# Java Programming – 18BCS43C

**Dr. S. Chitra,**
**Associate Professor,**
**Post Graduate & Research Department of Computer Science,**
**Government Arts College(Autonomous), Coimbatore - 641018**

| Year | Subject Title | Sem. | Sub Code |
|---|---|---|---|
| **2018 -19 Onwards** | JAVA PROGRAMMING | IV | 18BCS43C |

**UNIT - IV**

File Handling: Types of Errors – Exceptions- Syntax of Exception Handling Code-Multiple Catch Statements- Using Finally Statements- Managing Input / Output Files in Java: Concept of Streams-Stream Classes- Character Stream - Classes-Reading / Writing Characters- Reading / Writing Bytes-Handling Primitive Data Types- Random Access files.

## Exceptions

An exception is an event, which occurs during the execution of the program, that an interrupt the normal flow of the program's instruction. In other words, Exceptions are generated when a recognized condition, usually an error condition, arises during the execution of a method. Java includes a system for running exceptions, by tracking the potential for each method to throw specific exceptions. For each method that could throw an exception, your code must report to the Java compiler that it could throw that exact exception. The compiler marks that method as potentially throwing that exception, and then need any code calling the method to handle the possible exception. Exception handling is basically use five keyword as follows:

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

Exception can be generated by Java-runtime system or they can be manually generated by code. Error-Handling becomes a necessary while developing an application to account for exceptional situations that may occur during the program execution, such as

1. Run out of memory
2. Resource allocation Error
3. Inability to find a file
4. Problems in Network connectivity.

## Types of Errors

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

## Compile-Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of = = operator
- And so on

```
/* This program contains an error */
class Error1
{
        public static void main(String args[])
        {
            System.out.println("Hello Java!") //  Missing;
        }
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in Program .1, the following message will be displayed in the screen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^
1 error
```

## Run-Time Errors

Sometimes, a program may compile successfully creating the **.class** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And may more

When such errors are encountered, Java typically generates an error message and aborts the program.

## Illustration of run-time errors

```
class Error2
{
	public static void main(String args[ ])
	{
		int a = 10;
		int b = 5;
		int c = 5;
		int x = a/(b-c);        // Division by zero
		System.out.println("x = " + x);
		int y = a/(b+c);
		System.out.println("y = " + y);
	}
}
```

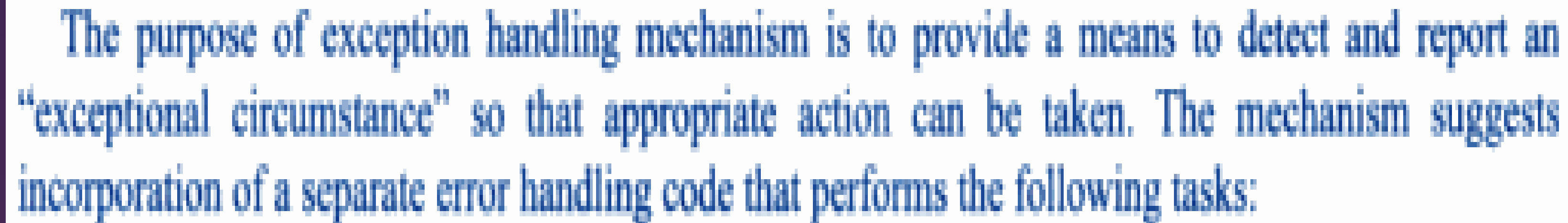
Program is syntactically correct and therefore does not cause any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

```
java.lang.ArithmeticException: / by zero
     at Error2.main(Error2.java:10)
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e. informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*.



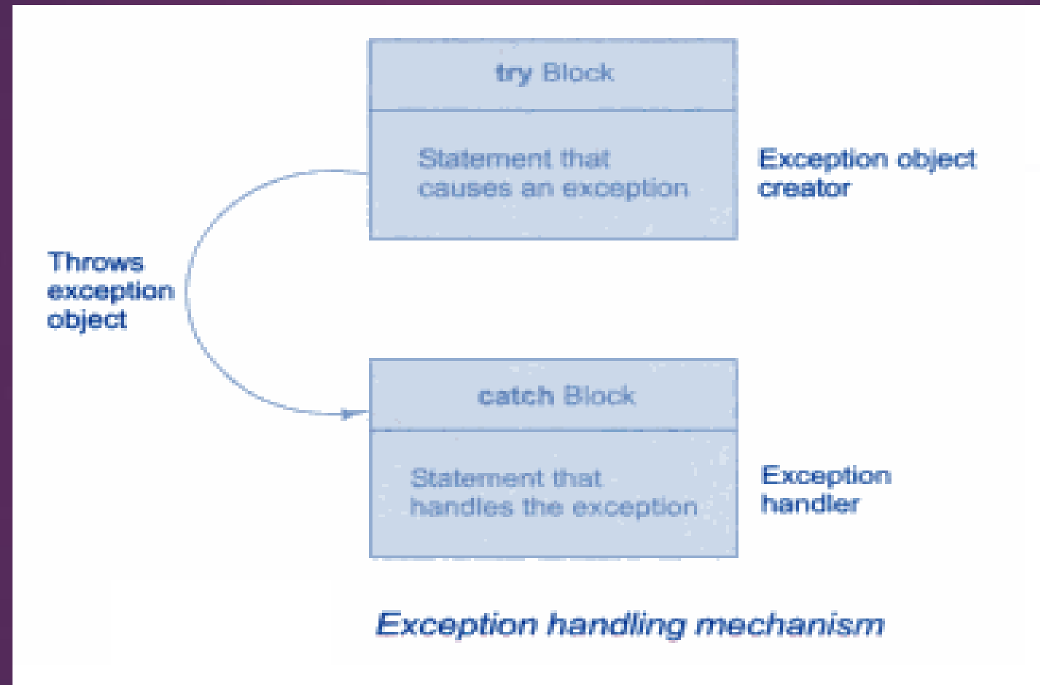
The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

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

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

| *Exception Type* | *Cause of Exception* |
|---|---|
| ArithmeticException | Caused by math errors such as division by zero |
| ArrayIndexOutOfBoundsException | Caused by bad array indexes |
| ArrayStoreException | Caused when a program tries to store the wrong type of data in an array |
| FileNotFoundException | Caused by an attempt to access a nonexistent file |
| IOException | Caused by general I/O failures, such as inability to read from a file |
| NullPointerException | Caused by referencing a null object |
| NumberFormatException | Caused when a conversion between strings and number fails |
| OutOfMemoryException | Caused when there's not enough memory to allocate a new object |
| SecurityException | Caused when an applet tries to perform an action not allowed by the browser's security setting |
| StackOverFlowException | Caused when the system runs out of stack space |
| StringIndexOutOfBoundsException | Caused when a program attempts to access a nonexistent character position in a string |

# Syntax of Exception Handling Code




*Exception handling mechanism*

Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by thee try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements:

```
...........
...........
try
{
    statement;    // generates an exception
}
catch (Exception-type e)
{
    statement;    // processes the exception
}
...........
..........
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

```
public class demo
{
public static void main(String[] args)
{
int ans1, ans2;
int a = 2, b = 2, c = 0;
try
{
ans1 = a/b;
System.out.println("a/b = " + ans1);
ans2 = a/c;
System.out.println("a/c = " + ans2);
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic
Exception!");
}
System.out.println("demo is over");
}
}
```

**Output:**

```
C:\>set path=C:\Java\jdk1.5.0_01\bin
C:\>javac demo.java
C:\>java demo
a/b = 1
Arithmetic Exception!
demo is over
```

**Using Multiple catch Blocks**

It is possible that a statement might throw more than one kind of exception

 you can list a sequence of catch blocks, one for each possible exception

 remember that there is an object hierarchy for exceptions –

```
class demo
{
public static void main (String args [])
{
int A[ ] = new int [5];
try
{
for (int c = 0; c <5; c++)
{
//do nothing
}
for (int c = 0; c <5; c++)
{
A[c] = c/ c;
}
}
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println ("Array out of bound ");
}
catch (ArithmeticException e)
{
System.out.println ("Zero divide error");
}
}
}
```

**Output:**

C:\>javac demo.java
C:\>java demo
Zero divide error
C:\>

**Finally Block**

To guarantee that a line of code runs, whether an exception occurs or not, use a finally block after the try and catch blocks The code in the finally block will *almost always* execute, even if an unhandled exception occurs; in fact, even if a return statement is encountered

1. if an exception causes a catch block to execute, the finally block will be executed after the catch block

2. if an uncaught exception occurs, the finally block executes, and then execution exits this method and the exception is thrown to the method that called this method

Syntax –

```
try
{
risky code/ unsafe code block
}
catch (ExceptionClassName exceptionObjectName)
{
code to resolve problem
}
finally
{
code that will always execute
}
```

**In summary:**

**1.** a try block is followed by zero or more catch blocks

2. There may one finally block as the last block in the structure.

3. There must be at least one block from the collective set of catch and finally after the try.

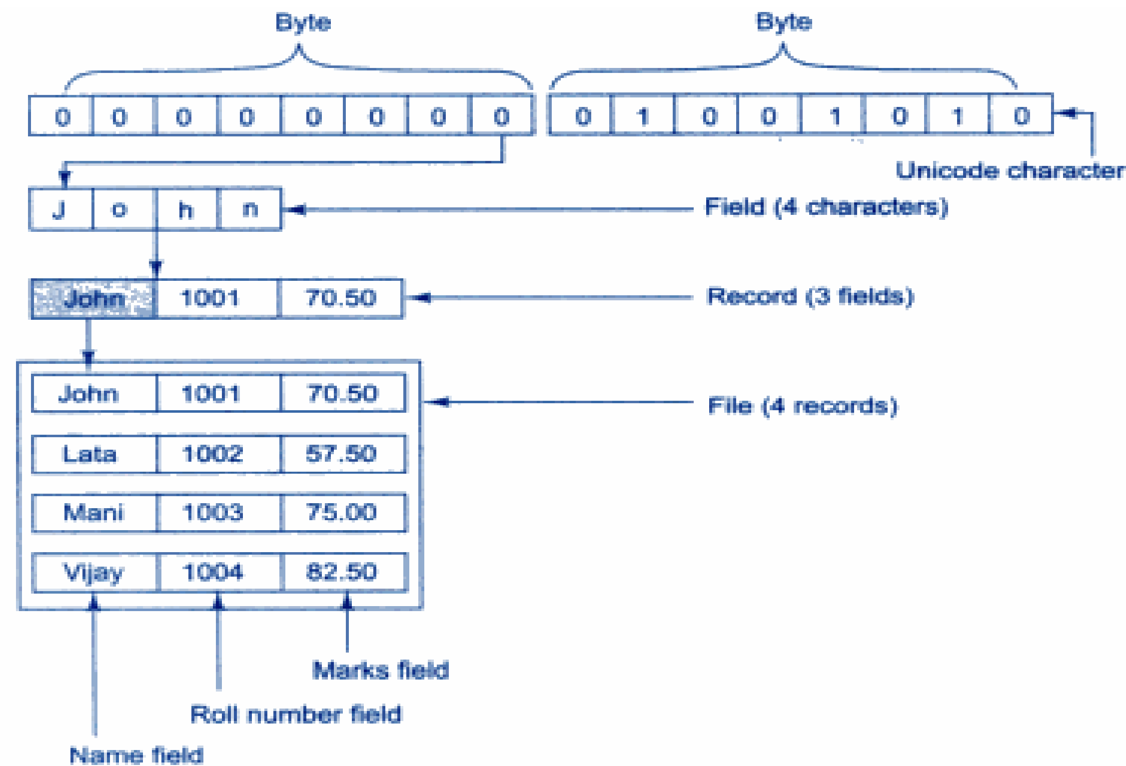
## Managing Input / Output Files in Java

Data can be stored using variables and their values are used during execution. There are 2 drawbacks in it. They are

1. The data is lost either when a variable goes out of scope or when the program is terminated. That is, the storage is temporary.
2. It is difficult to handle large volumes of data using variables and arrays.

We can overcome these problems by storing data on *secondary storage devices* such as floppy disks or hard disks. The data is stored in these devices using the concept of *files*.

A file is a collection of related *records* placed in a particular area on the disk. A record is composed of several fields and a field is a group of characters as illustrated in Fig. Characters in Java are *Unicode* characters composed of two *bytes*, each byte containing eight binary digits, 1 or 0.

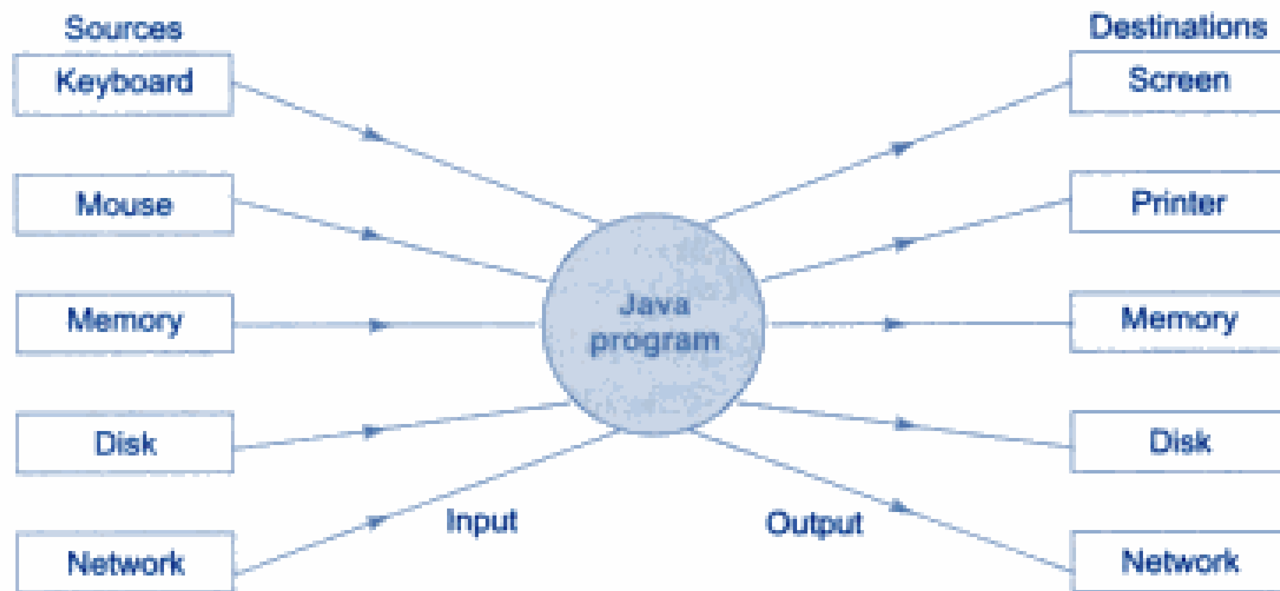



*Data representation in Java files*

Storing and managing data using files is known as *file processing* which includes tasks such as creating files, updating files and manipulation of data. Java supports many powerful features for managing input and output of data using files. Reading and writing of data in a file can be done at the level of bytes or characters or fields depending on the requirements of a particular application. Java also provides capabilities to read and write class objects directly. Note that a record may be represented as a class object in Java. The process of reading and writing objects is called *object serialization*.

## Concept of Streams

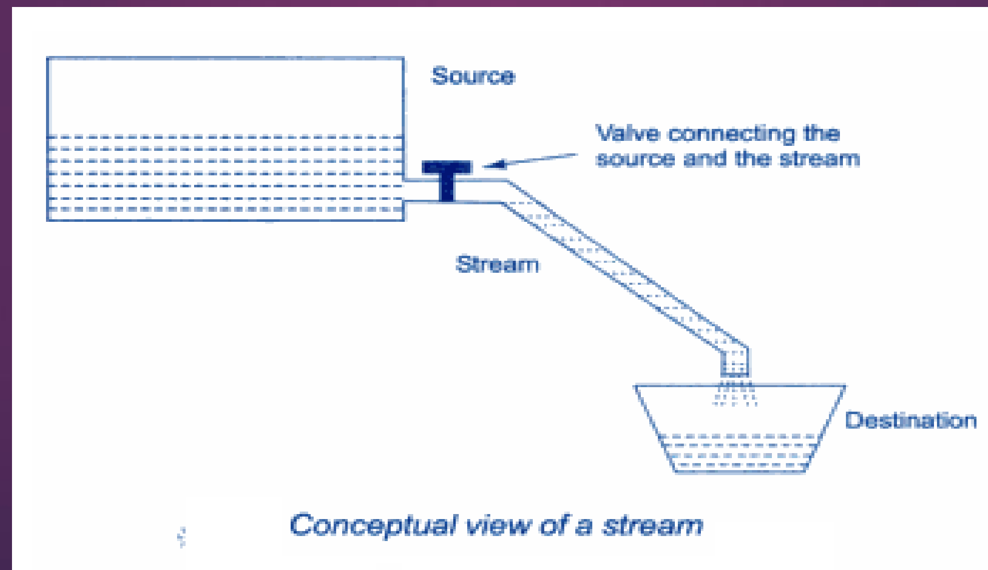
In file processing, input refers to the flow of data into a program and output means the flow of data out of a program. Input to a program may come from the keyboard, the mouse, the memory, the disk, a network, or another program. Similarly, output from a program may go to the screen, the printer, the memory, the disk, a network, or another program.



Relationship of Java program with I/O devices

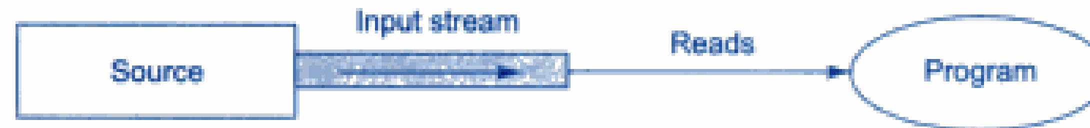
Java uses the concept of streams to represent the ordered sequence of data, a common characteristic shared by all the input/output devices as stated above. A stream presents a uniform, easy-to-use, object-oriented interface between the program and the input/output devices.

A stream in Java is a path along which data flows (like a river or a pipe along which water flows). It has a *source* (of data) and a *destination* (for that data) as depicted in Fig. Both the source and the destination may be physical devices or programs or other streams in the same program.

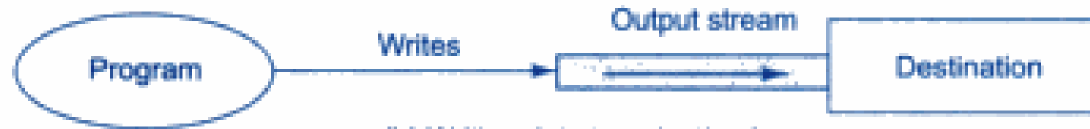



*Conceptual view of a stream*

The concept of sending data from one stream to another (like one pipe feeding into another pipe) has made streams in Java a powerful tool for file processing. We can build a complex file processing sequence using a series of simple stream operations. This feature can be used to filter data along the pipeline of streams so that we obtain data in a desired format. For example, we can use one stream to get raw data in binary format and then use another stream in series to convert it to integers.

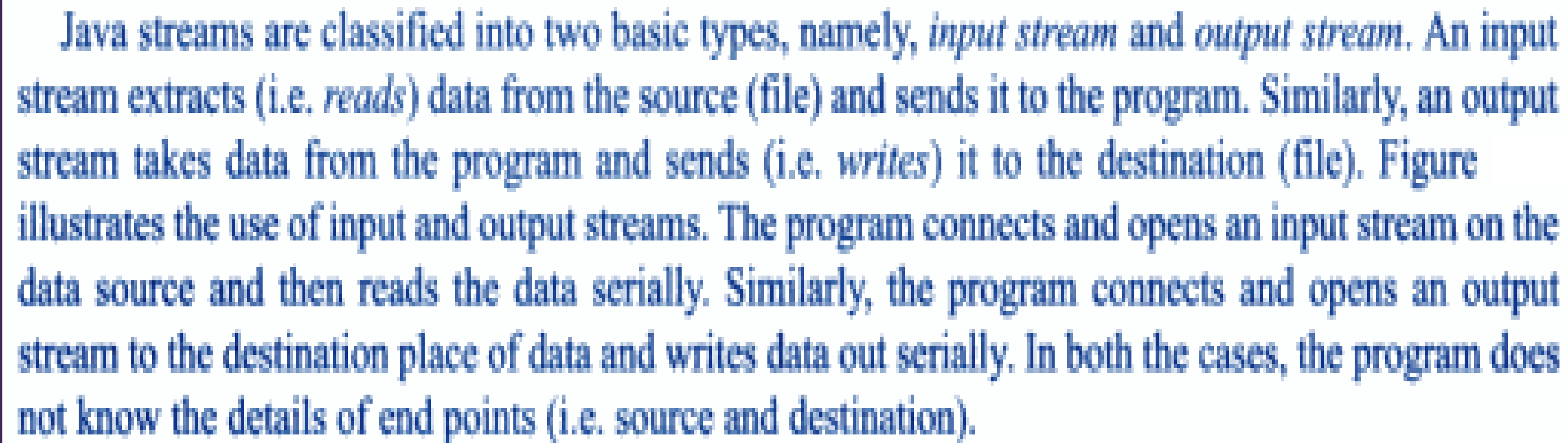



(a) Reading data into a program




(b) Writing data to a destination

*Using input and output streams*

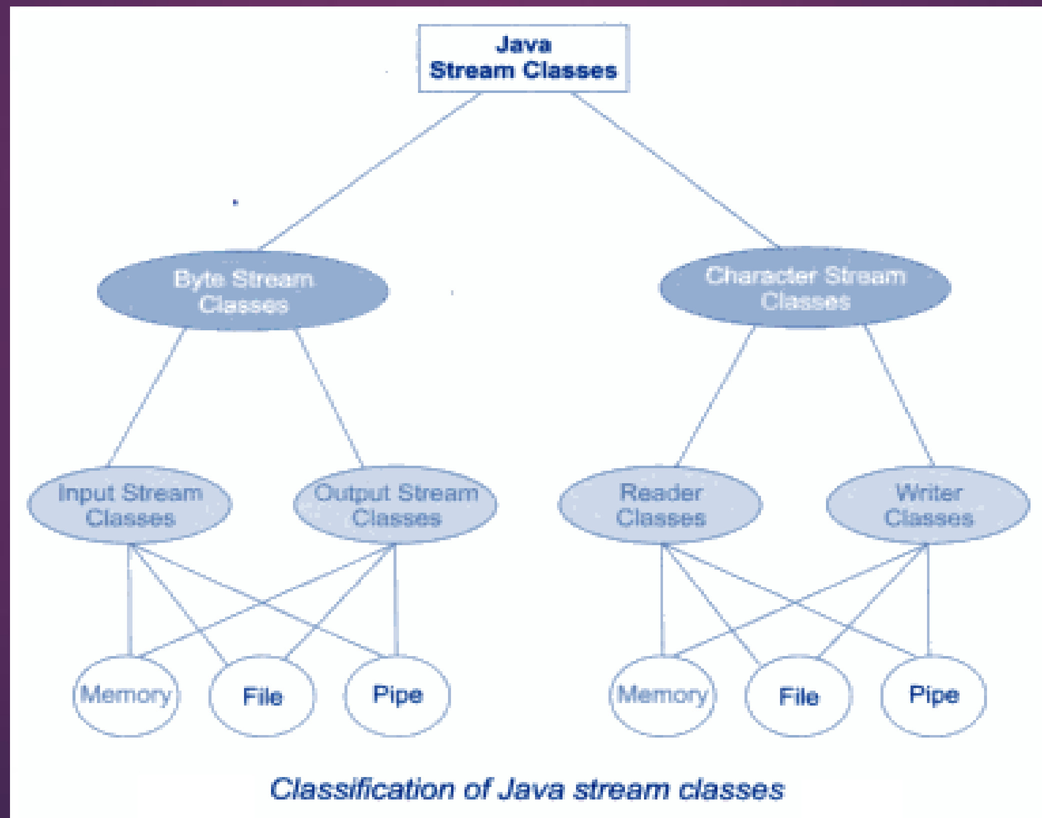


Java streams are classified into two basic types, namely, *input stream* and *output stream*. An input stream extracts (i.e. *reads*) data from the source (file) and sends it to the program. Similarly, an output stream takes data from the program and sends (i.e. *writes*) it to the destination (file). Figure illustrates the use of input and output streams. The program connects and opens an input stream on the data source and then reads the data serially. Similarly, the program connects and opens an output stream to the destination place of data and writes data out serially. In both the cases, the program does not know the details of end points (i.e. source and destination).

## Stream Classes

The **java.io** package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.




*Classification of Java stream classes*

## Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and, therefore, Java provides two kinds of byte stream classes: *input stream classes* and *output stream* classes.

## Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as **InputStream** and a number of subclasses for supporting various input-related functions.

The super class **InputStream** is an abstract class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class. The **InputStream** class defines methods for performing input functions such as

- Reading bytes
- Closing streams
- Marking positions in streams
- Skipping ahead in a stream
- Finding the number of bytes in a stream

*Hierarchy of input stream classes*

InputStream Methods

| | Method | Description |
|---|---|---|
| 1. | read( ) | Reads a byte from the input stream |
| 2. | read (byte b[ ]) | Reads an array of bytes into b |
| 3. | read (byte b[ ], int n, int m) | Reads m bytes into b starting from nth byte. |
| 4. | available( ) | Gives number of bytes available in the input |
| 5. | skip(n) | Skips over n bytes from the input stream |
| 6. | reset( ) | Goes back to the beginning of the stream |
| 7. | close( ) | Closes the input stream |

Note that the class **DataInputStream** extends **FilterInputStream** and implements the interface **DataInput.** Therefore, the **DataInputStream** class implements the methods described in **DataInput** in addition to using the methods of **InputStream** class. The **DataInput** interface contains the following methods:

- readShort( )
- readInt( )
- readLong( )
- readFloat( )
- readUTF( )
- readDouble( )
- readLine( )
- readChar( )
- readBollean( )

## Output Stream Classes

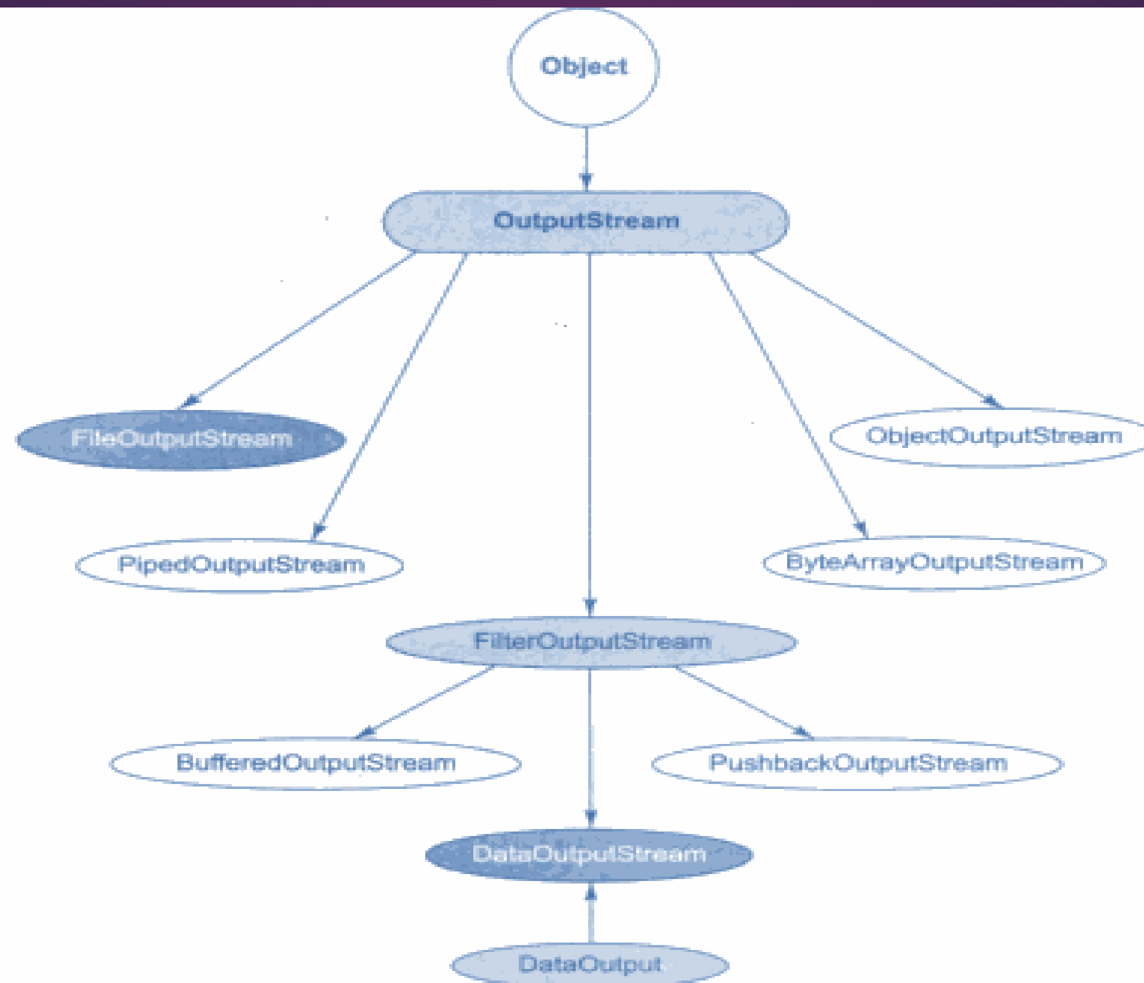
Output stream classes are derived from the base class **OutputStream** as shown in Fig. Like **InputStream,** the **OutputStream** is an abstract class and therefore we cannot instantiate it. The several subclasses of the **OutputStream** can be used for performing the output operations.

The **OutputStream** includes methods that are designed to perform the following tasks:

- Writing bytes
- Closing streams
- Flushing streams

### OutputStream Methods

| *Method* | *Description* |
|---|---|
| 1. write( ) | Writes a byte to the output stream |
| 2. write(byte[ ] b) | Writes all bytes in the array b to the output stream |
| 3. write(byte b[ ], int n, int m) | Writes m bytes from array b starting from nth byte |
| 4. close( ) | Closes the output stream |
| 5. flush( ) | Flushes the output stream |




*Hierarchy of output stream classes*

**Character Streams:**

It supports 16-bit Unicode character input and output. There are two classes of character stream as follows:

1. Reader
2. Writer

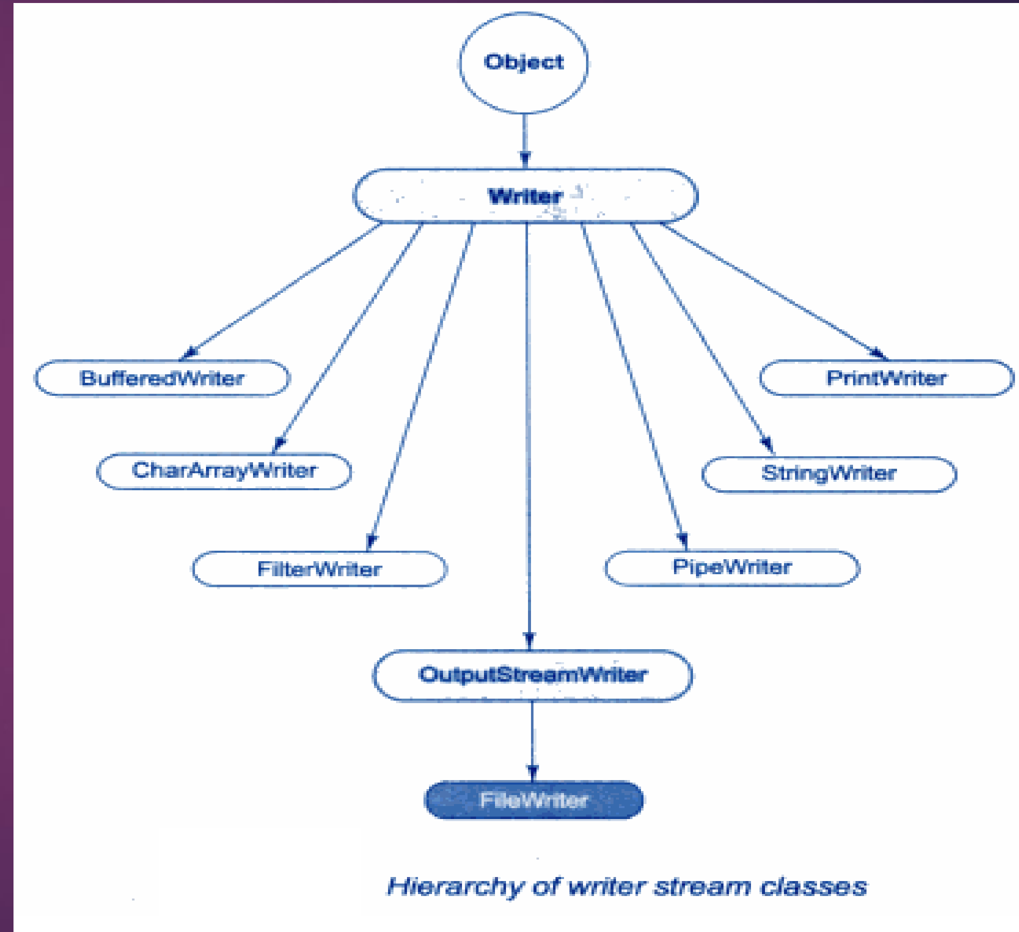
These classes allow internationalization of Java I/O and also allow text to be stored using international character encoding.

**Reader:**

- BufferedReader
    - LineNumberReader
- CharAraayReader
- PipedReader
- StringReader
- FilterReader
    - PushbackReader
- InputStreamReader
    - FileReader

**Writer:**

- BufferedWriter
- CharAraayWriter
- FileWriter
- PipedWriter
- PrintWriter
- String Writer
- OutputStreamWriter
    - FileWriter



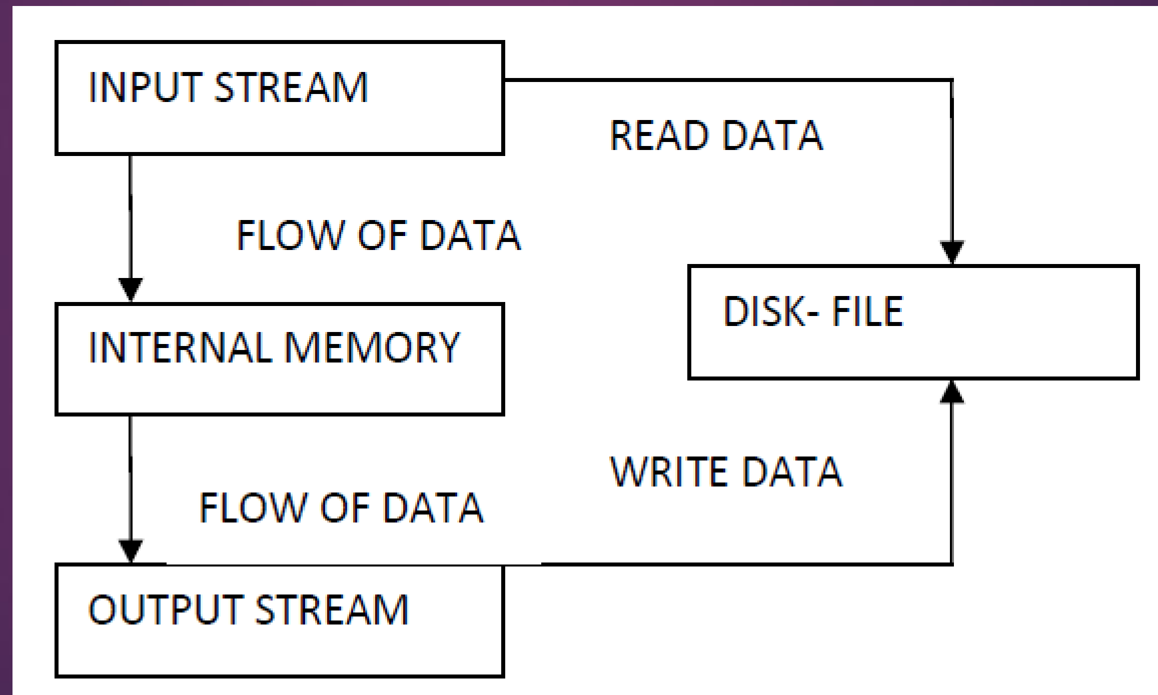
Hierarchy of writer stream classes

## HOW FILES AND STREAMS WORK:

Java uses **streams** to handle I/O operations through which the data will flow from one location to another. For example, an **InputStream** can flow the data from a disk file to the internal memory and an **OutputStream** can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file.

When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream.

The working process of the I/O streams can be shown in the given diagram.



## Using the File Class

The **java.io** package includes a class known as the **File** class that provides support for creating files and directories. The class includes several constructors for instantiating the **File** objects. This class also contains several methods for supporting the operations such as

- Creating a file
- Opening a file
- Closing a file
- Deleting a file
- Getting the name of a file
- Getting the size of a file
- Checking the existence of a file
- Renaming a file
- Checking whether the file is writable
- Checking whether the file is readable

## Input/Output Exceptions

When creating files and performing i/o operations on them, the system may generate i/o related exceptions. The basic i/o related exception classes and their functions are given in Table

| *I/O Exception class* | *Function* |
|---|---|
| EOFException | Signals that an end of the file or end of stream has been reached unexpectedly during input |
| FileNotFoundException | Informs that a file could not be found |
| InterruptedIOException | Warns that an I/O operations has been interrupted |
| IOException | Signals that an I/O exception of some sort has occurred |

Each i/o statement or group of i/o statements must have an exception handler around it as shown below or the method must declare that it throws an IOException.

```
try
{
..........
..........  //  I/O statements
..........
}
catch (IOException e)
{
..........    //  Message output statement
}
```

Proper use of exception handlers would help us identify and locate i/o errors more effectively.

## Creation of Files

If we want to create and use a disk file, we need to decide the following about the file and its intended purpose:

- Suitable name for the file
- Data type to be stored
- Purpose (reading, writing, or updating)
- Method of creating the file

A filename is a unique string of characters that helps identify a file on the disk. The length of a filename and the characters allowed are dependent on the OS on which the Java program is executed. A filename may contain two parts, a primary name and an optional period with extension. Examples:

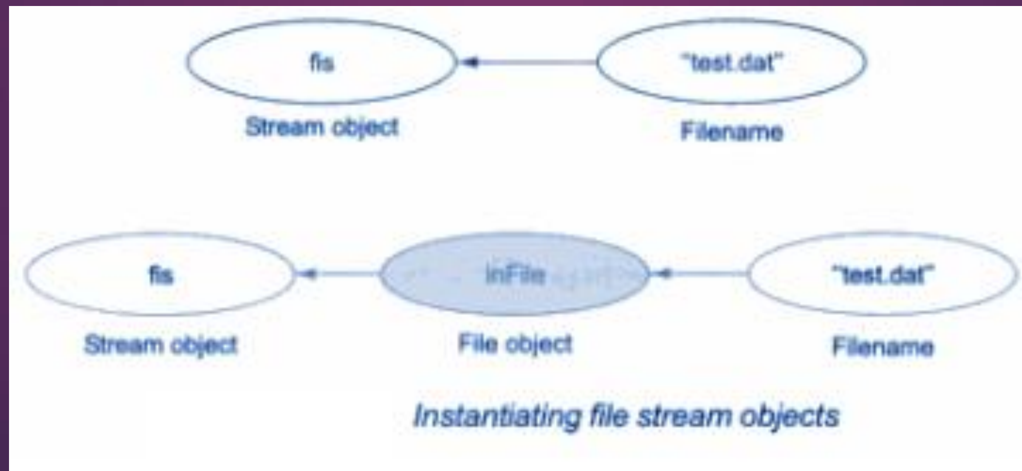
```
input.data      salary
test.doc        student.txt
inventory       rand.dat
```

Data type is important to decide the type of file stream classes to be used for handling the data. We should decide whether the data to be handled is in the form of characters, bytes or primitive type.

The purpose of using a file must also be decided before using it. For example, we should know whether the file is created for reading only, or writing only, or both the operations.

# Steps in Using Files

- Select a filename
- Declare a file object
- Give the selected name to the file object declared
- Declare a file stream object
- Connect the file to the file stream object




*Instantiating file stream objects*

## Reading/Writing Characters

subclasses of **Reader** and **Writer** implement streams that can handle characters. The two subclasses used for handling characters in files are **FileReader** (for reading characters) and **FileWriter** (for writing characters). Program uses these two file stream classes to copy the contents of a file named "input.dat" into a file called "output.dat".

**Program for *Copying characters***

```
// Copying characters from one file into another
import java.io.*;
class CopyCharacters
{   public static void main (String args [ ] )
    {
        // Declare and create input and output files
        File inFile = new File ("input.dat") ;
        File outFile  = new File ("output.dat") ;
        FileReader ins = null;             // Creates file stream ins
        FileWriter outs = null;            // Creates file stream outs
        try
        {
            ins = new FileReader (inFile) ;     // Opens inFile
            outs = new FileWriter (outFile) ;   // Opens outFile
            // Read and write till the end
            int ch;
            while ( (ch = ins.read( ) ) != – 1)
            {
                outs.write (ch) ;
            }
        }
        catch (IOException e)
        {
            System.out.println (e) ;
            System.exit (– 1) ;
        }
        finally   // Close files
        {
            try
            {
                ins.close ( ) ;
                outs.close ( ) ;
            }
            catch (IOException e) { }
        }
    }
}
```

This program is very simple. It creates two file objects **inFile** and **outFile** and initializes them with "input.dat" and "output.dat" respectively using the following code:

```
File inFile   = new File ("input.dat");
File outFile  = new File ("output.dat");
```

The program then creates two file stream objects ins and outs and initializes them with "null" as follows:

```
FileReader ins  = null;
FileWriter outs = null;
```

These streams are then connected to the named files using the following code:

```
ins  = new FileReader (inFile) ;
outs = new FileWriter (outFile) ;
```

This connects **inFile** to the **FileReader** stream **ins** and **outFile** to the **FileWriter** stream **outs**. This essentially means that the files "input.dat" and "output.dat" are opened. The statements

```
ch = ins.read ( )
```

reads a character from the **inFile** through the input stream **ins** and assigns it to the variable **ch.** Similarly, the statement

```
outs.write (ch) ;
```

writes the character stored in the variable **ch** to the **outFile** through the output stream **outs.** The character –1 indicates the end of the file and therefore the code

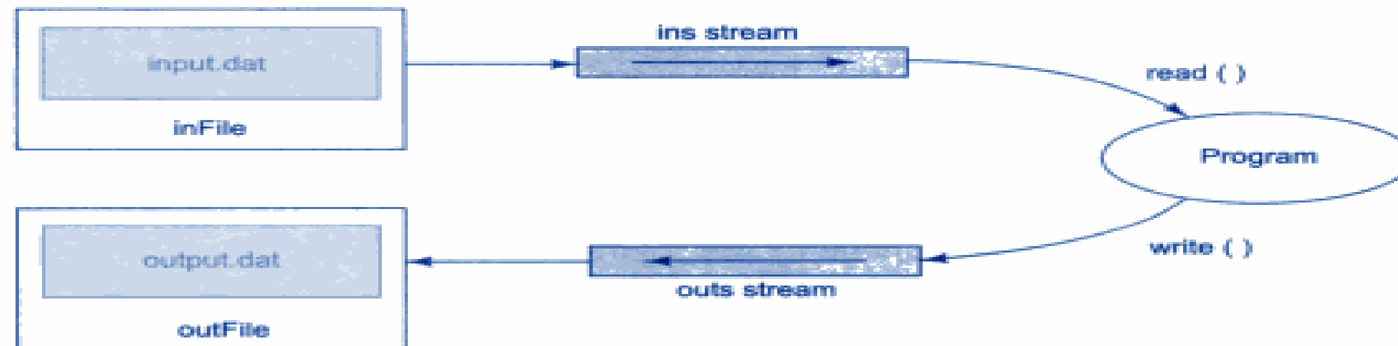
```
while ( (ch=ins.read( ) ) != -1)
```

causes the termination of the while loop when the end of the file is reached. The statements

```
ins.close ( ) ;
outs.close ( ) ;
```

enclosed in the **finally** clause close the files created for reading and writing. When the program catches an I/O exception, it prints a message and then exits from execution.

The concept of using file streams and file objects for reading and writing characters in Program is illustrated in Fig.




*Reading from and writing to files*

## Reading/Writing Bytes

Two commonly used classes for handling bytes are **FileInputStream** and **FileOutputStream** classes. Program demonstrates that **FileOutputStream** class is used for writing bytes to a file. The program writes the names of some cities stored in a byte array to a new file named "city.txt". We can verify the contents of the life by using the command

```
type city.txt
```

```
import java.io.*;
class WriteBytes
{
    public static void main (String args [ ] )
    {
        //  Declare and initialize a byte array
        byte cities [ ] = {'D', 'E', 'L', 'H', 'I', '\n', 'M',
            'R', 'A', 'S', '\n', 'L', 'O', 'N', 'D', 'O', 'N',
        //  Create an output file stream
        FileOutputStream outfile = null;
        try
        {
            //  Connect the outfile stream to "city.txt"
            outfile = new FileOutputStream ("city.txt") ;
            //  Write data to the stream
            outfile.write (cities) ;
            outfile.close ( ) ;
        }
        catch (IOException ioe)
        {
            System.out.println (ioe) ;
            System.exit (-1) ;
        }
    }
}
```

```
type city.txt
DELHI
    MADRAS
        LONDON
```

```java
// Reading bytes from a file
import java.io.*;
class ReadBytes
{
    public static void main (String args [ ] )
    {
        // Create an input file stream
        FileInputStream infile = null;
        int b;
        try
        {
            // Connect infile stream to the required file
            infile = new FileInputStream (args [ 0 ] )
            // Read and display data
            while ( (b = infile.read ( ) ) != -1)
            {
                System.out.print ( ( char) b) ;
            }
            infile.close ( ) ;
        }
        catch (IOException ioe)
        {
            System.out.println (ioe) ;
        }
    }
}
```

```
Prompt>java ReadBytes city.txt
    DELHI
        MADRAS
            LONDON
```

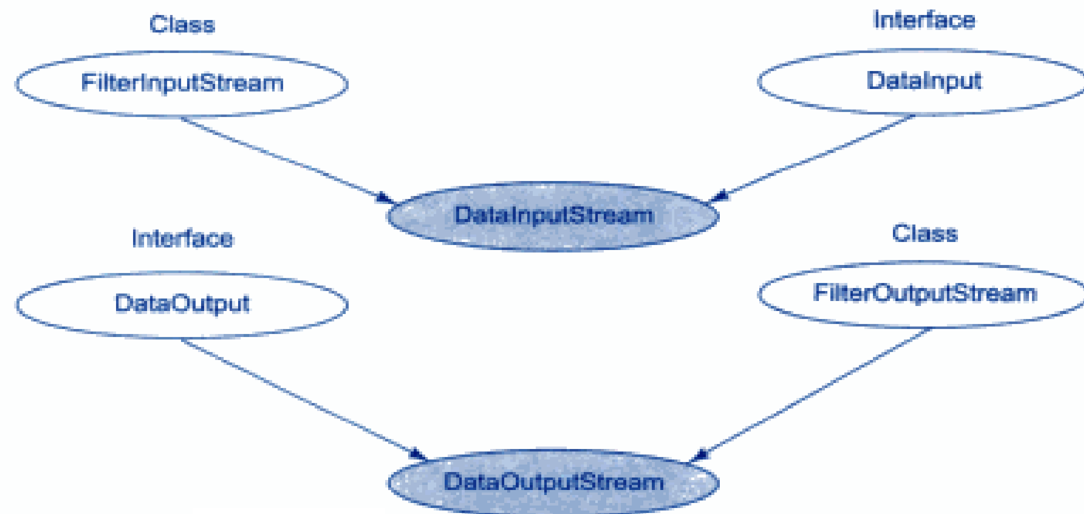
## Handling Primitive Data Types

The two filter classes used for creating "data streams" for handling primitive types are **DataInputStream** and **DataOutputStream**.

A data stream for input can be created as follows:

```
FileInputStream fis = new FileInputStream (infile) ;
DataInputStream dis = new DataInputStream (fis) ;
```

These statements first create the input file stream fis and then create the input data stream **dis**. These statements basically wrap **dis** on **fis** and use it as a "filter". Similarly, the following statements create the output data stream **dos** and wrap it over the output file stream **fos**.




*Hierarchy of data stream classes*

```
FileOutputStream fos = new FileOutputStream (outfile) ;
DataOutputStream dos = new DataOutputStream (fos) ;
```

```
import java.io.*;
class ReadWriteIntegers
{
  public static void main(String args [ ])
  {
    DataInputStream dis = null;
    DataoutputStream dos = null;
    File intFile = new File("rand.dat");
    try
    {
      dos = new DataOutputStream(new
       FileOutputStream (intFile) ) ;
                for (int i = 0; i<20;i++)
                        dos.writeInt ( (int) (math.random ( ) *100) ) ;
        }
        catch (IOException ioe)
        {
                System.out.println (ioe.getMessage ( ) ) ;
        }
        finally
        {
                try
                {
                        dos.close ( ) ;
                }
                catch (IOException ioe) { }
        }
        // Reading integers from rand.dat file
        try
        {
                // Create input stream for intFile file
                dis = new DataInputStream (new
                        FileInputStream (intFile) ) ;
                for (int i=0; i < 20; i++)
                {
                        int n = dis.readInt ( ) ;
                        System.out.print (n + " ") ;  }
        }
        catch (IOException ioe)
        {
                System.out.println (ioe.getMessage ( ) ) ; }
        finally
        {
                try
                {
                        dis.close ( ) ;
                }
                catch (IOException ioe) { }
        }
    }
  }
```

**Output of Program**

```
78 62 54 56 55 48 48 35 13 64 13 90 10 78 91 42 9 44 84 66
```

Java - RandomAccessFile

This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

# Constructors for RandomAccessFile class

| Constructor | Description |
| --- | --- |
| RandomAccessFile(File file, String mode) | Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. |
| RandomAccessFile(String name, String mode) | Creates a random access file stream to read from, and optionally to write to, a file with the specified name. |

| Modifier and Type | Method | Method |
|---|---|---|
| **void** | close() | It closes this random access file stream and releases any system resources associated with the stream. |
| **FileChannel** | getChannel() | It returns the unique FileChannel object associated with this file. |
| **int** | readInt() | It reads a signed 32-bit integer from this file. |
| **String** | readUTF() | It reads in a string from this file. |
| **void** | seek(long pos) | It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. |
| **void** | writeDouble(double v) | It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first. |
| **void** | writeFloat(float v) | It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first. |
| **void** | write(int b) | It writes the specified byte to this file. |
| **int** | read() | It reads a byte of data from this file. |
| **long** | length() | It returns the length of this file. |
| **void** | seek(long pos) | It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. |

```java
import java.io.IOException;
import java.io.RandomAccessFile;
 public class RandomAccessFileExample {
    static final String FILEPATH ="myFile.TXT";
    public static void main(String[] args) {
        try {
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I love my country and my people", 31);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static byte[] readFromFile(String filePath, int position, int size)
            throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
        return bytes;
    }
    private static void writeToFile(String filePath, String data, int position)
            throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "rw");
        file.seek(position);
        file.write(data.getBytes());
        file.close();
    }
}
```

The myFile.TXT contains text "This class is used for reading and writing to random access file.“ after running the program it will contains

References:

1. “Programming With JAVA – A Primer” , E. Balagurusamy, Fourth Edition, TMH Publications

2. “Java Programming” Internet Content – Anonomous Author

3. “The complete reference JAVA2”, Hervert schildt. TMH Publications.