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 **StackT raceElement** 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 **StackT raceElement**.
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:

1.1 oolean 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

2.5 tring 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

2.6 tring 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

2.7 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

2.8 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

2.9 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

2.10 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

8.5 tring 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

8.6 Input / Output Basics

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

8.6.1 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

8.6.2 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.

8.7 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.

8.8 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.

8.9 0 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**.

8.9.1 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
```