1.Why do we need to handle the exception? And how do we handle the exception? Explain with example

An exception normally disrupts the normal flow of the application that is why we use exception handling. Exceptions can be handled by using try,catch blocks try block throws the exception and the catch block catches the exception. EX:

```
 package files_exceptionhandling;
public class ExceptionEx {
            public static void main(String args[]){
                        try{
                                    int a[]= new int[5];
                                     a[7]=10;
                          // this statement throws an exception and it should be catch in catch block
                                     System.out.println("catch");
                        }
                        catch(ArrayIndexOutOfBoundsException e){ //it catches the above exception
                                    System.out.print("array index is out of bound");
                        }
            }
}
```

Output is : array index is out of bound

In the above example program the array size is 5 and we are trying to insert an element at the index position of 7 this gives an exception. try block throws this exception and catch block catches the exception

2.What is an Exception? Explain about kinds of Exceptions

Exception is an abnormal condition. An exception 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.

There are mainly 2 kinds of exceptions they are

I.checked exceptions

II.unchecked exceptions

I.checked exceptions − A checked exception is an exception that is checked or notified by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

II.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

3. Differentiate between an Error and Exception

Error : An Error “indicates serious problems that a reasonable application should not try to catch.”

Both Errors and Exceptions are the subclasses of java.lang.Throwable class.

Errors are the conditions which cannot get recovered by any handling techniques. It surely cause termination of the program abnormally.

Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error.

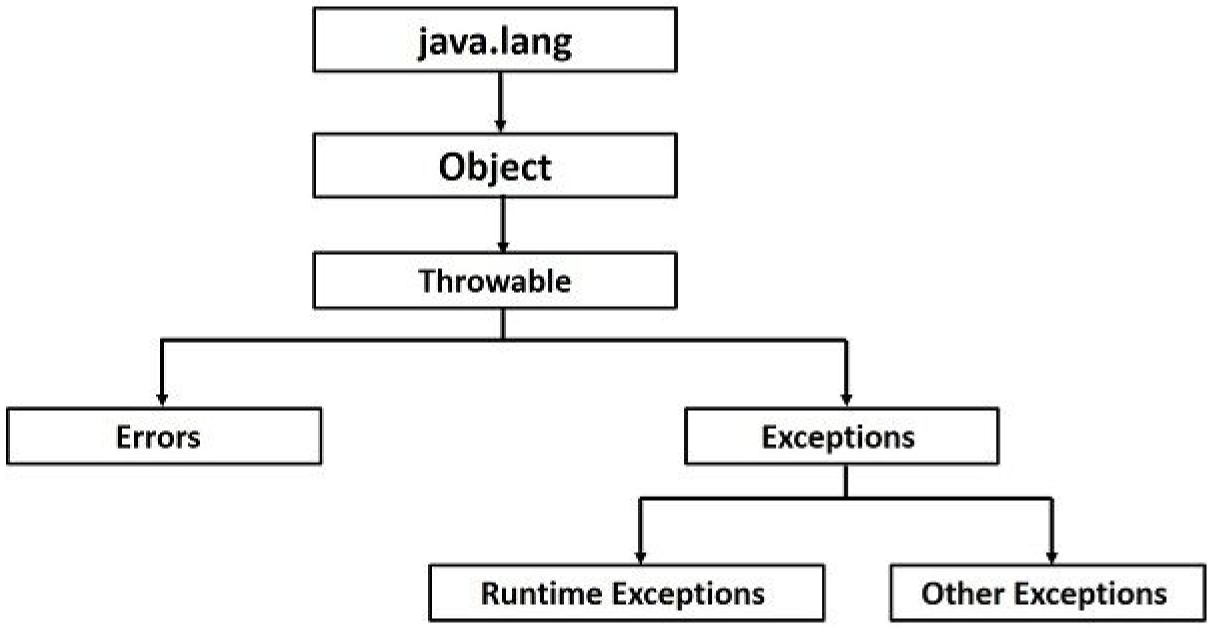
Exceptions : An Exception “indicates conditions that a reasonable application might want to catch.” Exceptions are the conditions that occur at runtime and may cause the termination of program. But they are recoverable using try, catch and throw keywords.
Exceptions are divided into two categories :checked exceptions and unchecked exceptions.
Checked exceptions like IOException known to the compiler at compile time while unchecked exceptions like ArrayIndexOutOfBoundException known to the compiler at runtime. It is mostly caused by the program written by the programmer.

4. Explain about 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.

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

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



5. Differentiate between throw and throws with examples

}

	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions eg: public void method() throws IOException, SQLException

Example program for throw:

```
package files_exceptionhandling;

import java.util.Scanner;

public class ThrowEx {

        void validate(int age){

                if(age<18){

                        throw new ArithmeticException("age is not sufficient for vote");
 //this is throw an exception

                }
```

```
		else{System.out.print("you can vote"); }
	}
	public static void main(String args[]){
		ThrowEx t = new ThrowEx();
		System.out.print("enter age :");
		Scanner sc = new Scanner(System.in);
		int age = sc.nextInt();
		t.validate(age);
		sc.close();
	}
}
```

In the above example we explicitly throw the exception using the throw keyword

Example for throws:

```
public class ThrowsEx {
	void method1() throws ArithmeticException{
// it throws an Arithmetic exception and it is handled in catch block
		throw new ArithmeticException();
	}
		void method3(){
		try{
			method1(); //control goes to method1
		}
		catch(ArithmeticException e){
			System.out.print("Arithmetic exception");
```

```
        }
    }
        public static void main(String args[]){
            ThrowsEx t = new ThrowsEx();
            t.method3(); // calling method3
        }
    }
```

6. What are the different types of Exceptions in java? Explain in detail

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.

1. **Arithmetic Exception**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundException**
   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.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.

9. **NullPointerException**

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

10. **NumberFormatException**

    This exception is raised when a method could not convert a string into a numeric format.

11. **RuntimeException**

    This represents any exception which occurs during runtime.

12. **StringIndexOutOfBoundsException**

    It is thrown by String class methods to indicate that an index is either negative than the size of the string

22. Explain about Autoboxing and Unboxing with an example

Autoboxing: Converting a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

Passed as a parameter to a method that expects an object of the corresponding wrapper class.

Assigned to a variable of the corresponding wrapper class

Unboxing: Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example conversion of Integer to int. The Java compiler applies unboxing when an object of a wrapper class is:Passed as a parameter to a method that expects a value of the corresponding primitive type.Assigned to a variable of the corresponding primitive type. EX:

```
package files_exceptionhandling;

public class AutoboxingUnboxing {

        public static void main(String[] args) {

                        Integer i = new Integer(10);        // creating an Integer Object

                        int i1 = i;  // unboxing the Object

                        System.out.println("value of i: " + i);
```

```
                              System.out.println("value of i1: " + i1);

                              char ch = 'g';

                              Character gfg =ch;  //Autoboxing of char

                              System.out.println("value of ch: " + ch);

                              System.out.println("value of gfg: " + gfg);           }
```

7. What is the output of the following programs ? Explain in detailed

(a)

```
import java.util.Scanner;

class Division {

  public static void main(String[] args) {

  int a, b, result;

  Scanner input = new Scanner(System.in);

  System.out.println("Input two integers");

  a = input.nextInt();

  b = input.nextInt();

  result = a / b;

  System.out.println("Result = " + result);

  } }
```

Output : if we give b value zero then there is an exception occure that is

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

(b)

```
class Division {

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

```
  int a, b, result;

  Scanner input = new Scanner(System.in);

  System.out.println("Input two integers");

  a = input.nextInt();

  b = input.nextInt();

  try {

            result  = a / b;

            System.out.println("Result = " + result);

  }

  catch (ArithmeticException e) {

            System.out.println("Exception caught: Division by zero.");

  }

 }

}
```

Output : if we give the value for b is zero then the output will be Exception caught: Division by zero

(c)

```
class Exceptions {

  public static void main(String[] args) {

  String languages[] = { "C", "C++", "Java", "Perl", "Python" };

  try {

            for (int c = 1; c <= 5; c++) {

            System.out.println(languages[c]); }

  }

  catch (Exception e) {
```

```
            System.out.println(e);
  }
  }
}
```

Output : Exception is , java.lang.ArrayIndexOutOfBoundsException: 5

8.What is Serialization and DeSerialization in java? explain with example

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.

The ObjectOutputStream class contains **writeObject()** method for serializing an object.

Example:

```
import java.io.*;
 class Demo implements java.io.Serializable {
     public int a;
     public String b;
     // Default constructor
     public Demo(int a, String b)  {
         this.a = a;
         this.b = b;
     }
 }
 class Test {
```

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

    Demo object = new Demo(1, "geeksforgeeks");

    String filename = "file.ser";

     try       // Serialization

    {

        //Saving of object in a file

        FileOutputStream file = new FileOutputStream(filename);

        ObjectOutputStream out = new ObjectOutputStream(file);

         // Method for serialization of object

        out.writeObject(object);

        out.close();

        file.close();

        System.out.println("Object has been serialized");

    }

    catch(IOException ex)

    {

        System.out.println("IOException is caught");

    }

    Demo object1 = null;

    try         // Deserialization

    {

        FileInputStream file = new FileInputStream(filename);

        ObjectInputStream in = new ObjectInputStream(file);

        // Method for deserialization of object
```

```
            object1 = (Demo)in.readObject();

            in.close();

            file.close();

            System.out.println("Object has been deserialized ");

            System.out.println("a = " + object1.a);

            System.out.println("b = " + object1.b);

        }

        catch(IOException ex)

        {

            System.out.println("IOException is caught");

        }

        catch(ClassNotFoundException ex)

        {

            System.out.println("ClassNotFoundException is caught");

        }

    }

}
```

9.What is Byte stream in java

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;   // Accessing FileReader, FileWriter, IOException

public class ByteStream
```

```
{
    public static void main(String[] args) throws IOException
    {
           try      {
           FileReader  sourceStream = new FileReader("test.txt");
            // Reading source file and writing content to
            // target file character by character.
            int temp;
            while ((temp = sourceStream.read()) != -1)
                 System.out.println((char)temp);
        }
        finally    {
            // Closing stream as no longer in use
            if (sourceStream != null)
                sourceStream.close();
        }
    }
}
```

23. What are annotations in java? Explain with an example

Annotations are used to provide supplement information about a program.

Annotations start with '@'.Annotations do not change action of a compiled program.Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.Annotations are not pure comments as they can change the way a program is treated by compiler. See below code for example.

```
/* Java program to demonstrate that annotations are  not barely comments (This program throws
compiler  error because we have mentioned override, but not overridden, we haver overloaded display)
*/

class Base {

    public void display()

    {

        System.out.println("Base display()");

    }

}

class Derived extends Base {

    @Override

    public void display(int x)

    {

        System.out.println("Derived display(int )");

    }

    public static void main(String args[])

    {

        Derived obj = new Derived();

        obj.display();

    }

}
```

10. What is InputStream and What is OutputStream

InputStream : Inputstream is used to read data from the source

Outputstream : outputstream is used to write the data to a destination

11. Mention about the methods used in FileInputStream and FileOutputStream

FileInputStream:

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.

5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

Fileoutputstream :

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.

4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

15. Mention about the methods in Java.io.File Class in Java

1	**boolean canExecute()** This method tests whether the application can execute the file denoted by this abstract pathname.
2	**boolean canRead()** This method tests whether the application can read the file denoted by this abstract pathname.
3	**boolean canWrite()** This method tests whether the application can modify the file denoted by this abstract pathname.
4	**int compareTo(File pathname)** This method compares two abstract pathnames lexicographically.

5	**boolean createNewFile()** This method atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
6	**static File createTempFile(String prefix, String suffix)** This method creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
7	**static File createTempFile(String prefix, String suffix, File directory)** This method Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
8	**boolean delete()** This method deletes the file or directory denoted by this abstract pathname.
9	**void deleteOnExit()** This method requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
10	**boolean equals(Object obj)** This method tests this abstract pathname for equality with the given object.

11	**boolean exists()** This method tests whether the file or directory denoted by this abstract pathname exists.
12	**File getAbsoluteFile()** This method returns the absolute form of this abstract pathname.
13	**String getAbsolutePath()** This method returns the absolute pathname string of this abstract pathname.
14	**File getCanonicalFile()** This method returns the canonical form of this abstract pathname.
15	**String getCanonicalPath()** This method returns the canonical pathname string of this abstract pathname.
16	**long getFreeSpace()** This method returns the number of unallocated bytes in the partition named by this abstract path name.
17	**String getName()** This method returns the name of the file or directory denoted by this abstract pathname.

18	**String getParent()** This method returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
19	**File getParentFile()** This method returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
20	**String getPath()** This method converts this abstract pathname into a pathname string.
21	**long getTotalSpace()** This method returns the size of the partition named by this abstract pathname.
22	**long getUsableSpace()** This method returns the number of bytes available to this virtual machine on the partition named by this abstract pathname.
23	**int hashCode()** This method computes a hash code for this abstract pathname.
24	**boolean isAbsolute()**

	This method tests whether this abstract pathname is absolute.
25	**boolean isDirectory()** This method tests whether the file denoted by this abstract pathname is a directory.
26	**boolean isFile()** This method tests whether the file denoted by this abstract pathname is a normal file.
27	**boolean isHidden()** This method tests whether the file named by this abstract pathname is a hidden file.
28	**long lastModified()** This method returns the time that the file denoted by this abstract pathname was last modified.
29	**long length()** This method returns the length of the file denoted by this abstract pathname.
30	**String[] list()** This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

31	**String[] list(FilenameFilter filter)** This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
32	**File[] listFiles()** This method returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
33	**File[] listFiles(FileFilter filter)** This method returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
34	**File[] listFiles(FilenameFilter filter)** This method returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
35	**static File[] listRoots()** This method lists the available filesystem roots.
36	**boolean mkdir()** This method creates the directory named by this abstract pathname.

37	**boolean mkdirs()** This method creates the directory named by this abstract pathname, including any necessary but non existent parent directories.
38	**boolean renameTo(File dest)** This method renames the file denoted by this abstract pathname.
39	**boolean setExecutable(boolean executable)** This is a convenience method to set the owner's execute permission for this abstract pathname.
40	**boolean setExecutable(boolean executable, boolean ownerOnly)** This method Sets the owner's or everybody's execute permission for this abstract pathname.
41	**boolean setLastModified(long time)** This method sets the last-modified time of the file or directory named by this abstract pathname.
42	**boolean setReadable(boolean readable)** This is a convenience method to set the owner's read permission for this abstract pathname.
43	**boolean setReadable(boolean readable, boolean ownerOnly)**

	This method sets the owner's or everybody's read permission for this abstract pathname.
44	**boolean setReadOnly()** This method marks the file or directory named by this abstract pathname so that only read operations are allowed.
45	**boolean setWritable(boolean writable)** This is a convenience method to set the owner's write permission for this abstract pathname.
46	**boolean setWritable(boolean writable, boolean ownerOnly)** This method sets the owner's or everybody's write permission for this abstract pathname.
47	**String toString()** This method returns the pathname string of this abstract pathname.
48	**URI toURI()** This method constructs a file : URI that represents this abstract pathname.

17. What is garbage collection?

In java, garbage means unreferenced objects.

Advantage of Garbage Collection

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?
There are many ways:

- By nulling the reference
- By assigning a reference to another

By anonymous object etc.Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

20. What is Enumeration? Explain with an example

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).

Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.

In Java (from 1.5), enums are represented using enum data type. Java enums are more powerful than C/C++ enums . In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types)

Example :

```
Enum Color {
               RED(5), GREEN(3), BLUE(9);
               private int j;
               Color(int i){
                    this.j=i;
               }
             }
             public class EnumEx
             {
               public static void main(String[] args)
               {
                 Color arr[] = Color.values();      // Calling values()
                 for (Color col : arr)    // enum with loop
                 {
                   // Calling ordinal() to find index of color
                   System.out.println(col + " at index " + col.ordinal());
                 }
                 // Using valueOf(). Returns an object of
                 // Color with given constant.
                 // Uncommenting second line causes exception
                 // IllegalArgumentException
                 System.out.println(Color.valueOf("RED"));
```

```
                    // System.out.println(Color.valueOf("WHITE"));
                  }
          }
.
```