CONTENTS

- Exception
- Exception Handling Techniques
- User-Defined Exception
- File Class
- Reading and Writing Data
- Randomly Accessing a File
- The java.io package
- Serialization & De-Serialization

Exceptions

An Exception is an object that describes an exception or error condition that has occurred in code.

OR

Exception is a run time error.

An exception (or exceptional event) is a problem that arises during the execution of a program.

When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

Checked exceptions – A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

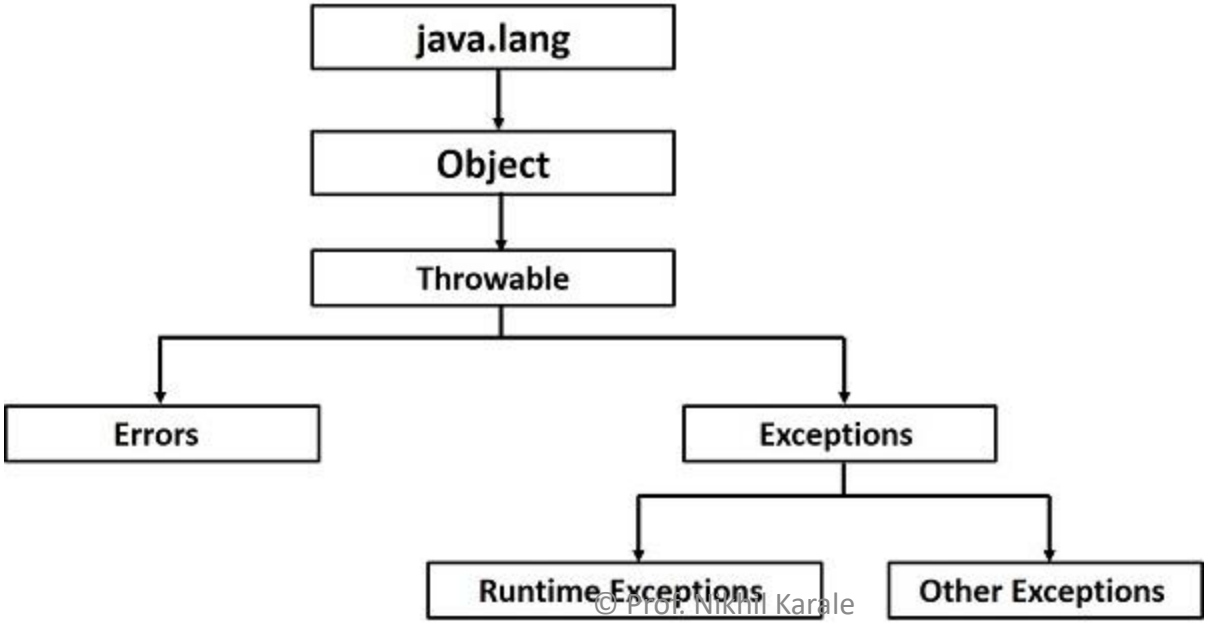
Unchecked exceptions– An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Errors – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

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



Exception Handling in Java

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.

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

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

try

catch

finally

throw

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 of java try-catch

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

Java catch block

❑ Java catch block is used to handle the Exception. It must be used after the try block only.

❑ You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
  public static void main(String args[]){
      int data=50/0;//may throw exception
      System.out.println("rest of the code...");
}
}
```

Output:

```
Exception in thread main
java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).
There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

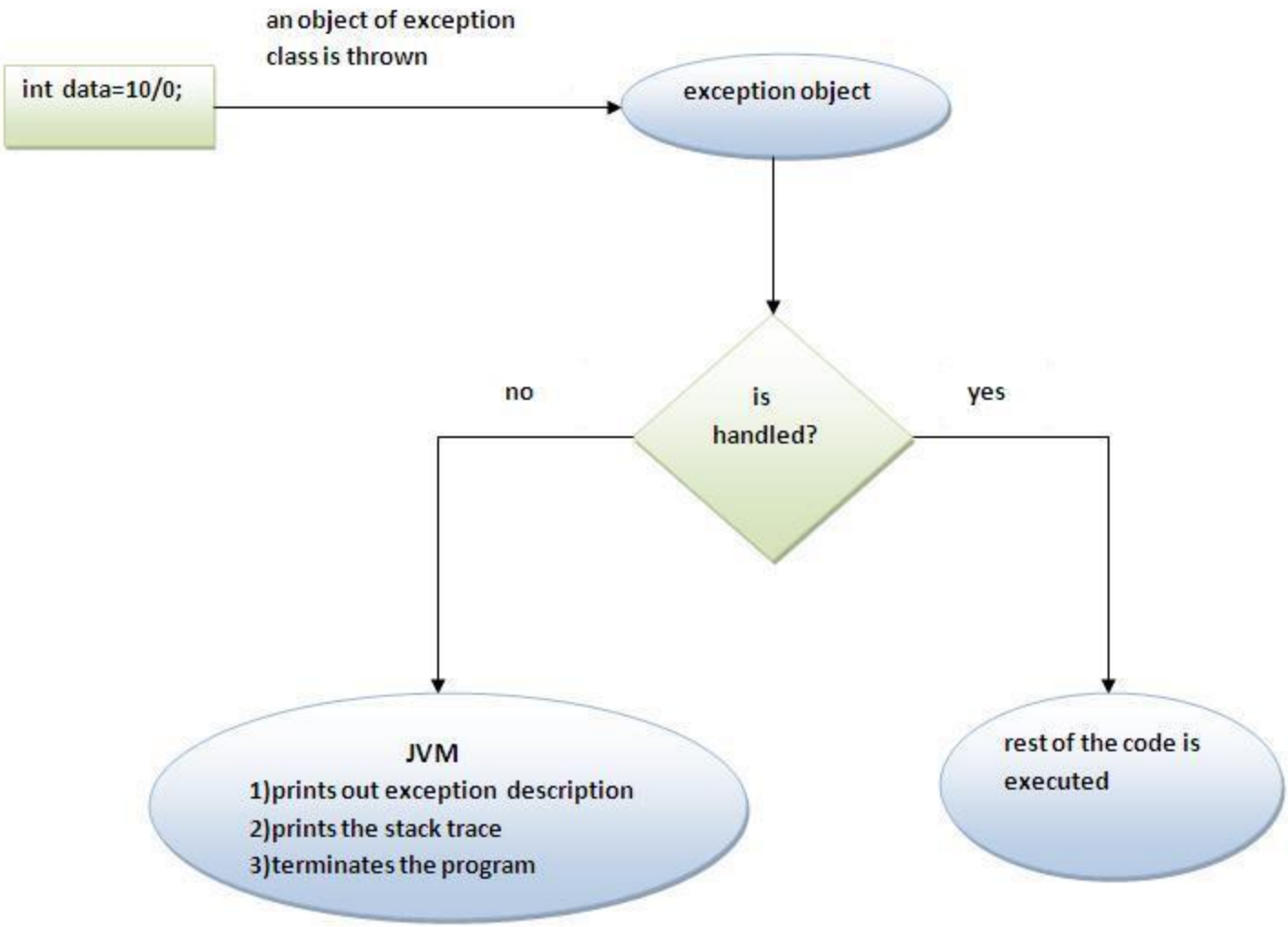
Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
      int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
}
}
```

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

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



Java catch multiple exceptions

Output:
task1 completed
rest of the code...

Java Multi catch block

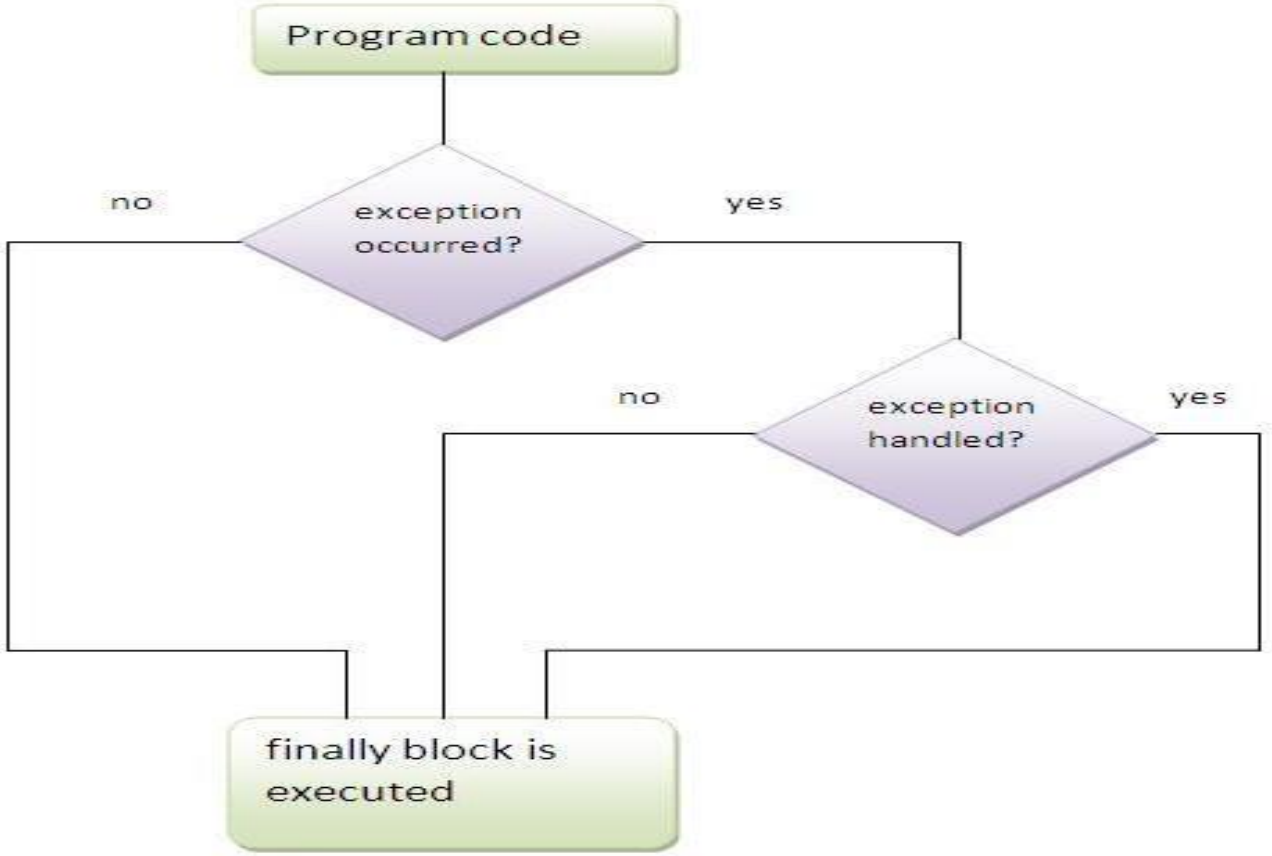
If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(ArithmeticException e){System.out.println("task1 is com
pleted");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println(
"task 2 completed");}
   catch(Exception e){System.out.println("common task complete
d");}

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

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.
Java finally block is always executed whether exception is handled or not.
Java finally block follows try or catch block.



Why use java finally

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1
Let's see the java finally example where **exception doesn't occur**.

Case 2
Let's see the java finally example where **exception occurs and not handled**.

Case 3
Let's see the java finally example where **exception occurs and handled**.

Case 3 Example:
Let's see the java finally example where **exception occurs and handled**.

```
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...
```

Java throw keyword

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
throw exception;
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

Exception in thread main java.lang.ArithmeticException: not valid

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code  }
```

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception    }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}    }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

```
exception handled
normal flow...
```

User-Defined Exceptions

If you are 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.
Let's see a simple example of java custom exception.

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

Continue Program on a next page

```
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...

Java File Class

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

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

Useful Methods

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.

- WAP to find the properties of file like path of file,, file exist or not, whether file or directory, length of file, parent of file, file last modified time. (14 Marks)

OR

- WAP to create file object and methods of file class to obtain its properties.

```
Import java.io.*;
public class Test {
public static void main(String args[]) {
File file=new File("sample.txt");
System.out.println("Does it Exist?" +file.exists());
System.out.println("Can it be read?"+file.canRead());
System.out.println("Can it be written?" +file.canWrite());
System.out.println("Is it a directory?" +file.isDirectory());
System.out.println("Is it  File?" +file.isFile());
System.out.println("Is it absolute?" +file.isAbsolute());
System.out.println("Absolute Path is:" +file.getAbsolutePath());
}
}
```

```
Does it Exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a File? true
Is it a absolute? false
Absolute Path is: c://sample.dat
```

- WAP to read a data (byte stream from) to a file using FileInputStream.
- WAP to write a data to a file using FileOutputStream.

```
//A program to save data to a file "sample.txt" using FileOutputStream
import java.io.*;
Class Test {
        public static void main(String args[]) throws IOException {
        File obj=new File("sample.txt");          //create file object
        FileOutputStream outStream=
                new FileOutputStream(obj);//create stream object
        String data = "This is my text";
        outStream.write(data); //write data to a file through stream
        System.out.println("Successfully written");
        outStream.close();
} }
```

```
//A program to read data from a file “sample.txt” using FileInputStream

Import java.io.*;
Class Test {
        public static void main(String args[]) throws IO Exception {
        File obj=new File(“sample.txt”);          //create file object
        FileInputStream inStream=
                        new FileInputStream(obj);//create stream object
        int fileSize=(int)obj.length();   //calcalute length of file.
        byte data[]=new byte[fileSize];           //create array of file size.
        inStream.read(data);              //read data
        for(int i=0;i<fileSize;i++)       {
        System.out.println(“\t” +data[i]);
        }
        inStream.close();
}
}
```

```
5       10      15      20      25      30      35      40
```

Java FileReader Class

- Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.
- It is character-oriented class which is used for file handling in java.

Constructors of FileReader class

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

WAP to read character stream to a file using fileReader

```
import java.io.*;
public class FileReaderDemo {
    public static void main(String args[])throws Exception{
FileReader fr=new FileReader("testout.txt");
BufferedReader br=new BufferedReader(fr);
        String s;
        while((s=br.readLine())!=null){
        System.out.println("\n" +s);
}
        fr.close();
    }
}
```

Welcome to File Reader Class

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.

WAP to write character stream to a file using fileWriter

```
import java.io.FileWriter;
public class FileWriterExample {
    public static void main(String args[]){
        try{
          FileWriter fw=new FileWriter("testout.txt");
          fw.write("Welcome to File Writer Class.");
          fw.close();
         }catch(Exception e){System.out.println(e);}
         System.out.println("Success...");
     }
}
```

```
Success...
```

Java BufferedInputStream Class

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about **BufferedInputStream** are:

- ❑ When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- ❑ When a BufferedInputStream is created, an internal buffer array is created.

Java BufferedInputStream class constructors

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

```
import java.io.*;
public class FileReaderDemo {
    public static void main(String args[])throws
Exception{
FileReader fr=new
FileReader("D:\\testout.txt");
BufferedReader br=new BufferedReader(fr);
        String s;
        while((s=br.readLine())!=null){
        System.out.print("\n" +s);
}
        fr.close();
    }
}
```

Welcome to File Reader Class

Java BufferedOutputStream Class

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Java BufferedOutputStream class constructors

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

    BufferedOutputStream bout=new BufferedOutputStream(fout);
    String s="Welcome to javaTpoint.";
    byte b[]=s.getBytes();
    bout.write(b);
    bout.flush();
    bout.close();
    fout.close();
    System.out.println("success");
}
}
```

```
Success
```

Randomly Accessing a File

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.
Class declaration

Following is the declaration for **Java.io.RandomAccessFile** class –

```
public class RandomAccessFile extends Object implements DataOutput, DataInput, Closeable
```

Class constructors

S.N.	Constructor & Description
1	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Class methods

S.N.	Method & Description
1	void close()-This method Closes this random access file stream and releases any system resources associated with the stream.
2	FileChannel getChannel()-This method returns the unique FileChannel object associated with this file.
3	FileDescriptor getFD()-This method returns the opaque file descriptor object associated with this stream.
4	long getFilePointer()-This method returns the current offset in this file.
5	long length()This method returns the length of this file.
6	int read()This method reads a byte of data from this file.
7	int read(byte[] b)-This method reads up to b.length bytes of data from this file into an array of bytes.
8	int read(byte[] b, int off, int len)-This method reads up to len bytes of data from this file into an array of bytes.
9	boolean readBoolean()-This method reads a boolean from this file.
10	byte readByte()-This method reads a signed eight-bit value from this file.
11	char readChar()-This method reads a character from this file.
12	double readDouble()-This method reads a double from this file.
13	float readFloat()-This method reads a float from this file.
14	void readFully(byte[] b)-This method reads b.length bytes from this file into the byte array, starting at the current file pointer.

15	**void readFully(byte[] b, int off, int len)**This method reads exactly len bytes from this file into the byte array, starting at the current file pointer.
16	**int readInt()**This method reads a signed 32-bit integer from this file.
17	**String readLine()**This method reads the next line of text from this file.
18	**long readLong()**This method reads a signed 64-bit integer from this file.
19	**short readShort()**This method reads a signed 16-bit number from this file.
20	**int readUnsignedByte()**This method reads an unsigned eight-bit number from this file.
21	**int readUnsignedShort()**This method reads an unsigned 16-bit number from this file.
22	**String readUTF()**This method reads in a string from this file.
23	**void seek(long pos)**This method sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
24	**void setLength(long newLength)**This method sets the length of this file.
25	**int skipBytes(int n)**This method attempts to skip over n bytes of input discarding the skipped bytes.
26	**void write(byte[] b)**This method writes b.length bytes from the specified byte array to this file, starting at the current file pointer.
27	**void write(byte[] b, int off, int len)**This method writes len bytes from the specified byte array starting at offset off to this file.

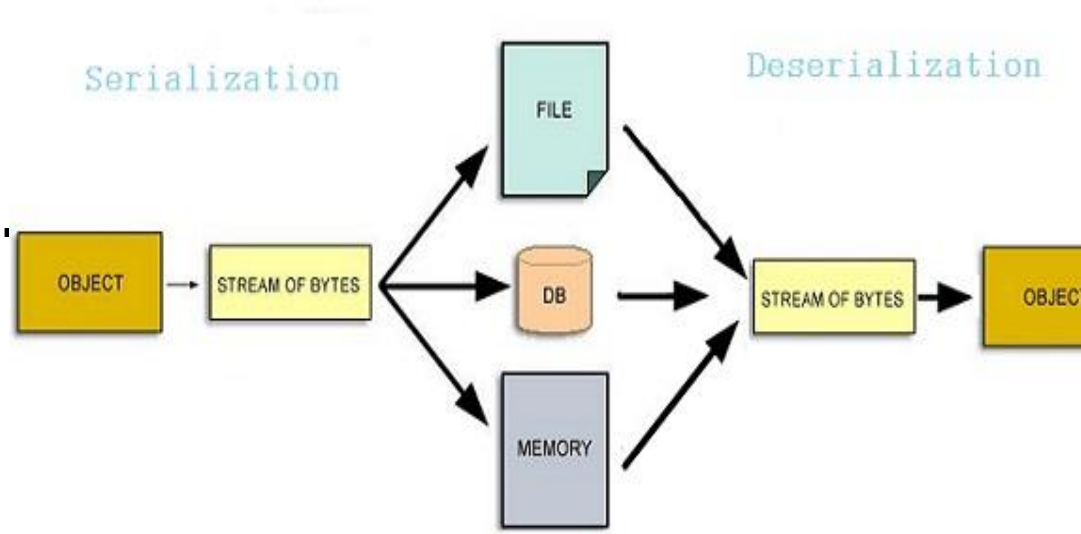
28	**void write(int b)**This method writes the specified byte to this file.
29	**void writeBoolean(boolean v)**This method writes a boolean to the file as a one-byte value.
30	**void writeByte(int v)**This method writes a byte to the file as a one-byte value.
31	**void writeBytes(String s)**This method writes the string to the file as a sequence of bytes.
32	**void writeChar(int v)**This method writes a char to the file as a two-byte value, high byte first.
33	**void writeChars(String s)**This method writes a string to the file as a sequence of characters.
34	**void writeDouble(double v)**This method converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
35	**void writeFloat(float v)**This method converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
36	**void writeInt(int v)**This method writes an int to the file as four bytes, high byte first.
37	**void writeLong(long v)**This method writes a long to the file as eight bytes, high byte first.
38	**void writeShort(int v)**This method writes a short to the file as two bytes, high byte first.
39	**void writeUTF(String str)**This method writes a string to the file using modified UTF-8 encoding in a machine-independent manner.

Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

A class must implement **Serializable** interface present in java.io package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behaviour to the class implementing it.

Java provides **Serializable** API encapsulated under java.io package for serializing and deserializing objects which include,

java.io.serializable
java.io.Externalizable
ObjectInputStream
and ObjectOutputStream etc.



java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get 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 implements *java.io.Serializable* interface by default.

Let's see the example given below:

```
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.

Example of Java Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

```
import java.io.*;
class Persist{
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi");

  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);

  out.writeObject(s1);
  out.flush();
  System.out.println("success");
 }
}
```

```
success
```

Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.
ObjectInputStream class
An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

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

FileInputStream fin = new FileInputStream("f.txt")
  ObjectInputStream in=new ObjectInputStream(fin);
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name);

  in.close();
 }
}
```

```
211 ravi
```