CHAPTER 03
INPUT/OUTPUT, ERROR HANDLING AND PARSING TEXTUAL FORMATS

- Command-line arguments
- Files and streams
- Errors and exceptions

COMMAND-LINE ARGUMENTS

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.

```
class CommandLineExample{

public static void main(String args[]){

System.out.println("Your first argument is: "+args[0]);

}
}
```

CONVENTIONS FOR COMMAND LINE ARGUMENTS

- Word Arguments

verbose mode

https://www.youtube.com/watch?v=OGkMg-xsE_Q

- Arguments that Require Arguments
- Flags

3.1 FILES AND STREAMS

- The Concept of a Stream
- I/O Streams

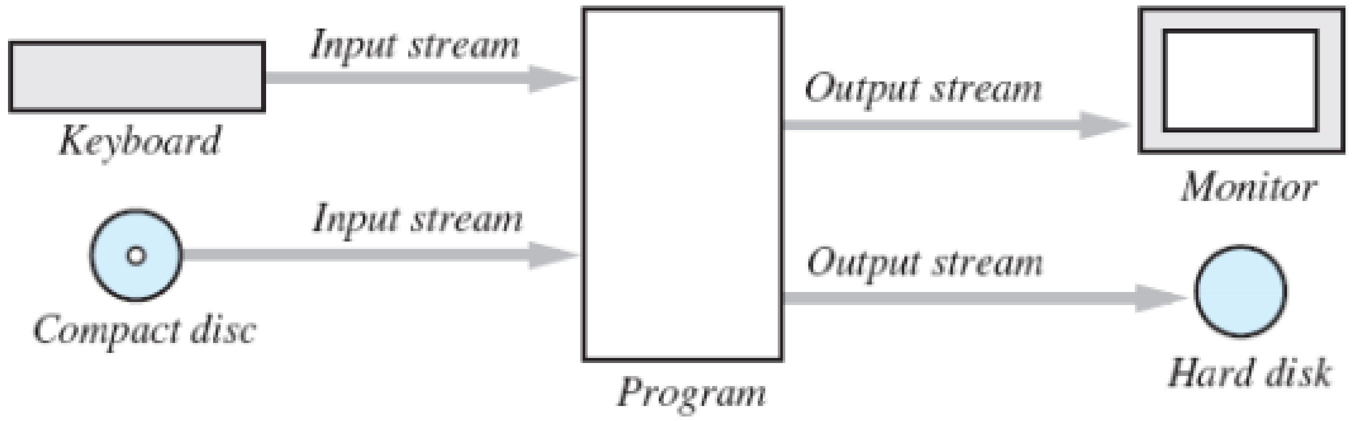
3.1.1 THE CONCEPT OF A STREAM

- **Use of files**
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O

- **A stream is a flow of input or output data**
  - Characters
  - Numbers
  - Bytes

THE CONCEPT OF A STREAM

- Streams are implemented as objects of special stream classes
  - Class Scanner
  - Object System.out

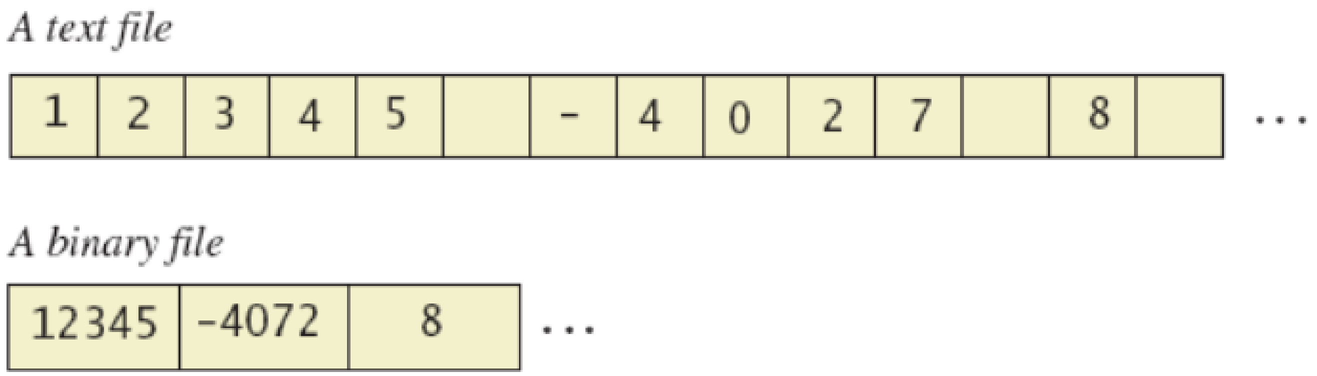


WHY USE FILES FOR I/O

- Keyboard input, screen output deal with
  - temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

TEXT FILES AND BINARY FILES

- All data in files stored as binary digits
  - Long series of zeros and ones
- Files treated as sequence of characters called text files
  - Java program source code
  - Can be viewed, edited with text editor
- All other files are called binary files
  - Movie, music files
  - Access requires specialized program



THE CLASS FILE

- Class provides a way to represent file names in a general way
  - A File object represents the name of a file
- The object
  - new File ("treasure.txt") is not simply a string
- It is an object that knows it is supposed to name a file

EXAMPLE

- Reading a file name from the keyboard
- See TextFileInputDemo.java

3.1.2 I/O STREAMS

- Byte Streams
- Character Streams
- Buffered Streams

BYTE STREAMS

- All byte stream classes are descended from InputStream and OutputStream.
- See ByteStreamProgram.java

CHARACTER STREAMS

- The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set.
- All character stream classes are descended from Reader and Writer.
- See CharStreamProgram.java
- **Line-Oriented I/O**
- See LineStreamProgram.java

BUFFERED STREAMS

- There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.
- See BufferedCopy.java

PATH

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- A relative path needs to be combined with another path in order to access a file.

3.2 PROGRAM ERRORS AND EXCEPTION HANDLING

3.2.1 Types of program errors

- Syntax errors: errors due to the fact that the syntax of the language is not respected.
- Semantic errors: errors due to an improper use of program statements.
- Logical errors: errors due to the fact that the specification is not respected.

When detect errors;

- Compile time errors: syntax errors and static semantic errors indicated by the compiler.
- Runtime errors: dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

SYNTAX ERRORS

Examples

```
int a = 5
x = ( 3 + 5;
y = 3 + * 5;
```

SEMANTIC ERRORS

Examples

- Use of a non-initialized variable: int i; i++;
- Type incompatibility: int a = "hello";
- Errors in expressions: String s = "..."; int a = 5 - s;
- Unknown references: Strin x; system.out.println("hello"); String s; s.println();
- Array index out of range (dynamic semantic error) int[] v = new int[10]; v[10] = 100;

LOGICAL ERRORS

Examples

- Non termination: String s = br.readLine(); while (s != null) { System.out.println(s); }

ERRORS DETECTED BY THE COMPILER AND RUNTIME ERRORS

Examples

- Division by zero:

```
int a, b, x;
a = 10;
b = Integer.parseInt(kb.readLine());
x = a / b; //ERROR if b = 0
```

- File does not exist:

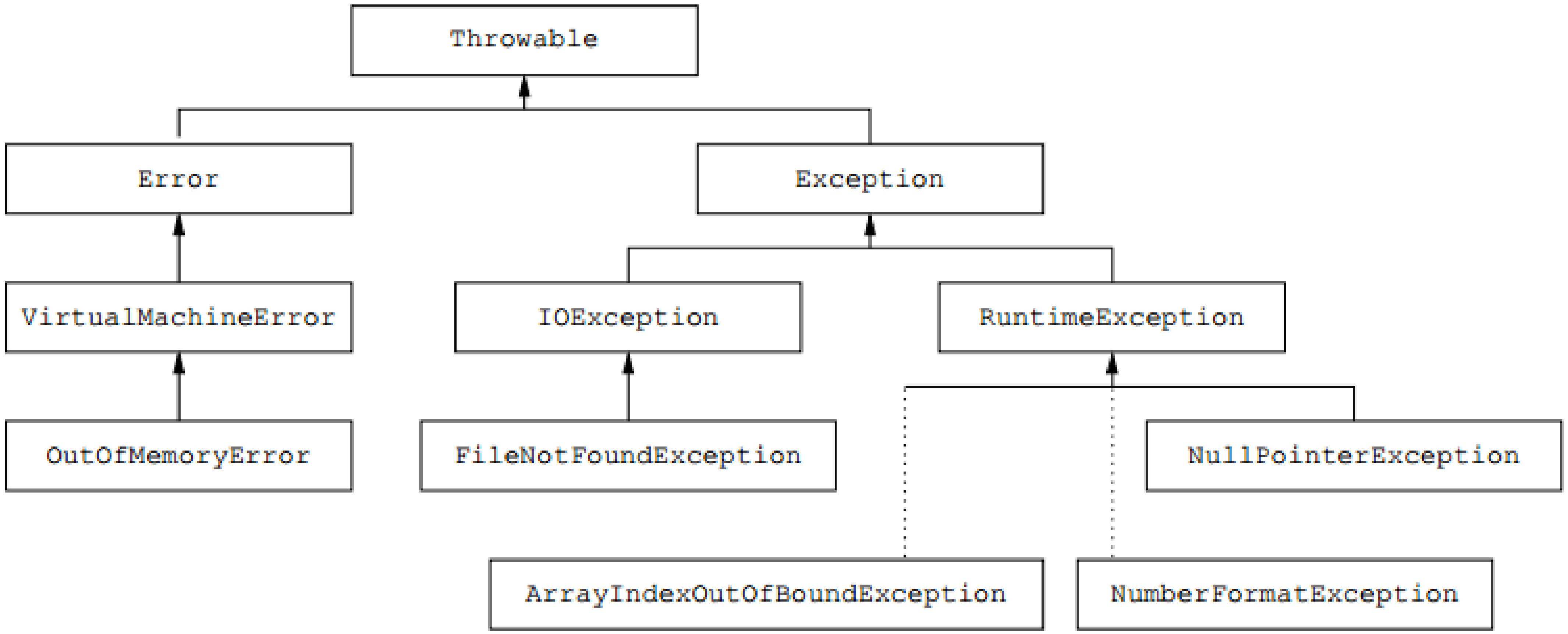
```
FileReader f = new FileReader("pippo.txt");
```

HANDLING ERRORS DURING THE EXECUTION OF PROGRAMS

```
public class TestException {
public static void main (String[] args) {
int falseNumber = Integer.parseInt("OK");
System.out.println("this println statement is not executed");
}
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "OK"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:468)
at java.lang.Integer.parseInt(Integer.java:518)
at TestException.main(TestException.java:3)
```

3.2.2 THE HIERARCHY OF EXCEPTIONS



HANDLE EXCEPTIONS

- There are two possibilities for handling exceptions:

  handling the exception where it is generated

  handling the exception in another point of the program

```
EXCEPTION IN THREAD "MAIN" JAVA.LANG.ARITHMETICEXCEPTION: / BY
ZERO AT DIVBYZERO.MAIN(DIVBYZERO.JAVA:7)
```

THE MOST COMMON TYPES OF EXCEPTIONS

- NullPointerException
- ArrayIndexOutOfBoundsException
- IOException
- FileNotFoundException
- NumberFormatException

3.2.3 THE THROWS CLAUSE

- All Java methods can use the throws clause to handle the exceptions generated by the methods they invoke. The throws clause is added to the header of the method definition.

For example:

```
public static void main(String[] args) throws IOException {
... }
```

THE THROW STATEMENT

```
import java.io.*;
public class UseMyException {
            public static void main(String[] args) throws MyException, IOException {
int min = 10, max = 30;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Input a number between " + min + " and " + max + " included");
int value = Integer.parseInt(br.readLine());
if (value < min || value > max)
throw new MyException("The value is not in the allowed interval"); System.out.println("The value is in the allowed interval");
}
}
```

3.2.4 CATCH AN EXCEPTION

```
try {
 try-block }
catch(ExceptionClass1 e) {
catch-block }
catch(ExceptionClass2 e) {
catch-block }
...
finally {
finally-block }
```

CATCH AN EXCEPTION

- try-block : sequence of statements that will be executed under the control of the following catch clauses
- catch-block : sequence of statements that will be executed if a statement in the try-block generates an exception of the type specified in the corresponding catch clause
- finally-block : sequence of statements that will be always executed (both in the case where the try-block is executed without exceptions, and in the case where a catch-block is executed to catch an exception).

CATCH AN EXCEPTION

```
try {
System.out.println(Integer.parseInt(br.readLine()); }
catch(IOException e1) {
System.out.println("An IO error occurred."); } catch(NumberFormatException e2) {
System.out.println("The string read does not contain an integer."); } finally {
System.out.println("Block executed."); }
```