UNIT III EXCEPTION HANDLING AND I/O 9

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 – Reading and Writing Files

3.1 EXCEPTIONS

A Java exception is an object that finds an exceptional condition occurs from a piece of code. An exception object is created and thrown to the method from the code where an exception is found.

Types of Exceptions

There are two types of exceptions

1. Predefined Exceptions-The Exceptions which are predefined are called predefined exceptions
2. Userdefined Exceptions- The Exceptions which are defined by the user are called userdefined exceptions

Purpose of exception handling

The main purpose of exception handling mechanism is used to detect and report an "exceptional circumstance" so that necessary action can be taken. It performs the following tasks

1. Find the problem(Hit the exception)
2. Inform that an error occurred(throw the exception).
3. Receive the error information(Catch the exception)
4. Take corrective actions(Handle the exception)

Java exception handling is managed via five keywords.

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

1.try block:

Program statements that need to be monitored for exceptions are placed within a try block.

2.catch block:

If an exception occurs within a try block, it is thrown to the catch block. This block can catch this exception and handle it

3.throw:

Some exceptions are automatically thrown by the Java run time system. To throw the exceptions manually, we can use the keyword throw.

4.throws:

Any exception that is thrown out of a method is specified as such by a throws clause.

5.finally:

finally blocks contains any code that need to be executed after a try block completes .

This is the general form of an exception-handling block:

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
// ...
finally
{
// block of code to be executed after try block ends
}
```

ExcetptionType is the type of exception that has occurred. exOb is an exception object.

3.2 EXCEPTION HIERARCHY

Throwable is a superclass for all exception types. Thus, Throwable is at top of the exception hierarchy.

There are two subclasses under Throwable class.

1. Exception
2. Error

1. Exception:

This class is used for exceptional conditions that user programs should catch. We can also subclass this class to create own custom exception types. The important subclass of this class is **RuntimeException**.

RuntimeException

Exceptions of this type are automatically defined for the programs that you write and include thing such as division by zero and invalid array indexing.

2.Error

The another subclass is by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

Eg: Stack overflow is an example of such an error.

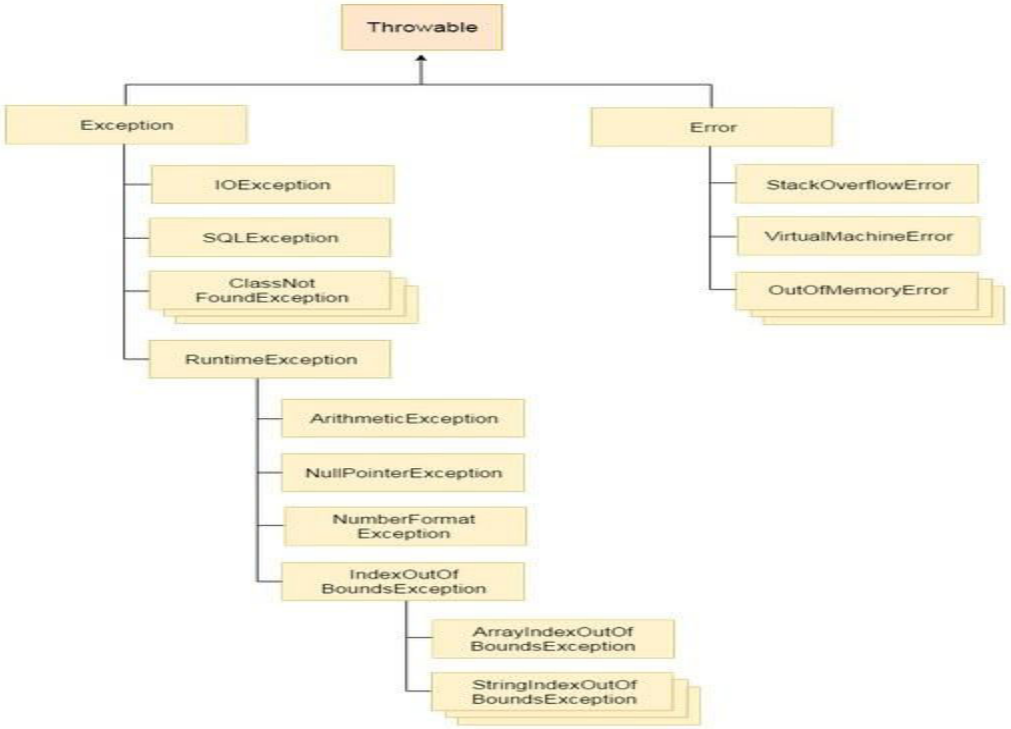


Figure 3.1 Exception Hierarchy

3.3 THROWING AND CATCHING EXCEPTIONS

3.3.1 Using try and catch

The try block allows us to fix the errors. Catch block prevents the program from automatically terminating. To handle a run-time error, enclose the code to be monitored inside a try block. After the try block, include a catch block that specifies the exception type that needs to be caught.

Syntax:

```
try
{
statement;
```

```
}
catch(Exception-type exOb)
{
statement;
}
```

The try block can have one or more statements that would generate an exception. If one statement of the try block generates an exception, the remaining statements in the block are not be executed and execution jumps to the catch that is placed next to the try block.

Example Program:

```
class Ex1
{
public static void main(String args[])
{
int a,b;
try
{
// monitor a block of code.
a = 0;
b = 42 / a;
System.out.println("This will not be printed.");
}
catch(ArithmeticException e)
{
// catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

Output:

```
Division by zero.
After catch statement.
```

Example Program:

```
class Ex2
{
public static void main(String[] args)
{
int numbers[] = { 1, 2, 3, 4, 5 };
try
```

```
{
for (int c = 1; c <= 5; c++)
{
System.out.println(numbers[c]);
}
}
catch (Exception e)
{
System.out.println(e);
}
}
}
```

Output:

```
2
3
4
5
java.lang.ArrayIndexOutOfBoundsException: 5
```

3.3.2 Multiple catch Clauses

In some situation, more than one exception can occur by a single piece of code. To handle this situation, we can use two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch block is executed in order, and the first one whose type matches that exception is executed. After one catch block executes, the others are bypassed, and continues after the try/catch block.

Syntax:

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
```

Example Program:

```
class Ex3
```

```
{
public static void main(String args[])
{
try
{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("task1 is completed");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 completed");
}
catch(Exception e)
{
System.out.println("common task completed");
}
System.out.println("rest of the code...");
}
}
```

Output:

```
task1 completed
rest of the code...
```

Example Program:

```
class Ex4
{
public static void main(String args[])
{
try
{
int s = args.length;
System.out.println("s = " + s);
int b = 42 / s;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
```

```
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Output:

```
s = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

java Ex4 TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

This program will cause a division-by-zero exception if it is started with no command line arguments, since **s** will equal zero. It will survive the division if you provide a command line argument, setting **s** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

3.3.3 Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

Syntax:

```
//Main try block
try
{
statement 1;
statement 2;
```

```
//try-catch block inside another try block
try
{
statement 3;
statement 4;
//try-catch block inside nested try block
try
{
statement 5;
statement 6;
}
catch(Exception e2)
{
//Exception Message
}
}
catch(Exception e1)
{
//Exception Message
}
}
//Catch of Main(parent) try block
catch(Exception e3)
{
//Exception Message
}
```

Example Program:

```
class Nest
{
public static void main(String args[]){
//Parent try block
try
{
//Child try block1
try
{
System.out.println("Inside block1");
int b =45/0;
System.out.println(b);
}
catch(ArithmeticException e1)
{
System.out.println("Exception: e1");
```

```
}
//Child try block2
try
{
System.out.println("Inside block2");
int b =45/0;
System.out.println(b);
}
catch(ArrayIndexOutOfBoundsException e2)
{
System.out.println("Exception: e2");
}
System.out.println("Just other statement");
}
catch(ArithmeticException e3)
{
System.out.println("Arithmetic Exception");
System.out.println("Inside parent try catch block");
}
catch(ArrayIndexOutOfBoundsException e4)
{
System.out.println("ArrayIndexOutOfBoundsException");
System.out.println("Inside parent try catch block");
}
catch(Exception e5)
{
System.out.println("Exception");
System.out.println("Inside parent try catch block");
}
System.out.println("Next statement..");
}
}
```

Output:

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

3.3.4 throw statement

To throw an exception explicitly, we can use **throw** keyword.

Syntax:

throw new exception_class("error message");

For example:
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");

Example Program:

```
class ThrowDemo
{
static void demoproc()
{
try
{
throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[])
{
try
{
demoproc();
}
catch(NullPointerException e)
{
System.out.println("Recaught: " + e);
}
}
}
```

output:
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

3.3.5 throws clause

Using throws clause, We can list the types of exceptions that a method might throw.The exceptions which are thrown in a method might be using **throws** clause. If they are not, a compile-time error will result.

Syntax:

type method-name(parameter-list) throws exception-list
{
// body of method
}
Exception-list is a comma-separated number of exceptions that a method can throw.

Example Program:

```
class ThrowsDemo
{
static void throwOne() throws IllegalAccessException
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
try
{
throwOne();
}
catch (IllegalAccessException e)
{
System.out.println("Caught " + e);
}
}
}
```

Output:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

3.3.6 finally

finally creates a block of code that is to be executed after a try/catch block has completed its execution.The finally block will execute if an exception is thrown or not thrown. The finally clause is optional.Each try block requires either one catch or a finally clause**.**

Syntax:

```
try
{
   //Statements that may cause an exception
```

```
}
catch
{
  //Handling exception
}
finally
{
  //Statements to be executed
}
```

Example Program:

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

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

Output:

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

3.4 Built-in Exceptions

Several exception classes are defined in the standard package java.lang.These exceptions are subclasses of the standard type RuntimeException.

There are two types of exceptions

1. Unchecked Exceptions
2. Checked Exceptions

1.Unchecked Exceptions

These are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either

handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

2.Checked Exceptions

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

3.4.1 Unchecked Exceptions

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

Table 3.1 Java Unchecked RuntimeException subclasses

3.4.2 Checked Exceptions

Exception	Meaning

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

Table 3.2 Java checked RuntimeException subclasses

3.5 Creating Own Exceptions

We can throw our own exceptions using throw keyword.

Syntax:

throw new Throwable_subclass;

Eg:

throw new ArithmeticException;

Example Program1:

```
import java.lang.Exception;
class MyownException extends Exception
{
MyownException(String mes)
{
super(mes);
}
}
class TestException
{
public static void main(String args[])
{
int a=5,b=1000;
try
{
float c=(float)a/(float)b;
if(c<0.01)
{
throw new MyownException("number is too small");
}
}
catch (MyownException e)
```

```
{
System.out.println("caught my exception");
System.out.println(e.getMessage());
}
finally
{
System.out.println("This is a finally block");
}
}
}
```

Output:

```
caught my exception
number is too small
This is a finally block
```

Example Program2

```
class MyownException2 extends Exception
{
private int value;
MyownException2(int x)
{
value = x;
}
public String toString()
{
return "MyownException2[" + value + "]";
}
}
class DemoException
{
static void compute(int x) throws MyownException2
{
System.out.println("Called compute(" + x + ")");
if(x>10)
throw new MyownException2(x);
System.out.println("Normal exit");
}
public static void main(String args[])
{
try
{
compute(1);
```

```
compute(20);
}
catch (MyownException2 e)
{
System.out.println("Caught " + e);
}
}
}
```

Output:
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]

3.6 Stack Trace Elements

The StackTraceElement is a class that describes a single stack frame,which is an element of a stack trace when an exception occurs.The getStackTrace() method is used to return an array of StackTraceElements.
Each stack frame contains the following

1. the class name
2. the method name
3. The file name
4. And the source-code line number

StackTraceElement Constructor:
StackTraceElement(String className,String methName,string fileName,int line)

Parameters:
className-The name of the class
methName-The name of the method
filename-The name of the file
line-The line number is passed

Method	**Description**
boolean equals(Object *ob*)	Returns **true** if the invoking **StackTraceElement** is the same as the one passed in *ob*. Otherwise, it returns **false**.
String getClassName( )	Returns the name of the class in which the execution point described by the invoking **StackTraceElement** occurred.
String getFileName(	Returns the name of the file in which the source code

)	of the execution point described by the invoking **StackTraceElement** is stored.
int getLineNumber( )	Returns the source-code line number at which the execution point described by the invoking **StackTraceElement** occurred. In some situations, the line number will not be available, in which case a negative value is returned.
String getMethodName( )	Returns the name of the method in which the execution point described by the invoking **StackTraceElement** occurred.
int hashCode( )	Returns the hash code for the invoking **StackTraceElement**.
boolean isNativeMethod( )	Returns **true** if the execution point described by the invoking **StackTraceElement** occurred in a native method. Otherwise, it returns **false**.
String toString( )	Returns the **String** equivalent of the invoking sequence.

Table 3.3 Methods in StackTraceElement class

Methods:

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

Example Program:

```
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
```

2.String getClassName(): Returns the class name of the execution point described by the invoking **StackTraceElement**.

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo1
{
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
```

3.String getFileName(): Returns the file name of the execution point described by the invoking **StackTraceElement**.

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo2
{
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].getFileNam
e());
  }
  }
```

Output:
file name:
Thread.java
StackTraceElementDemo.java

4. 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.

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo3
{
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].getLineNumbe
r());
}
}
```

Output:
line number:
1589
10

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

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo4
{
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].getMethodNa
me());
}
}
```

Output:
method name:
getStackTrace
main

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

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo5
{
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.

Example Program:

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

```
public class StackTraceElementDemo6
{
public static void main(String[] arg)
{
for(int i = 0; i<2; i++)
System.out.println(Thread.currentThread().getStackTrace()[i].isNativeMetho
d());
}
}
```

Output:
false
false

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

Example Program:

```
import java.lang.*;
import java.io.*;
import java.util.*;
public class StackTraceElementDemo7
{
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

3.7 Input / Output Basics

Java's basic I/O system,including I/O is supported by **io** package.

3.7.1 Streams

Java implements streams within class hierarchies defined in the java.io package. Java programs perform input and output operations through streams.A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the java I/O system.The input stream may abstract many different kinds of input: from a disk file,a keyboard,or a network socket. Likewise,an output stream may refer to the console such as a disk file, or a network connection.

There are two types of streams

1. Byte streams
2. Character streams

3.7.2 The Predefined Streams

All Java programs automatically import java.lang package. This package defines a class called System, which contains several aspects of the run-time environment.
System class contains three predefined stream variables:

1. in
2. out
3. err

System.in refers to the standard input stream. System.out refers to the standard output stream.System.err refers to the standard error stream.System.**System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, they are typically used to read and write characters from and to the console.

3.8 Byte Streams

Byte Streams provides a convenient way for handling input and output of bytes. When reading or writing binary data, byte streams are used.
There are two abstract classes defined in byte streams

1. InputStream
2. OutputStream

Each of these above classes has some subclasses that handle the various devices such as disk files,network connections, and memory buffers.

3.8.1 Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading

	the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 3.4 The Byte Stream I/O Classes in java.io

The abstract classes InputStream and OutputStream define several methods that other stream classes implement.The methods read() and write() are used to read and write bytes of data.

3.9 Character Stream Classes

Character Streams provides a convenient way for handling input and output of characters.

There are two abstract classes defined in byte streams

1. Reader
2. Writer

These abstract classes handle Unicode character streams.

3.9.1 Character Stream Classes

Stream Class	**Meaning**
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer

InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input Stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 3.5 The Character Stream I/O Classes in java.io

The abstract classes Reader and Writer define several methods that other stream classes implement.The methods read() and write() are used to read and write characters of data.

3.10 Reading and Writing Console

In Java, **System.in is used to read console input**. To obtain a character based stream , wrap **System.in** in a **BufferedReader** object. **BufferedReader** refers a buffered input stream.

Constructor:
BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.

Constructor:
InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of **InputStream**, it can be used for *inputStream*. The following reads the input from the keyboard

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
After this statement executes, **br** is a character-based stream that is linked to the console
through **System.in**.

3.10.1 Reading Characters

To read a character from a BufferedReader,We can use read() method.

Syntax:

int read( ) throws IOException

Whenever read() method is called, it reads a character from the input stream and returns an integer value.It returns -1 when the end of the stream is encountered.

Example Program:

```
import java.io.*;
class ReadBR
{
public static void main(String args[]) throws IOException
{
char a;
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
a = (char) br.read();
System.out.println(a);
} while(a != 'q');
}
}
```

Output:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

3.10.2 Reading Strings

To read a string from the keyboard,we can use readLine() method.readLine() is a member of the BufferedReader class.

Syntax:

String readLine( ) throws IOException

It returns a String object.

Example Program:

```
import java.io.*;
class BRReadLines
{
public static void main(String args[]) throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do
{
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

Output:

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
stop
```

3.10.3 Writing Console Output

Console output is normally done with print() and println().These are the methods of PrintStream. System.out is a byte stream, which is useful for output the data.PrintStream is an output stream derived from OutputStream,Which contains write() method to write to the console.

Syntax:

void write(int byteval)

This method is used to rite the byte specified in byteval. byteval is declared as an integer.

Example Program:

```
class WriteDemo
{
public static void main(String args[])
{
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

Output:

A

3.10.4 The PrintWriter Class

PrintWriter is one of the character-based classes.

Constructor:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

ouputStream is an object of OutputStream class.flushOnNewline controls when Java flushes the output stream every time a println() method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline. For example, this line of code creates a **PrintWriter** that is connected to console output:

Example:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Example Program:

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

```
}
}
```

Output:
This is a string
-7
4.5E-7

3.11 Reading and Writing files

Java provides a number of classes and methods that allow you to read and write files.
There are two stream classes

1. **FileInputStream**
2. **FileOutputStream**

These above classes are used to create byte streams linked to files.

FileInputStream(String *fileName*) throws FileNotFoundException
FileOutputStream(String *fileName*) throws FileNotFoundException

Where
fileName specifies the name of the file that want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

3.11.1 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.

Java FileInputStream class declaration

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.

Table 3.6 Java FileInputStream Class Methods

Example Program1:

```
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:

testout.txt
Welcome to Java Stream Classes.

Output:
W

Example Program2

```
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);
}
}
}
```

testout.txt

Welcome to Java Stream Classes.

Output:

Welcome to Java Stream Classes.

3.11.2 Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file. 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

public class FileOutputStream extends OutputStream

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream
void write(byte[] ary)	It is used to write **ary.length** bytes from the

	byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to close the file output stream.

Table 3.7 Java FileOutputStream Class Methods

Example Program1:

```
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...
```

textout.txt

```
A
```

Example Program2:

```
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...

testout.txt
Welcome to Java Stream Classes.

2 Mark Questions and Answers

1. What is an Exception?

A Java exception is an object that finds an exceptional condition occurs from a piece of code. An exception object is created and thrown to the method from the code where an exception is found.

2. Write down the purpose of exception handling mechanism.

The main purpose of exception handling mechanism is used to detect and report an "exceptional circumstance" so that necessary action can be taken. It performs the following tasks

5. Find the problem(Hit the exception)
6. Inform that an error occurred(throw the exception).
7. Receive the error information(Catch the exception)
8. Take corrective actions(Handle the exception)

3. What are the types of exceptions?

There are two types of exceptions

1. Predefined Exceptions-The Exceptions which are predefined are called predefined exceptions

2. Userdefined Exceptions- The Exceptions which are defined by the user are called userdefined exceptions

4. How the exception handling is managed?

Java exception handling is managed via five keywords.

- try
- catch
- throw
- throws and
- finally

5. Write down the general form of an exception-handling block.

The general form of an exception-handling block

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
// ...
finally
{
// block of code to be executed after try block ends
}
```

6. What are the two subclasses under Throwable class?

Throwable is a superclass for all exception types. Thus, Throwable is at top of the exception hierarchy.

There are two subclasses under Throwable class.

3. Exception
4. Error

7. What is an Error?

The another subclass is by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.

Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
Eg: Stack overflow is an example of such an error.

8. What is an Exception?

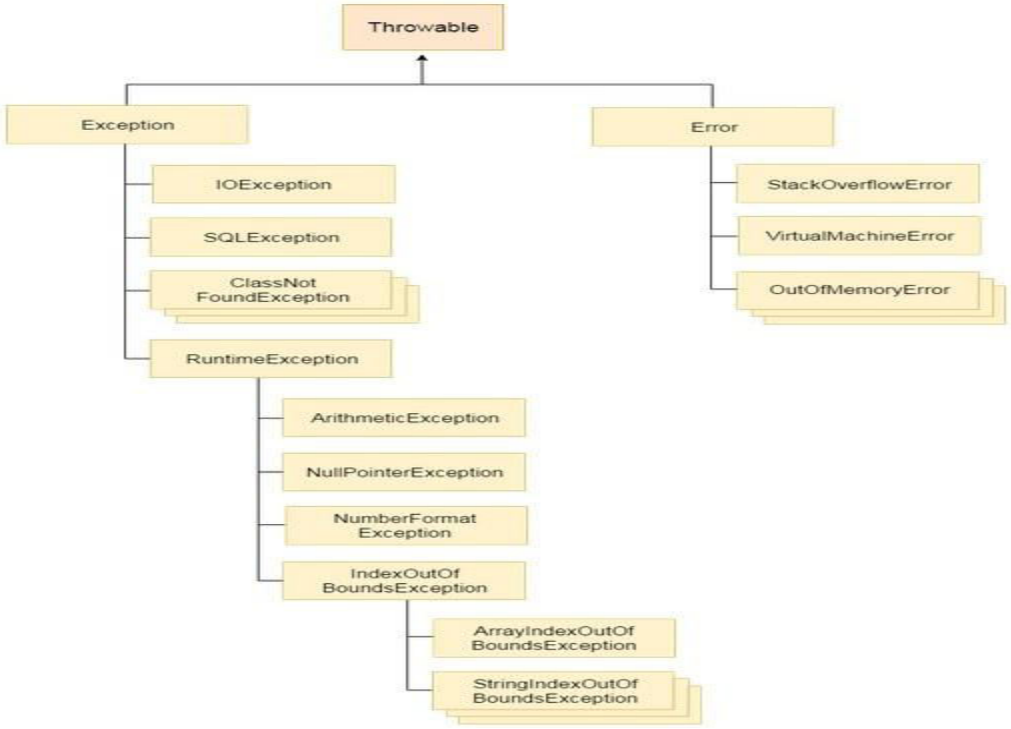
This class is used for exceptional conditions that user programs should catch. We can also subclass this class to create own custom exception types.

The important subclass of this class is **RuntimeException**.

RuntimeException

Exceptions of this type are automatically defined for the programs that you write and include thing such as division by zero and invalid array indexing.

9. Draw the diagrammatical representation for Exception Hierarchy



10. Write down the use of try and catch block in exception handling.

The try block allows us to fix the errors. Catch block prevents the program from automatically terminating. To handle a run-time error, enclose the code to be monitored inside a try block. After the try block, include a catch block that specifies the exception type that needs to be caught.

Syntax:

```
try
{
statement;
```

```
}
catch(Exception-type exOb)
{
statement;
}
```

11. Explain the situation where we need to use multiple catch clauses.

In some situation, more than one exception can occur by a single piece of code. To handle this situation, we can use two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch block is executed in order, and the first one whose type matches that exception is executed.After one catch block executes, the others are bypassed, and continues after the try/catch block.

12. Write down the syntax for multiple catch clauses.

The syntax for multiple catch clauses

```
try
{
// block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType2
}
```

13. Explain the situation where we need to use nested try statements.

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

14. Write down the syntax for nested try statement.

The syntax for nested try statement:

```
//Main try block
```

```
try
{
statement 1;
statement 2;
//try-catch block inside another try block
try
{
statement 3;
statement 4;
//try-catch block inside nested try block
try
{
statement 5;
statement 6;
}
catch(Exception e2)
{
//Exception Message
}
}
catch(Exception e1)
{
//Exception Message
}
}
//Catch of Main(parent) try block
catch(Exception e3)
{
//Exception Message
}
```

15. Write down the use of throw statement.

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

Syntax:

```
throw new exception_class("error message");
```

For example:

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

16. Write down the use of throws clause.

Using throws clause, We can list the types of exceptions that a method might throw.The exceptions which are thrown in a method might be using **throws** clause. If they are not, a compile-time error will result.

Syntax:

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

Exception-list is a comma-separated number of exceptions that a method can throw.

17. Write down the use of finally clause.

finally creates a block of code that is to be executed after a try/catch block has completed its execution.The finally block will execute if an exception is thrown or not thrown. The finally clause is optional.Each try block requires either one catch or a finally clause

Syntax:

try
{
  //Statements that may cause an exception
}
catch
{
  //Handling exception
}
finally
{
  //Statements to be executed
}

18. What is Unchecked Exception?

These are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

19. What is Checked Exception?

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

20. Write down some of the unchecked exceptions in RuntimeException?

The unchecked exceptions in RuntimeException are ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, EnumConstantNotPresentException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, NumberFormatException, SecurityException, StringIndexOutOfBoundsException, TypeNotPresentException, UnsupportedOperationException

21. Write down some of the checked exceptions in RuntimeException?

The checked exceptions in RuntimeException ClassNotFoundException, CloneNotSupportedException, IllegalAccessException, InstantiationException, InterruptedException, NoSuchFieldException, NoSuchMethodException, ReflectiveOperationException

22. Explain the way to create own exceptions.

We can throw our own exceptions using throw keyword.

Syntax:

throw new Throwable_subclass;

Eg:

throw new ArithmeticException;

23. Define Stack Trace Elements.

The StackTraceElement is a class that describes a single stack frame,which is an element of a stack trace when an exception occurs.The getStackTrace() method is used to return an array of StackTraceElements.

Each stack frame contains the following

5. the class name
6. the method name
7. The file name
8. And the source-code line number

24. List out the methods in StackTraceElements

The methods in StackTraceElements are

- boolean equals(Object *ob*)
- String getClassName( )
- String getFileName( )
- int getLineNumber( )
- String getMethodName( )

- int hashCode( )
- boolean isNativeMethod( )
- String toString( )

25. What is meant by a stream?

A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the java I/O system.The input stream may abstract many different kinds of input: from a disk file,a keyboard,or a network socket. Likewise,an output stream may refer to the console such as a disk file, or a network connection.

26. What are the two types of streams?

There are two types of streams

3. Byte streams
4. Character streams

27. What are the predefined stream variables in System class?

System class contains three predefined stream variables:

4. in
5. out
6. err

System.in refers to the standard input stream. System.out refers to the standard output stream.System.err refers to the standard error stream.System.**System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, they are typically used to read and write characters from and to the console.

28. What is meant by a byte stream?

Byte Streams provides a convenient way for handling input and output of bytes. When reading or writing binary data, byte streams are used.

29. What are abstract classes in byte streams?

There are two abstract classes defined in byte streams

3. InputStream
4. OutputStream

30. List out some of the Byte Stream Classes.

BufferedInputStream, BufferedOutputStream, ByteArrayInputStream, ByteArrayOutputStream, DataInputStream, DataOutputStream, FileInputStream, FileOutputStream, InputStream, OutputStream

31. What is meant by a Character stream?

Character Streams provides a convenient way for handling input and output of characters.

32. What are abstract classes in Character streams?

There are two abstract classes defined in byte streams

3. Reader
4. Writer

33. List out some of the Character Stream Classes.

BufferedReader, BufferedWriter, CharArrayReader, CharArrayWriter, FileReader, FileWriter, Reader, Writer

34. How to read a character from BufferedReader Class?

To read a character from a BufferedReader,We can use read() method.

Syntax:

int read( ) throws IOException

Whenever read() method is called, it reads a character from the input stream and returns an integer value.It returns -1 when the end of the stream is encountered.

35. How to read the string from BufferedReader Class?

To read a string from the keyboard,we can use readLine() method.readLine() is a member of the BufferedReader class.

Syntax:

String readLine( ) throws IOException

36. What is the use of PrintWriter class?

PrintWriter is one of the character-based classes.

Constructor:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

ouputStream is an object of OutputStream class.flushOnNewline controls when Java flushes the output stream every time a println() method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline.

37. Write the use of 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.

38. Write down the methods in FileInputStream Class.

The methods in FileInputStream class are

- int available()
- int read()
- long skip(long x)
- FileChannel getChannel()
- FileDescriptor getFD()
- protected void finalize()
- void close()

39. Write the use of FileOutputStream class.

Java FileOutputStream is an output stream used for writing data to a file. 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.

40. Write down the methods in FileOutputStream Class.

The methods in FileOutputStream class are

- protected void finalize()
- void write(byte[] ary)
- void write(int b)
- FileChannel getChannel()
- FileDescriptor getFD()
- void close()

13 Marks Questions

1. Explain the Throwing and Catching Exception.
2. What is exception? How to throw an exception? Give an example.
3. What is finally class? How to catch exceptions? Write an example.
4. What is meant by exceptions? Why it is needed? Describe the exception hierarchy.
5. Write note on Stack Trace Elements. Give example.
6. Define Exception and explain its different types with example.
7. Discuss about character stream classes.
8. With suitable coding discuss all kinds of exception handling.
9. Write a note on java.io package with its stream classes and methods in it.
10. Write a Java program to demonstrate I/O character stream classes.
11. Write short notes on PrintStream class.
12. Explain how user-defined exception subclasses are created in Java.

15 Marks Questions

1. What is the necessity of exception handling? Explain exception handling taking "Divide –by Zero" as an example.
2. What is an exception? Create your own exception for "Temperature>40" in an "Atomic Reactor Based Application" and write appropriate exception handling code in the main program.
3. Write a Java program to copy the data from one file to another file.
4. Write a program that uses a sequence input stream to output the contents of two files.
5. List the five keywords in Java exception handling. Describe the use of the keywords.