4.Exception and File Handling

Exception handling in Java allows developers to manage runtime errors effectively by using mechanisms like try-catch block, finally block, throwing Exceptions, Custom Exception handling, etc.

An Exception is an unexpected event that occurs during the execution of a program (i.e., at runtime) and disrupts the normal flow of the program’s instructions. It occurs when something unexpected things happen, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.

Exception in Java is an error condition that occurs when something wrong happens during the program execution.

Example of Arithmetic Exception

```
public static void main(String[] args)
    {
        int n = 10;
        int m = 0;
        int ans = n / m;
        System.out.println("Answer: " + ans);
    }
```

Types of errors

1.Syntax error

2.Logical error

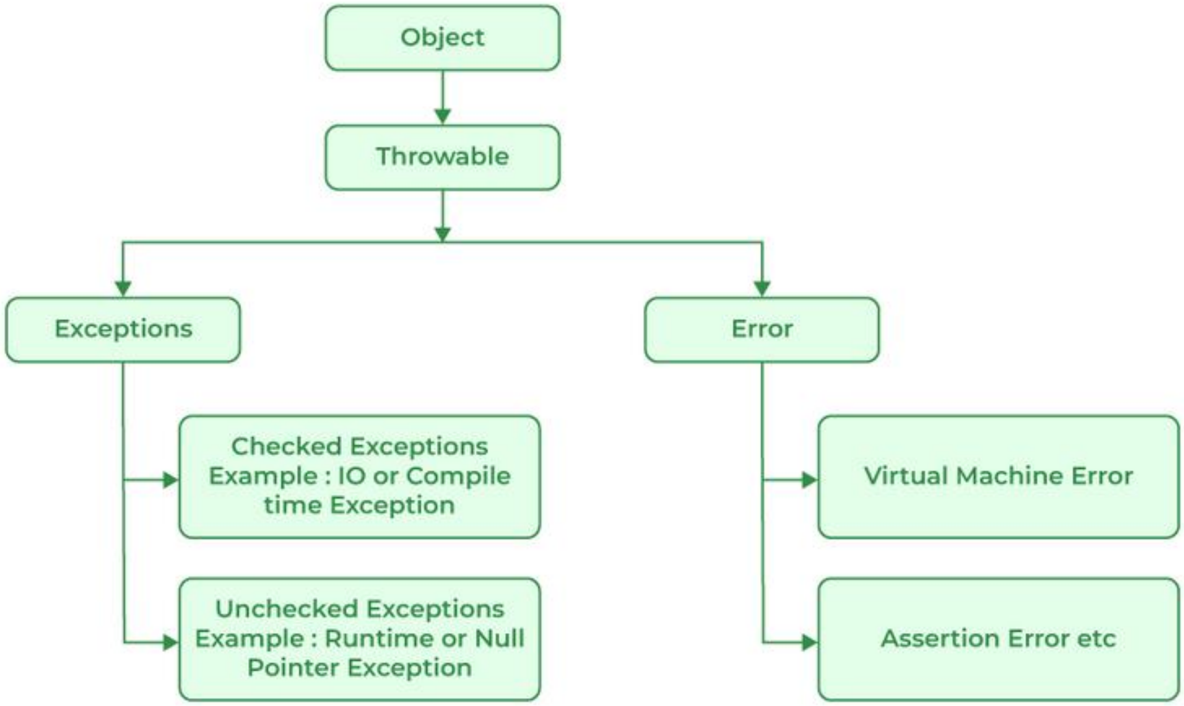
3.Another kind of errors are called exception.

Causes of error

1. Invalid user input
2. Device failure
3. Loss of network connection
4. Physical limitations (out-of-disk memory)
5. Code errors
6. Out of bound
7. Null reference
8. Type mismatch
9. Opening an unavailable file
10. Database errors
11. Arithmetic errors

Java Exception Hierarchy

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

Difference Between Exception and Error

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

Error	**Exception**
Caused by issues with the JVM or hardware.	Caused by conditions in the program such as invalid input or logic errors.
OutOfMemoryError StackOverFlowError	IOException NullPointerException

1.1 Checked Exceptions

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

1. **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because its not present in the correct location or it is missing from the project.
2. **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
3. **IOException:** Throws when input/output operation fails

4. **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
5. **SQLException:** Throws when there’s an error with the database.
6. **FileNotFoundException:** Thrown when the program tries to open a file that doesn’t exist

1.2 Unchecked Exceptions

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error. Examples of Unchecked Exception are listed below:

1. **ArithmeticException:** It is thrown when there’s an illegal math operation.

2**.NullPointerException:** It is thrown when you try to use a null object (e.g. accessing its methods or fields)

3.ArrayIndexOutOfBoundsException: It occurs when we try to access an array element with an invalid index.

}

4. **ArrayStoreException:** It happens when you store an object of the wrong type in an array.

Exception Handling Keywords

try,catch,throw,throws,finally

try-catch block

This ensures that the application continues to run even if an error occurs. The code inside the try block is executed, and if any exception occurs, it is then caught by the catch block.

Syntax of try Catch Block

```
try
 {

 // Code that might throw an exception

}
catch (ExceptionType e)
 {

 // Code that handles the exception

}
```

finally: The “finally block” is used in exception handling to ensure that a certain piece of code is always executed whether an exception occurs or not.

Syntax:

```
try
 {
    // Code that might throw an exception
}
 catch (ExceptionType e)
{
    // Code to handle the exception
}
finally
{
 // Code that will always execute
}
```

Example:

```
import java.util.*;
class Main {

    public static void main(String[] args) {
        int a,b,c;
Scanner sc=new Scanner(System.in);

        try {
System.out.println("try block");
a=sc.nextInt();
b=sc.nextInt();
c=a/b;
System.out.println("Quotient"+c);

         }
         catch (ArithmeticException e) {
              System.out.println(
                  "Caught ArithmeticException: " + e);
         }
        finally
        {
            System.out.println("always execute");
        }

    }
}
```

Multiple Catch blocks

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Syntax:

```
try
 {
    // Code that might throw an exception
}
 catch (ExceptionType1 variable)
{
    // Code to handle the exception1
}
catch (ExceptionType2 variable)
{
    // Code to handle the exception2
}
```

Example:

```
class Main {
    public static void main(String[] args) {
         try{
               int a[]=new int[5];
               System.out.println(a[10]);
              }
```

```
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }

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

Nested try-Catch

One try-catch block can have another try-catch block inside it.

Syntax:

```
try
{
try
{
//code
}catch(exceptiontype object)
{
//code
}
}
Catch(exceptiontype object)
{
//code
}
```

Example:

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
    try{
     System.out.println("going to divide by 0");
     int b =39/0;
    }
    //catch block of inner try block 1
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }
    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }
    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
    System.out.println("handled the exception (outer
catch)");
  }
  System.out.println("normal flow..");
 }
}
```

```
}
```

Declaring Exceptions

Checked exceptions follow handle or declare rule,means either handle by the programmer or if not possible to handle,declare that the method can throw the exception.

Syntax:

```
modifier returntype methodname(arguments) throws
exceptiontype1,exceptiontype2…
{
//method body
}
```

User Defined Exceptions

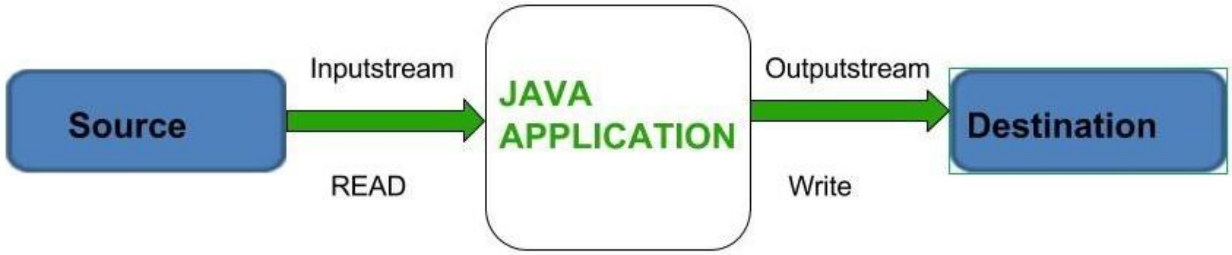
we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Syntax:

```
class userdefinedexception extends Exception
{
//code
}
```

File and Streams

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc. to fully execute the I/O operations.



Stream

A stream can be defined as a sequence of data. There are two kinds of Streams.

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

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

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen.

Standard Input − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output − This is used to output the data produced by the user's program and represented as System.out.

Standard Error − This is used to output the error data produced by the user's program and represented as System.err.

Types of streams

1.Byte oriented stream: used for reading and writing data in the form of raw byte.

2.Character oriented stream: Used for reading and writing characters.

Difference Between Byte Stream and Character Stream

Byte Stream	**Character Stream**
Byte stream is used to perform input and output operations of 8-bit bytes.	Character stream is used to perform input and output

Byte Stream	**Character Stream**
	operations of 16-bit Unicode.
It processes data byte by byte.	It processes data character by character.
Common classes for Byte stream are FileInputStream and FileOutputStream.	Common classes for Character streams are FileReader and FileWriter.
Example- Byte streams are used to read or write binary data.	Example- Character streams are used to read/write characters.

The InputStream and OutputStream classes (abstract) are the super classes of all the input/output stream classes: classes that are used to read/write a stream of bytes. Following are the byte array stream classes provided by Java −

InputStream	**OutputStream**
FileInputStream	FileOutputStream
ByteArrayInputStream	ByteArrayOutputStream
ObjectInputStream	ObjectOutputStream
PipedInputStream	PipedOutputStream

InputStream	**OutputStream**
FilteredInputStream	FilteredOutputStream
BufferedInputStream	BufferedOutputStream
DataInputStream	DataOutputStream

The Reader and Writer classes (abstract) are the super classes of all the character stream classes: classes that are used to read/write character streams. Following are the character array stream classes provided by Java −

Reader	**Writer**
BufferedReader	BufferedWriter
CharacterArrayReader	CharacterArrayWriter
StringReader	StringWriter
FileReader	FileWriter
InputStreamReader	InputStreamWriter
FileReader	FileWriter

File handling

Input data entered through the console and output data displayed on the screen are both lost after program terminates.If we need input or output data later,it must be strored in a file.

File Operations

The following are the several operations that can be performed on a file in Java:

- Opening file
- Closing a File
- Read from a File
- Write to a File
- Updating contents of an existing file
- Delete a File
- Accessing file properties

File Class

In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used to create an object of the class and then specifying the name of the file.

How to Create a File Object?

A File object is created by passing in a string that represents the name of a file, a String, or another File object. For example,

```
File a = new File("/usr/local/bin/a.txt");
```

Example:

```
class CheckFileExist
{
    public static void main(String[] args)
    {
        // pass the filename or directory name to File
        // object
        File f = new File(fname);

        // apply File class methods on File object
        System.out.println("File name :" + f.getName());
        System.out.println("Path: " + f.getPath());
        System.out.println("Absolute path:" +
f.getAbsolutePath());
        System.out.println("Parent:" + f.getParent());
        System.out.println("Exists :" + f.exists());
        if (f.exists()) {
            System.out.println("Is writable:" + f.canWrite());
            System.out.println("Is readable" + f.canRead());
            System.out.println("Is a directory:" + 
f.isDirectory());
            System.out.println("File Size in bytes " + 
f.length());
        }
    }
}
```

Reading and writing data using files

File I/O can be performed using byte as well as character oriented streams.

Use stream class methods read() and write() with the file stream object.

FileInputStream

This class reads the data from a specific file (byte by byte).

To read the contents of a file using this class −

```
FileInputStream inputStream = new 
FileInputStream("file_path");
```

or,

```
File file = new File("file_path");
FileInputStream inputStream = new
FileInputStream(file);
```

read() Method

read() - reads a single byte from the file

read(byte[] array) - reads the bytes from the file and stores in the specified array.

read(byte[] array, int start, int length) - reads the number of bytes equal to length from the file and stores in the specified array starting from the position start.

Example:

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

     FileInputStream input = new
FileInputStream("input.txt");

     System.out.println("Data in the file: ");
     // Reads the first byte
     int i = input.read();
    while(i != -1) {
       System.out.print((char)i);
```

```
        // Reads next byte from the file
        i = input.read();
      }
      input.close();
    }
```

FileOutputStream

This writes data into a specific file (byte by byte).

To write the contents of a file using this class −

```
FileOutputStream outputStream = new
FileOutputStream("file_path");
```

or,

```
File file = new File("file_path");
FileOutputStream outputStream = new
FileOutputStream (file);
```

write() Method

write() - writes the single byte to the file output stream

write(byte[] array) - writes the bytes from the specified array to the output stream

write(byte[] array, int start, int length) - writes the number of bytes equal to length to the output stream from an array starting from the position start.

Example:

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

        String data = "This is a line of text inside the file.";

            FileOutputStream output = new
FileOutputStream("output.txt");

            byte[] array = data.getBytes();

            // Writes byte to the file
            output.write(array);

            output.close();
        }

    }
```

The getBytes() method used in the program converts a string into an array of bytes.

BufferedInputStream

The BufferedInputStream class of the java.io package is used with other input streams to read the data (in bytes) more efficiently.

During the read operation in BufferedInputStream, a chunk of bytes is read from the disk and stored in the internal buffer. And from the internal buffer bytes are read individually.

Hence, the number of communication to the disk is reduced. This is why reading bytes is faster using the BufferedInputStream.

Create a BufferedInputStream

In order to create a BufferedInputStream, we must import the java.io.BufferedInputStream package first. Once we import the package here is how we can create the input stream.

```
// Creates a FileInputStream
FileInputStream file = new FileInputStream(String path);

// Creates a BufferedInputStream
BufferedInputStream buffer = new
BufferInputStream(file);
```

available() Method
To get the number of available bytes in the input stream, we can use the available() method.

skip() Method
To discard and skip the specified number of bytes, we can use the skip() method.

close() Method
To close the buffered input stream, we can use the close() method.

mark() mark the current position in input stream up to which data has been read.

reset() returns the control to the point in the input stream where the mark was set.

BufferedOutputStream
The BufferedOutputStream class of the java.io package is used with other output streams to write the data (in bytes) more efficiently.

During the write operation, the bytes are written to the internal buffer instead of the disk. Once the buffer is filled or the stream is closed, the whole buffer is written to the disk.

Create a BufferedOutputStream

In order to create a BufferedOutputStream, we must import the java.io.BufferedOutputStream package first. Once we import the package here is how we can create the output stream.

```
// Creates a FileOutputStream
FileOutputStream file = new FileOutputStream(String
path);

// Creates a BufferedOutputStream
BufferedOutputStream buffer = new
BufferOutputStream(file);
```

flush() Method

To clear the internal buffer, we can use the flush() method. This method forces the output stream to write all data present in the buffer to the destination file.

DataInputStream and DataOutputStream

A data input stream enables an application to read primitive Java data types from an input stream instead of raw bytes. That is why it is called DataInputStream – because it reads data (numbers) instead of just bytes.

An application uses a data output stream to write data that can later be read by a data input stream.

Methods:readShort(),readFloat(),readChar(),readInt(),readDouble(),readBoolean(),readLong(),writeInt(),writeFloolean(),etc.

Example:

```
DataOutputStream dout =
                    new DataOutputStream(new
FileOutputStream("file.dat")) ) {

          dout.writeDouble(1.1);
          dout.writeInt(55);
          dout.writeBoolean(true);
          dout.writeChar('4');

DataInputStream din =
                    new DataInputStream(new
FileInputStream("file.dat")) ) {
```

```
            double a = din.readDouble();
             int b = din.readInt();
             boolean c = din.readBoolean();
             char d = din.readChar();
             System.out.println("Values: " + a + " " + b + " "
+ c + " " + d);
        }
```

FileReader and FileWriter

The FileReader class of the java.io package can be used to read data (in characters) from files.

Create a FileReader

In order to create a file reader, we must import the java.io.FileReader package first. Once we import the package, here is how we can create the file reader.

1. Using the name of the file

```
FileReader input = new FileReader(String name);
```

Here, we have created a file reader that will be linked to the file specified by the name.

2. Using an object of the file

```
FileReader input = new FileReader(File fileObj);
```

Example:

```
char[] array = new char[100];

    // Creates a reader using the FileReader
    FileReader input = new FileReader("input.txt");

    // Reads characters
    input.read(array);
    System.out.println("Data in the file: ");
    System.out.println(array);

    // Closes the reader
    input.close();
```

The FileWriter class of the java.io package can be used to write data (in characters) to files.

Create a FileWriter

In order to create a file writer, we must import the Java.io.FileWriter package first. Once we import the package, here is how we can create the file writer.

1. Using the name of the file

```
FileWriter output = new FileWriter(String name);
```

Here, we have created a file writer that will be linked to the file specified by the name.

2. Using an object of the file

```
FileWriter  input = new FileWriter(File fileObj);

String data = "This is the data in the output file";

    // Creates a FileWriter
    FileWriter output = new FileWriter("output.txt");

    // Writes the string to the file
    output.write(data);

    // Closes the writer
    output.close();
```