UNIT III EXCEPTION HANDLING AND I/O

Exceptions – exception hierarchy – throwing and catching exceptions – built-in exceptions, creating own exceptions, Stack Trace Elements. Input / Output Basics – Streams – Byte streams and Character streams – Reading and Writing Console –

Question : 1.a.What is Exception?

Or

1.b.Define Exception

Answer

The exception handling in java is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained. Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Question : 2.a.What are the types of Exception?

Or

2.b.Write the classification of Exception?

Answer:

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

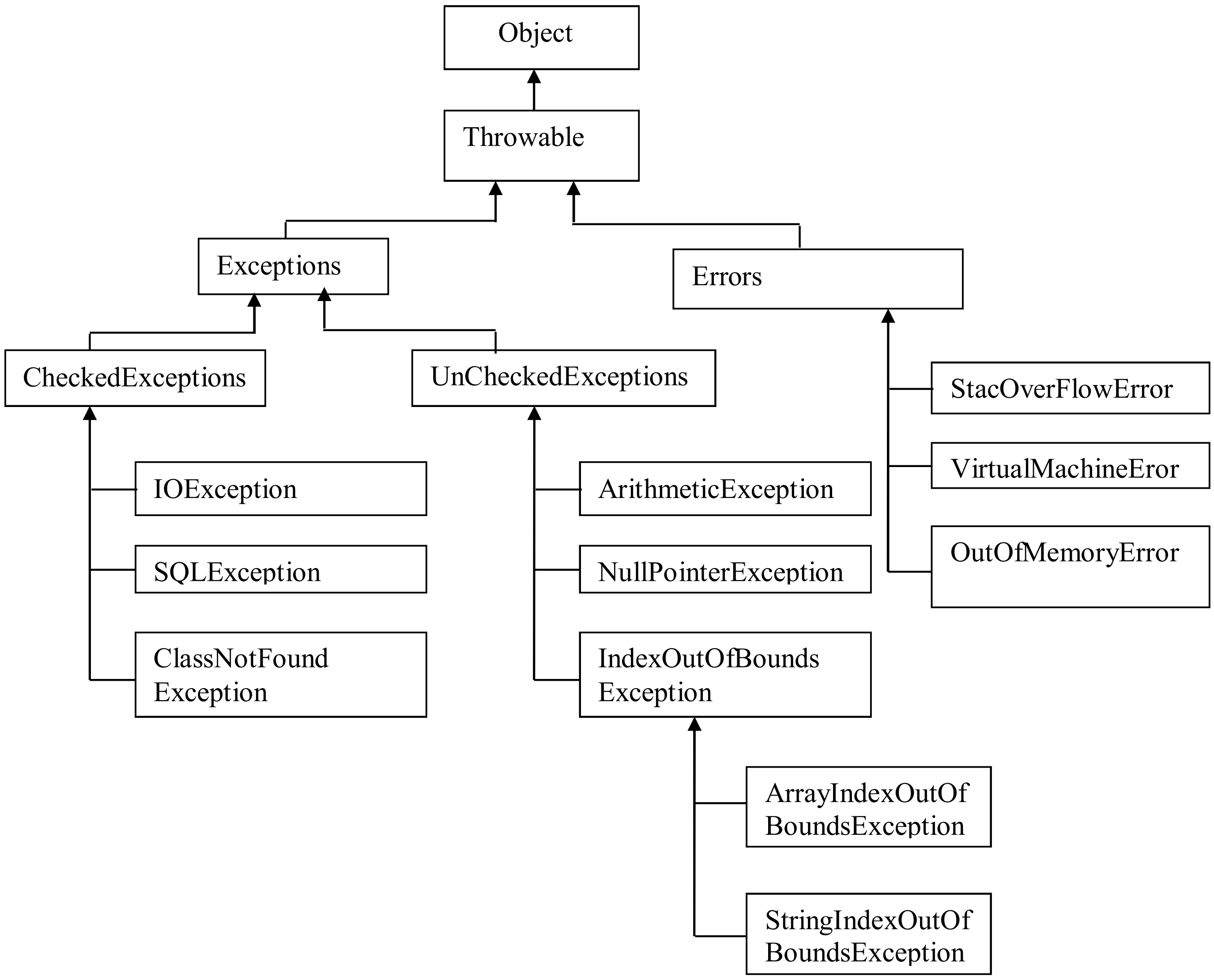
Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Question : 3.a.Draw the Hierarchy of Java Exception classes

Or

3.b. Write about Exception Hierarchy in Java

Answer



<u>Question:</u>
4.a.Explain the throwing and catching the exception in java
Or
4.b.Discuss on exception handling in java.
Or
4.c.Explain exception handling in java with examples.
Or
4.d.Explain 5 keywords in exception handling in java with examples.

Answer:

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

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

Java Try Block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.Java try block must be followed by either catch or finally block.

Syntax

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

Catching Exceptions

A method catches an exception using a combination of the try and catchkeywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

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 occurred is handled by catch block associated with it. Every trya block should be immediately followed either by a catch block or finally block.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest
{
            public static void main(String args[])
            {
                        try
                        {
                                    int a[] = new int[2];
                                    System.out.println("Access element three :" + a[3]);

                        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

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

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try
{
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}
 catch (IOException i)
{
  i.printStackTrace();
  return -1;
} catch (FileNotFoundException f) // Not valid! {
  f.printStackTrace();
  return -1;
}
```

The Throws/Throw Keywords

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

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

Try to understand the difference between throws and throw keywords, *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 RemoteException −

Example

```
import java.io.*;
public class className
{
    public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // 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 occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

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

Syntax

```
try {

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

Example

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

This will produce the following result −

Output

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

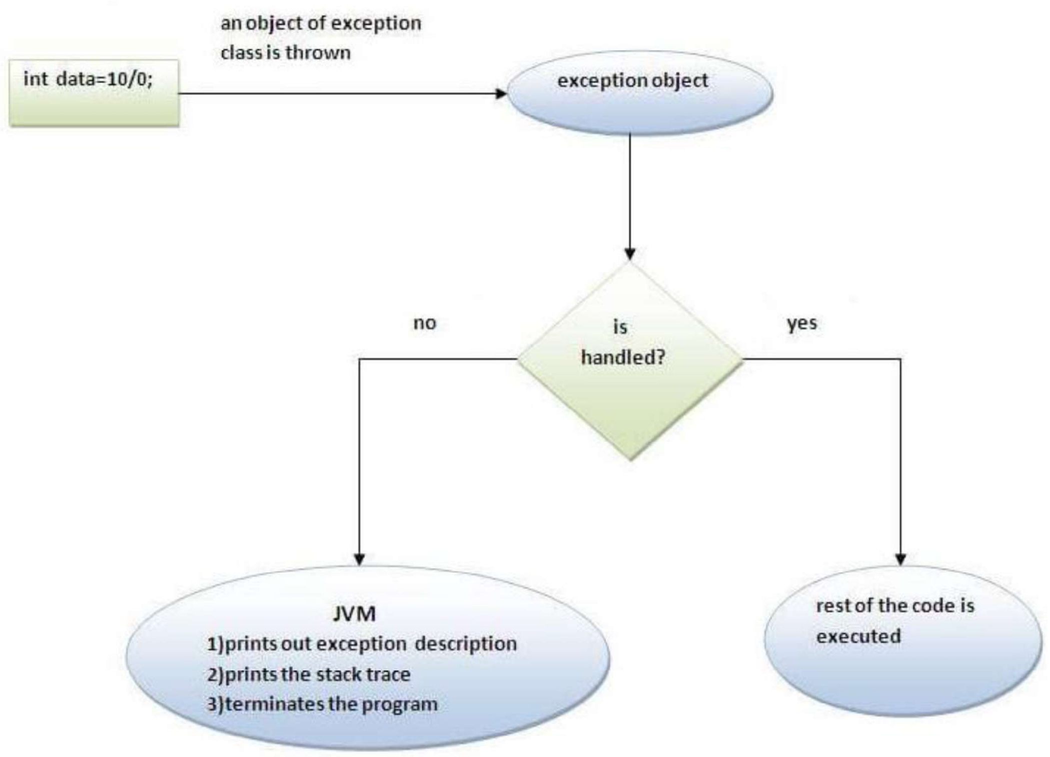
Question

5.a.Draw the Internal working of java try-catch block

Or

5.b. How the try-catch block works in Java?

Answer:


an object of exception class is thrown
int data=10/0;
exception object
no
is handled?
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Question :
6.a. Explain Built-In Exceptions in Java with example programs.
Or
6.b.What are the predefined exceptions in java with example programs?

Answer

SI.NO.	Java UnChecked Exceptions Defined in java.lang.	Java Checked Exceptions Defined in java.lang.
1.	ArithmeticException Arithmetic error, such as divide-by-zero.	ClassNotFoundException Class not found.
2.	ArrayIndexOutOfBoundsException Array index is out-of-bounds.	CloneNotSupportedException Attempt to clone an object that does not

		implement the Cloneable interface.
3.	ArrayStoreException Assignment to an array element of an incompatible type.	IllegalAccessException Access to a class is denied.
4.	ClassCastException Invalid cast.	InstantiationException Attempt to create an object of an abstract class or interface.
5.	IllegalArgumentException Illegal argument used to invoke a method.	InterruptedException One thread has been interrupted by another thread.
6.	IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.	NoSuchFieldException A requested field does not exist.
7.	IllegalStateException Environment or application is in incorrect state.	NoSuchMethodException A requested method does not exist.
8.	IllegalThreadStateException Requested operation not compatible with the current thread state.	
9.	IndexOutOfBoundsException Some type of index is out-of-bounds.	
10.	NegativeArraySizeException Array created with a negative size.	
11.	NullPointerException Invalid use of a null reference.	
12.	NumberFormatException Invalid conversion of a string to a numeric format.	
13.	SecurityException Attempt to violate security.	
14.	StringIndexOutOfBounds Attempt to index outside the bounds of a string.	

15.	UnsupportedOperationException An unsupported operation was countered.	

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.

Examples of Built-in Exception:

1. **Arithmetic exception :** It is thrown when an exceptional condition has occurred in an arithmetic operation.

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

Output:

```
Can't divide a number by 0
```

2. **ArrayIndexOutOfBounds 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.

```
// Java program to demonstrate
// ArrayIndexOutOfBoundException
```

```
class ArrayIndexOutOfBound_Demo
{
public static void main(String args[])
   {
      try {
         int a[] = new int[5];
         a[6] = 9; // accessing 7th element in an array of
         // size 5
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array Index is Out Of Bounds");
      }
   }
}
```

Output:

```
Array Index is Out Of Bounds
```

3. **ClassNotFoundException** : This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to illustrate the
// concept of ClassNotFoundException
class Bishal
{
}
 class Geeks
{
}
 class MyClass
{
public static void main(String[] args)
   {
      Object o = class.forName(args[0]).newInstance();
      System.out.println("Class created for" + o.getClass().getName());
   }
```

```
            }
```

Output:

```
        ClassNotFoundException
```

4. **FileNotFoundException** : This Exception is raised when a file is not accessible or does not open.

```
// Java program to demonstrate
// FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo
{
public static void main(String args[])
   {
      try {

         // Following file does not exist
         File file = new File("E:// file.txt");

         FileReader fr = new FileReader(file);
      }
      catch (FileNotFoundException e) {
         System.out.println("File does not exist");
      }
   }
}
```

Output:

```
    File does not exist
```

5. **IOException** : It is thrown when an input-output operation failed or interrupted

```
// Java program to illustrate IOException
import java.io.*;
class Geeks
 {
public static void main(String args[])
   {
      FileInputStream f = null;
      f = new FileInputStream("abc.txt");
      int i;
      while ((i = f.read()) != -1) {
         System.out.print((char)i);
      }
```

```
            f.close();
          }
        }
```

Output:

error: unreported exception IOException; must be caught or declared to be thrown

6. **InterruptedException** : It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

```
// Java Program to illustrate
// InterruptedException
class Geeks {
public static void main(String args[])
  {
     Thread t = new Thread();
     t.sleep(10000);
  }}
```

Output:

error: unreported exception InterruptedException; must be caught or declared to be thrown

7. **NoSuchMethodException :** t is thrown when accessing a method which is not found.

```
// Java Program to illustrate
// NoSuchMethodException
class Geeks {
public Geeks()
 {
     Class i;
     try {
       i = Class.forName("java.lang.String");
       try {
          Class[] p = new Class[5];
       }
       catch (SecurityException e) {
          e.printStackTrace();
       }
```

```
                catch (NoSuchMethodException e) {
                    e.printStackTrace();
                }
            }
            catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    public static void main(String[] args)
        {
            new Geeks();
        }
    }
```

Output:

```
error: exception NoSuchMethodException is never thrown
in body of corresponding try statement
```

8. **NullPointerException** : This exception is raised when referring to the members of a null object. Null represents nothing

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

Output:

```
NullPointerException..
```

9. **NumberFormatException** : This exception is raised when a method could not convert a string into a numeric format.

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

Output:

```
Number format exception
```

10. **StringIndexOutOfBoundsException** : It is thrown by String class methods to indicate that an index is either negative than the size of the string.

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

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

Question :
7.a.Explain Creating Own Exception in java with examples
Or
7.b.Expain about Custom Exception with example
Or
7.c.Write about User-Defined exception with examples

Answer

Creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Example of java custom exception.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s)
{
  super(s);
 }
}
class TestCustomException1
{
    static void validate(int age)throws InvalidAgeException
    {

        if(age<18)
         throw new InvalidAgeException("not valid");
        else
         System.out.println("welcome to vote");
      }
     public static void main(String args[])
    {
            try{
                validate(13);
                    }catch(Exception m){System.out.println("Exception occured: "+m);
        }
        System.out.println("rest of the code...");
      }
    }
```

Output: Exception occured: InvalidAgeException:not valid

```
    rest of the code...
```

Question :
8.a.What are the methods in Stack Trace Elements?Give Examples
Or
8.b.Explain about Stack trace elements in java with example programs.

Answer

The java.lang.StackTraceElement class element represents a single stack frame. All stack frames except for the one at the top of the stack represent a method invocation. The frame at the top of the stack represents the execution point at which the stack trace was generated.

Class Declaration

Following is the declaration for java.lang.StackTraceElement class −

public final class StackTraceElement extends Object implements Serializable

Class Constructors

S.No.	Constructor & Description
1	StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber) This creates a stack trace element representing the specified execution point.

Class methods

S.No.	Method & Description
1	boolean equals(Object obj) This method returns true if the specified object is another StackTraceElement instance representing the same execution point as this instance.
2	String getClassName()

	This method returns the fully qualified name of the class containing the execution point represented by this stack trace element.
3	String getFileName() This method returns the name of the source file containing the execution point represented by this stack trace element.
4	int getLineNumber() This method returns the line number of the source line containing the execution point represented by this stack trace element.
5	String getMethodName() This method returns the name of the method containing the execution point represented by this stack trace element.
6	int hashCode() This method returns a hash code value for this stack trace element.
7	boolean isNativeMethod() This method returns true if the method containing the execution point represented by this stack trace element is a native method.
8	String toString() This method returns a string representation of this stack trace element

boolean equals(ob): Returns try if the invoking StackTraceElement is as the one passed in ob. Otherwise it returns false.

Syntax: public boolean equals(ob)

Returns: true if the specified object is

another StackTraceElement instance representing the same execution

point as this instance.

Exception: NA

```
// Java code illustrating equals() method
import java.lang.*;

import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
      StackTraceElement st1 = new StackTraceElement("foo", "fuction1",
                                "StackTrace.java", 1);
      StackTraceElement st2 = new StackTraceElement("bar", "function2",
                                "StackTrace.java", 1);
     Object ob = st1.getFileName();
     // checking whether file names are same or not
      System.out.println(st2.getFileName().equals(ob));
   }
}
```

Output:

```
true
```

1. **String getClassName():** Returns the class name of the execution point described by the invokingStackTraceElement.

Syntax: public String getClassName().
Returns: the fully qualified name of the Class
containing the execution point represented by this stack trace element.
Exception: NA.

```
// Java code illustrating getClassName() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
   public static void main(String[] arg)
   {
      System.out.println("Class name of each thread involved:");
      for(int i = 0; i<2; i++)
      {
         System.out.println(Thread.currentThread().getStackTrace()[I].
                getClassName());
            }
        }
    }
```

Output:

```
Class name of each thread involved:
java.lang.Thread
StackTraceElementDemo
```

2. **String getFileName():** Returns the file name of the execution point described by the invokingStackTraceElement.

Syntax: public String getFileName().

Returns: the name of the file containing

the execution point represented by this stack trace element,

or null if this information is unavailable.

Exception: NA.

```
// Java code illustrating getFileName() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("file name: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      getFileName());
  }
}
```

Output:

```
file name: Thread.java
        StackTraceElementDemo.java
```

3. int getLineNumber(): Returns the source-code line number of the execution point described by the invoking StackTraceElement. In some situation the line number will not be available, in which case a negative value is returned.

Syntax: public int getLineNumber().

Returns: the line number of the source line containing the execution point represented by this stacktrace element, or a negative number if this information is unavailable.

Exception: NA.

```
// Java code illustrating getLineNumber() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("line number: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      getLineNumber());
  }
}
```

Output:

```
line number:
1556
10
```

4. String getMethodName(): Returns the method name of the execution point described by the invoking StackTraceElement.

Syntax: public String getMethodName().

```
// Java code illu
  // Java code illustrating getFileName() method.
  import java.lang.*;
  import java.io.*;
  import java.util.*;
```

```
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("file name: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      getFileName());
  }
}
```

Output:

```
file name: Thread.java
         StackTraceElementDemo.java
```

5. int getLineNumber(): Returns the source-code line number of the execution point described by the invoking StackTraceElement. In some situation the line number will not be available, in which case a negative value is returned.

Syntax: public int getLineNumber().

Returns: the line number of the source line containing the execution point represented by this stack trace element, or a negative number if this information is unavailable.

Exception: NA.

```
// Java code illustrating getLineNumber() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("line number: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      getLineNumber());
      }
    }
```

Output:

```
line number:
1556
10
```

6. String getMethodName(): Returns the method name of the execution point described by the invoking StackTraceElement.

Syntax: public String getMethodName().

// Java code illustrating getMethodName() method.

```
import java.lang.*;
import java.io.*;
import java.util.*;

public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("method name: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      getMethodName());
  }}
```

Output:

```
    method name:
    getStackTrace
    main
```

int hashCode(): Returns the hash code of the invoking StackTraceElement.

Syntax: public int hashCode().

Returns: a hash code value for this object.

Exception: NA.

```
// Java code illustrating hashCode() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
        public static void main(String[] arg)
        {
              System.out.println("hash code: ");
              for(int i = 0; i<2; i++)
              System.out.println(Thread.currentThread().getStackTrace()[i].
              hashCode());
        }
}
```

Output:

```
hash code:
-1225537245
-1314176653
```

7. **boolean isNativeMethod():** Returns true if the invoking StackTraceElement describes a native method. Otherwise returns false.

Syntax: public boolean isNativeMethod().

Returns: true if the method containing the execution point represented by this stack trace element is a native method.

```
// Java code illustrating isNativeMethod() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void ma`in(String[] arg)
  {

     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      isNativeMethod());
  }
}
```

Exception: NA.

Output:

```
false
false
```

8. String toString(): Returns the String equivalent of the invoking sequence.

Syntax: public String toString().

Returns: a string representation of the object.

Exception: NA.

```
// Java code illustrating toString() method.
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo
{
  public static void main(String[] arg)
  {
     System.out.println("String equivlaent: ");
     for(int i = 0; i<2; i++)
     System.out.println(Thread.currentThread().getStackTrace()[i].
      toString());
  }
}
```

Output:

```
String equivlaent:
java.lang.Thread.getStackTrace
StackTraceElementDemo.main
```

Question :

9.a. What is I/O?Explain in detail.

Or

9.b.Define I/P .

Answer

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. 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.

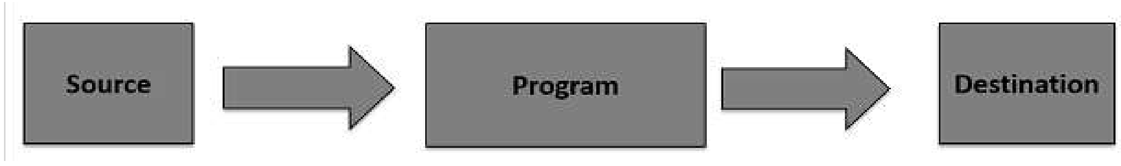
Question :
10.a.What is Stream? Explain in detail.
Or
10.b.Define Stream and its types.

Answer

- InputStream − The InputStream is used to read data from a source.

- OutputStream − The OutputStream is used for writing data to a destination.

In java, 3 streams are created for us automatically. All these streams are attached with 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 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
```

Question :
11.a. Explain the methods and hierarchy of Output stream class.
Or
11.b.Write about Outputstream class in detail.

Answer

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.

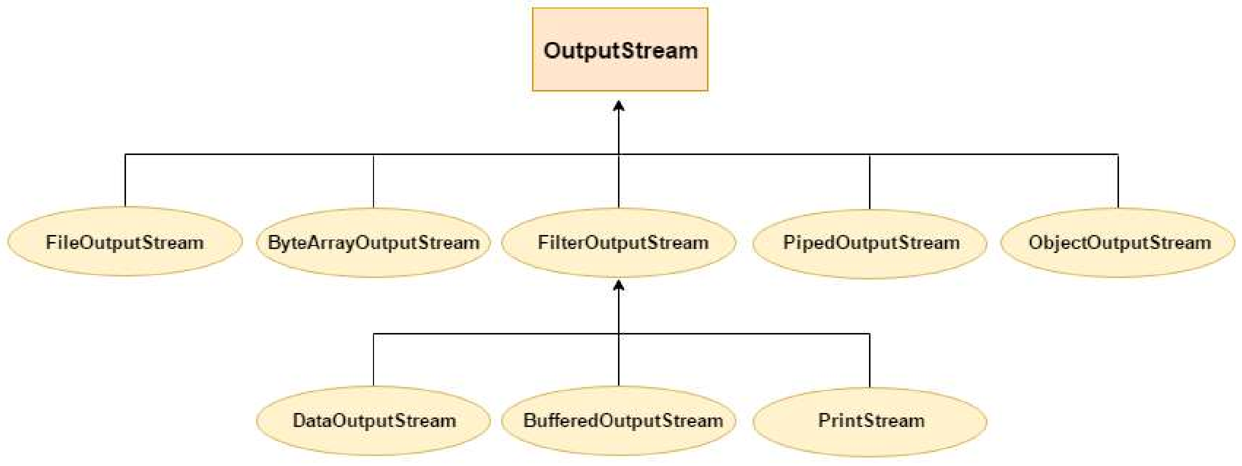
OutputStream class

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.

Useful Methods of OutputStream

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

OutputStream Hierarchy



Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file. If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream Class Declaration

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

public class FileOutputStream extends OutputStream

FileOutputStream Class Methods

Method	Description
protected void finalize()	It is sued 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 FileOutputStreamExample
{
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);}
}
}
```

Output:

```
        Success...
```

The content of a text file testout.txt is set with the data A.

testout.txt

A

Java FileOutputStream example 2: write string

```
import java.io.FileOutputStream;
public class FileOutputStreamExample
{
public static void main(String args[])
{
        try{
          FileOutputStream fout=new FileOutputStream("D: \\testout.txt");
          String s="Welcome to javaTpoint.";
          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);}
        }
}
```

Output:

```
Success...
```

The content of a text file testout.txt is set with the data Welcome to java

testout.txt

Welcome to Java

12.a. Explain the methods and hierarchy of Input stream class.
Or
12.b.Write about Inputstream class in detail.
Or
12.c.What is the uses of input stream class? Explain the methods defined by input stream class.

Answer

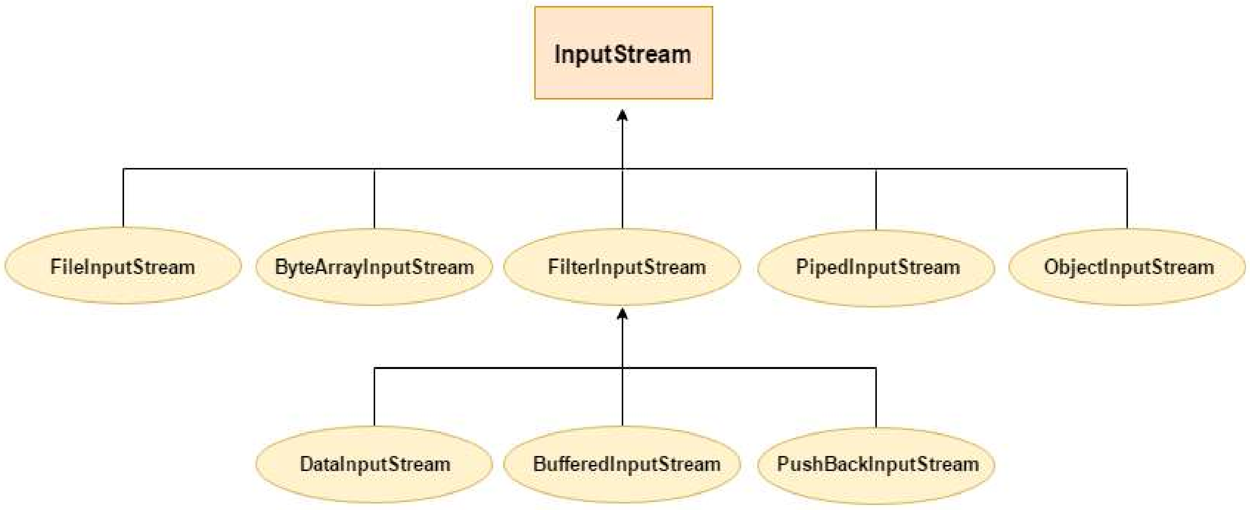
InputStream class

InputStream class is an abstract class. It is the super class 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 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 FileInputStream Class

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

Java FileInputStream Class Declaration

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

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 DataStreamExample
{
public static void main(String args[])
{
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=fin.read();
          System.out.print((char)i);
```

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

Before running the code, a text file named as "testout.txt" is required to be created. In this file, we are having following content:

Welcome to java

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output: W

Java FileInputStream example 2: read all characters

```
package com.javatpoint;
import java.io.FileInputStream;
public class DataStreamExample
{
public static void main(String args[])
          {
               try{
                 FileInputStream fin=new FileInputStream("D:\\testout.txt");
                 int i=0;
                 while((i=fin.read())!=-1)
          {

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

Output: Welcome to java

13.a.Write about Bytestream class in detail.
Or
13.b.What is the uses of Bytestream?Explain the methods Java ByteArrayOutputStream Class.

Answer

Java ByteArrayOutputStream class is used to write common data into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later. The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams. The buffer of ByteArrayOutputStream automatically grows according to data.

Java ByteArrayOutputStream Class Declaration

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

public class ByteArrayOutputStream extends OutputStream

Java ByteArrayOutputStream Class Constructors

Constructor	Description
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Java ByteArrayOutputStream class methods

Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len	It is used for writing len bytes from specified byte array starting from the offset off to the byte array output stream.

void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.

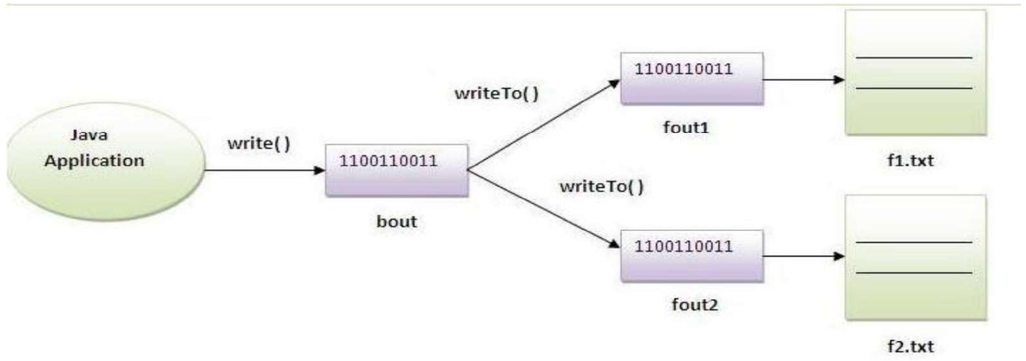
Example of Java ByteArrayOutputStream

Let's see a simple example of java ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

```
package com.javatpoint;
import java.io.*;
public class DataStreamExample
{
public static void main(String args[])throws Exception
{
      FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
     FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");
     ByteArrayOutputStream bout=new ByteArrayOutputStream();
     bout.write(65);
     bout.writeTo(fout1);
     bout.writeTo(fout2);
     bout.flush();
     bout.close();//has no effect
     System.out.println("Success...");
    }
    s}
```

Output:

```
Success...
f1.txt:
A
f2.txt:
A
```



14.a.Write about Character stream class in detail.
Or
14.b.What is the uses of Character stream?Explain the methods.

Answer

Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile
 {
  public static void main(String args[]) throws IOException
  {
          FileReader in = null;
          FileWriter out = null;
          try {
               in = new FileReader("input.txt");
               out = new FileWriter("output.txt");
               int c;
               while ((c = in.read()) != -1) {

              out.write(c);
              }
          }finally {
       if (in != null)
       {
         in.close();
       }
       if (out != null)
      {
         out.close();
       }
      }
    }
}
```

Now let's have a file input.txt with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

15.a.What are the Reading and writing methods in Java Console class.

Or

15.b.Explain how to read and write datas using Java Console class methods.

Answer

The Java Console class is be used to get input from console. It provides methods to read texts and passwords. If you read password using Console class, it will not be displayed to the user. The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console Class Declaration

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

```
public final class Console extends Object implements Flushable
```

Java Console Class Methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

```
public static Console console(){}
```

Let's see the code to get the instance of Console class.

```
Console c=System.console();
```

Java Console Example

```
import java.io.Console;
class ReadStringTest
```

```
{
    public static void main(String args[])
    {
        Console c=System.console();
        System.out.println("Enter your name: ");
        String n=c.readLine();
        System.out.println("Welcome "+n);
    }
}
```

Output

```
Enter your name: Nakul Jain
Welcome Nakul Jain
```

Java Console Example to read password

```
import java.io.Console;
class ReadPasswordTest
{
        public static void main(String args[])
        {
        Console c=System.console();
        System.out.println("Enter password: ");
        char[] ch=c.readPassword();
        String pass=String.valueOf(ch);//converting char array into string
        System.out.println("Password is: "+pass);
    }
}
```

Output

```
Enter password:
Password is: 123
```

16.a.What are the Reading and writing methods in Java Reader and Java Writer class.

Or

16.b.Explain how to read and write datas using Java Reader and Writer class methods.

Answer

Java Reader is an abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods to provide higher efficiency, additional functionality, or both.

Some of the implementation class are BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader.

Fields

Modifier and Type	Field	Description
protected Object	lock	The object used to synchronize operations on this stream.

Constructor

Modifier	Constructor	Description
Protected	Reader()	It creates a new character-stream reader whose critical sections will synchronize on the reader itself.
Protected	Reader(Object lock)	It creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods

Modifier and Type	Method	Description
abstract void	close()	It closes the stream and releases any system resources associated with it.
Void	mark(int readAheadLimit)	It marks the present position in the stream.
Boolean	markSupported()	It tells whether this stream supports the mark() operation.

Int	read()	It reads a single character.
Int	read(char[] cbuf)	It reads characters into an array.
abstract int	read(char[] cbuf, int off, int len)	It reads characters into a portion of an array.
Int	read(CharBuffer target)	It attempts to read characters into the specified character buffer.
Boolean	ready()	It tells whether this stream is ready to be read.
Void	reset()	It resets the stream.
Long	skip(long n)	It skips characters.

Example

```
import java.io.*;
public class ReaderExample
{
    public static void main(String[] args)
    {
         try
        {
                Reader reader = new FileReader("file.txt");
                int data = reader.read();
                while (data != -1)
                 {
                       System.out.print((char) data);
                       data = reader.read();
                }
            reader.close();
         } catch (Exception ex) {
            System.out.println(ex.getMessage());
            }
        }
    }
```

file.txt:

I love my country

Output:

I love my country

JAVA FILEWRITER CLASS

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

Java FileWriter class declaration

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

```
public class FileWriter extends OutputStreamWriter
```

Constructors of FileWriter Class

Constructor	**Description**
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Methods of FileWriter Class

Method	**Description**
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

Java FileWriter Example

In this example, we are writing the data in the file testout.txt using Java FileWriter class.

```
package com.javatpoint;
import java.io.FileWriter;
public class FileWriterExample
{
 public static void main(String args[])
{
        try
{
                                        FileWriter fw=new FileWriter("D:\\testout.txt");
            fw.write("Welcome to javaTpoint.");
            fw.close();
           }catch(Exception e){System.out.println(e);}
           System.out.println("Success...");


      }
}
```

Output:

```
Success...
```

testout.txt:

```
Welcome to java.
```

Part A – Question Bank

1.a.What is exception?

Or

1.b.Define Exception?

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

2.a.What is error?

Or

2.b.Define error?

An Error indicates that a non-recoverable condition has occurred that should not be caught. Error, a subclass of Throwable, is intended for drastic problems, such as OutOf-MemoryError, which would be reported by the JVM itself.

3.a.Which is super class of Exception?

Or

3.b.What is super class of Exception?