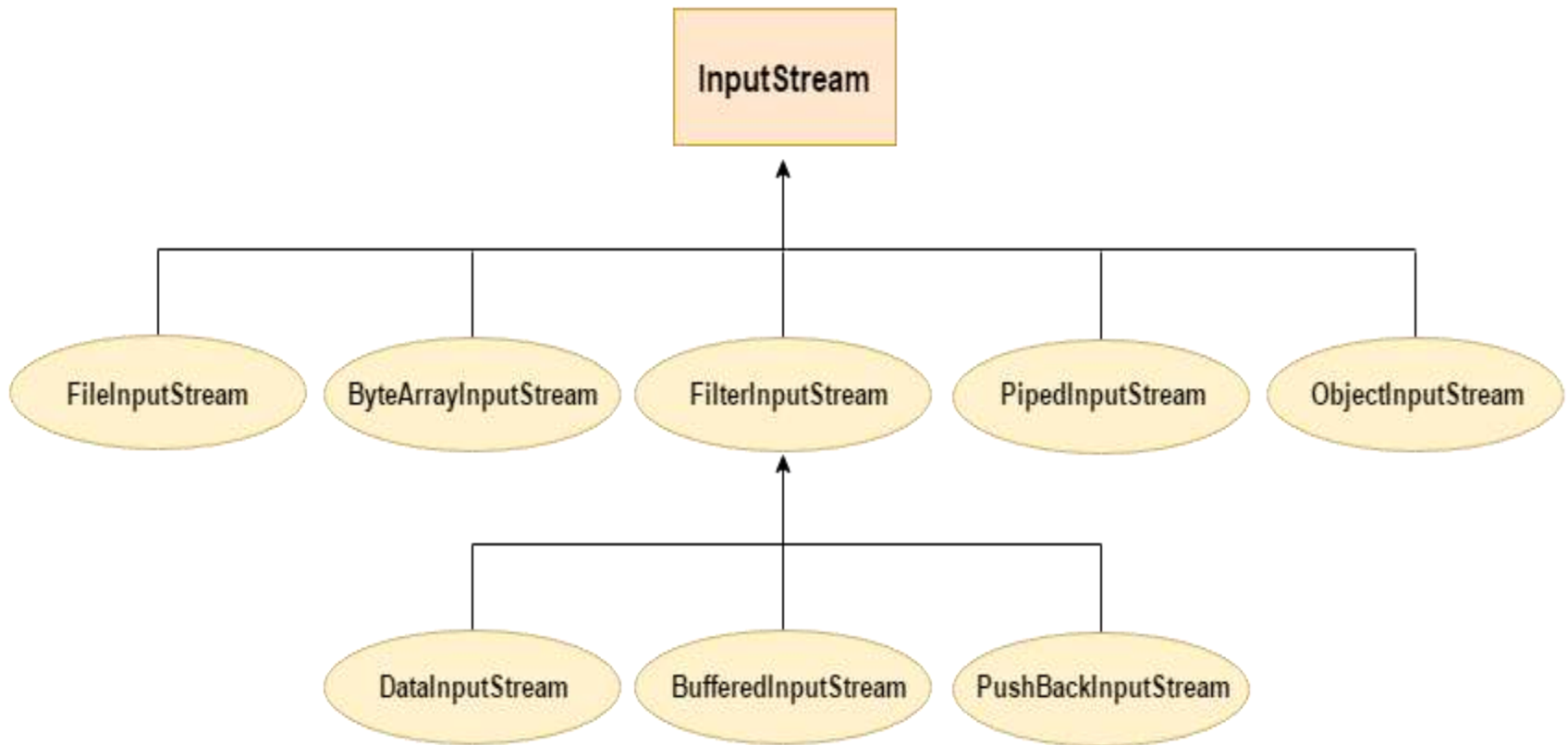
InputStream class

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

InputStream Hierarchy



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.

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

```
public class FileOutputStream extends OutputStream
```


Methods of Java FileOutputStream Class

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	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

```
import java.io.FileOutputStream;
public class Main{
    public static void main(String args[]){
        try{
            FileOutputStream fout=new
            FileOutputStream("D:\\testout.txt");
            fout.write(65);
            fout.close();
            System.out.println("success...");
        }catch(Exception e){ System.out.println(e);}
    }
}
```

Java FileOutputStream example 2: write string

```
import java.io.FileOutputStream;
public class Main {
    public static void main(String args[]){
        try{
            FileOutputStream fout=new FileOutputStream("testout.txt");
            String s="Welcome to file handling.";
            byte b[]=s.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
        }catch(Exception e){System.out.println(e);}
    }
}
```

Appending data through FileOutputStream

```
import java.io.FileOutputStream;
public class Main{
    public static void main(String args[]){
        try{
            String textToAppend = "Hello !!"; //new line in content
            FileOutputStream outputStream = new FileOutputStream("testout.txt",
true); //For appending the data here we need to give true in second argument
            byte[] strToBytes = textToAppend.getBytes();
            outputStream.write(strToBytes);
            outputStream.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

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.

```
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.length bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to len 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 FileDescriptor 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 stream .

Java FileInputStream example 1: read single character

```
import java.io.FileInputStream;
public class Main{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("testout.txt");
            int i=fin.read();
            System.out.print((char)i);

            fin.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

Java FileInputStream example 2: read all characters

```
import java.io.FileInputStream;
public class Main{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("testout.txt");
            int i=0;
            while((i=fin.read())!=-1){
                System.out.print((char)i);
            }
            fin.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```


Practice Programs to do

WAP to copy the data of one file into another file

WAP to read all characters from a file and count total number of vowels from it

WAP to read the content from two files and write their merged content into third file

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.
- Java allows us to wrap a byte-oriented file stream within a character-based object.
- We can use **Scanner** and **PrintWriter** class to read and write Files.

PrintWriter Class

This class gives Prints formatted representations of objects to a text-output stream. It implements all of the print methods found in `PrintStream`.

`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`.

PrintWriter Methods

java.io.PrintWriter

```
+PrintWriter(file: File)
+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void
```

Also contains the overloaded
`println` methods.

Also contains the overloaded
`printf` methods.

Example 1

```
import java.io.*;
public class Main
{
    public static void main(String args[])
    {
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("Using PrintWriter Object");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Example 2

```
import java.io.*;
public class printwriter{
    public static void main(String args[]){
        try{
            PrintWriter pw=new PrintWriter(new File("target.txt"));
            pw.print("Hi");
            pw.flush();
            pw.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

Using Scanner

- The `java.util.Scanner` class is used to read strings and primitive values from the console.
- Scanner breaks the input into tokens delimited by whitespace characters.
- To read from the keyboard, we create a Scanner as follows:
`Scanner s = new Scanner(System.in);`
- To read from a file, create a Scanner for a file, as follows:
`Scanner s = new Scanner(new File(filename));`

Scanner Methods

java.util.Scanner

```
+Scanner(source: File)
+Scanner(source: String)
+close()
+hasNext(): boolean
+next(): String
+nextLine(): String
+nextByte(): byte
+nextShort(): short
+nextInt(): int
+nextLong(): long
+nextFloat(): float
+nextDouble(): double
+useDelimiter(pattern: String):
  Scanner
```


Serialization and Deserialization in Java

Serialization in Java is a mechanism of writing the state of an object into a byte-stream.

The reverse operation of serialization is called deserialization where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

For serializing the object, we call the **writeObject()** method of `ObjectOutputStream`, and for deserialization we call the **readObject()** method of `ObjectInputStream` class.

- We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

- It is mainly used to travel object's state on the network (which is known as marshaling).

java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

```
import java.io.Serializable;
public class Student implements Serializable
{
    int id;
    String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

```
public ObjectOutputStream(OutputStream out) throws  
IOException {}
```

creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) <code>public final void writeObject(Object obj) throws IOException {}</code>	writes the specified object to the <code>ObjectOutputStream</code> .
2) <code>public void flush() throws IOException {}</code>	flushes the current output stream.
3) <code>public void close() throws IOException {}</code>	closes the current output stream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

```
public ObjectInputStream(InputStream in) throws  
IOException {}
```

creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Important Methods

Method	Description
1) <code>public final Object readObject() throws IOException, ClassNotFoundException{}</code>	reads an object from the input stream.
2) <code>public void close() throws IOException {}</code>	closes <code>ObjectInputStream</code> .

Example ObjectOutputStream

```
import java.io.*;
class student implements Serializable
{
    int roll_no;
    String name;
    student (int r,String s)
    {
        roll_no=r;
        name=s;
    }
}
public class Main
{
    public static void main(String[] args) throws Exception
    {
        FileOutputStream fos = new FileOutputStream("t.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        student s= new student(1,"kk");
        oos.writeObject(s);

        oos.close();
    }
}
```

Example ObjectOutputStream

```
import java.io.*;
class student implements Serializable
{
    int roll_no;
    String name;
    student (int r,String s)
    {
        roll_no=r;
        name=s;
    }
}
public class Example
{
    public static void main(String[] args) throws Exception
    {
        FileInputStream fos = new FileInputStream("E:\\t.txt");
        ObjectInputStream oos = new ObjectInputStream(fos);
        student s=(student)oos.readObject();
        System.out.println(s.roll_no+" "+s.name);
        oos.close();
    }
}
```

Java Serialization with Inheritance

If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:

```
import java.io.Serializable;
class Person implements Serializable{
    int id;
    String name;
    Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

```
class Student extends Person{  
    String course;  
    int fee;  
    public Student(int id, String name, String course, int fee) {  
        super(id,name);  
        this.course=course;  
        this.fee=fee;  
    }  
}
```

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Practice questions

Create a class Book, with fields: Book_id, name and price. Initialize these attributes through constructor. Perform serialization and create 2 objects. Write that object into the file whose price is less, and then perform the deserialization also.

Create a class first with attributes: principle, rate, initialize their values parameterized constructor. Extend this class into another class: second, with attribute: time and simple interest, initialize the attributes through parameterized constructor. Calculate simple interest also and write the object into file through serialization and also perform deserialization

Which of these methods are used to read in from file?

a) `get()`

b) `read()`

c) `scan()`

d) `readFileInput()`

Which of these values is returned by read() method is end of file (EOF) is encountered?

- a) 0
- b) 1
- c) -1
- d) Null

Which of these exception is thrown by close() and read() methods?

- a) IOException
- b) FileNotFoundException
- c) FileNotFoundException
- d) FileInputOutputException

Which of these methods is used to write() into a file?

- a) put()
- b) putFile()
- c) write()
- d) writeFile()

What will be the output of the following Java program?

```
import java.io.*;
class filesinputoutput
{
    public static void main(String args[])
    {
        InputStream obj = new FileInputStream("inputoutput.java");
        System.out.print(obj.available());
    }
}
```

Note: inputoutput.java is stored in the disk.

- a) true
- b) false
- c) prints number of bytes in file
- d) prints number of characters in the file