

Chapter 12

The Java I/O System

Learning Objectives

After completing this chapter, you will be able to understand:

- *The Stream Class.*
- *The Byte Stream Classes.*
- *The Character Stream Classes.*
- *The Reader Stream Classes.*
- *The Writer Stream Classes.*
- *The File Class.*
- *The Random Access Files.*

INTRODUCTION

In this chapter, you will learn about the Input/Output(I/O) files in Java. You will also learn about I/O operation in Java and the classes used for file handling. The chapter discusses in detail about the serialization process that helps an object to write data into the streams and read it back again. It also covers some file system operations, including random access files. Most of the classes covered in this chapter are from the **java.io** package or the **java.util** package. These packages are home to most of the classes that you will use in this chapter for performing input and output operations.

STREAM CLASSES

In Java terminology, **Stream** means flow of data. A stream should always have a source and a destination. A stream is a sequence of information (bytes) of undetermined length that either brings in or takes out information to/from a source/destination. The Java stream classes either move information from an external source such as text file to Java or from Java to the external source/destination.

Most of the stream classes are part of the **java.io** package. There are various I/O Stream classes that can be used for handling I/O operation in Java. These classes can be grouped into two categories, as given below:

1. The Byte Stream classes
2. The Character Stream classes

The Byte Stream Classes

The classes Byte Stream perform I/O operations on bytes. You can use the classes Byte Stream for reading and writing bytes in streams and files. One of the limitations of these classes is that they can transmit data only in one direction. This means that flow of data using these classes are uni-directional. The classes Byte Stream contain the following two classes:

1. The InputStream classes
2. The OutputStream classes

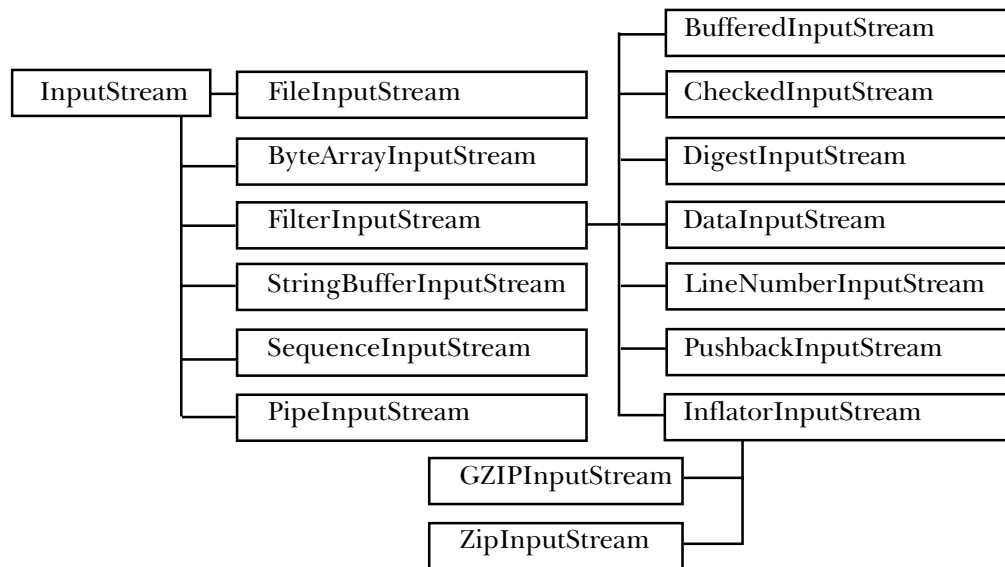
The InputStream Classes

The classes **InputStream** are used to read bytes from the stream. Also, these classes are used to read bytes or an array of bytes from an input source. The input source can be a file, a string, or a location that contains data to read. When you create an input stream, it is automatically opened. After reading, you do not need to close the input stream explicitly because if the object finds nothing in the data source, it closes implicitly. However, you can close an input stream explicitly by calling the **close()** method of the class **InputStream**.

The class **InputStream** can read the following sources:

1. An array of bytes
2. A file
3. A pipe

The class **InputStream** supports a number of subclasses for various input related functions. Figure 12-1 shows various subclasses of the **InputStream** class.



*Figure 12-1 Subclasses of the **InputStream** class*

The classes inherited from the class **InputStream** provided by the **java.io** package are used for various purposes, refer to Figure 12-1.

The class **InputStream** defines some methods that can be used for reading bytes or arrays of bytes, skipping bytes of input, resetting the current position of bytes within the stream, marking locations in the stream, and finding out the number of bytes available for reading. These methods are discussed next.

The available Method

This method returns the number of bytes that can be read or skipped from the input stream without blocking the next call of the method for input stream.

The close Method

This method is used to close the input stream and release the resources associated with it.

The mark Method

This method is used to mark the current position of the input stream. The syntax for using the **mark** method is as follows:

```
mark(int read_limit)
```

In the above syntax, the parameter **read_limit** is an integer value. It indicates the maximum number of bytes that can be read before the mark position becomes invalidate.

The **markSupported** Method

This method is used to test whether the input stream supports the **mark** and **reset** methods of the class **InputStream**. This method returns **True** when the stream instance supports the **mark** and **reset** methods. Otherwise, it returns **False**.

The **read()** Method

This method is used to read bytes next to the input stream. It returns the number of bytes return as an integer value ranging from 0 to 255. It returns -1, when no byte is available to read or the stream reaches at the end of the file.

The **read(byte[] byt)** Method

This method reads some of the bytes from the input stream and stores these bytes into the **byt** buffer array. It returns the integer value that is equivalent to the number of bytes it reads. If **byt** is null, a **NullPointerException** is thrown. It returns zero, when the length of **byt** is zero.

The **read(byte[] byt, int start, int len)** Method

This method reads the **len** bytes of the data from input stream into the **byt** buffer array. It returns the integer value that is equivalent to the number of bytes it reads. If **byt** is null, a **NullPointerException** is thrown. This method returns zero, if the length of **byt** is zero. If **start** or **len** is negative, or if the sum of the values of **start** and **len** is greater than the length of the buffer array **byt**, it will throw **IndexOutOfBoundsException**.

The **reset** Method

This method is used to reposition the pointer input stream to the position where the **mark** method was called last.

The **skip** Method

This method is used to skip over or discard the bytes of data from the input stream. The **skip** method returns the actual number of bytes skipped.

The syntax for using the **skip** method is as follows:

```
skip(long n)
```

In the above syntax, **n** is the number of bytes to be skipped. If **n** is negative, no bytes will be skipped.

The **OutputStream** Classes

The **OutputStream** classes are used to write bytes in the memory location. This class is used to write a byte or an array of bytes to an output source. The output sources can be a file, a string, or any memory location that contains data. When you create an output stream, it is

automatically opened. After writing, you do not need to close the output stream explicitly because if the object finds nothing in the data source, it closes implicitly. However, you can close it explicitly by calling the **close()** method of the class **OutputStream**.

The class **OutputStream** can write bytes in three different sources, which are as follows:

1. An array of bytes
2. A file
3. A pipe

The class **OutputStream** supports a number of subclasses for various output related functions, which are shown in Figure 12-2.

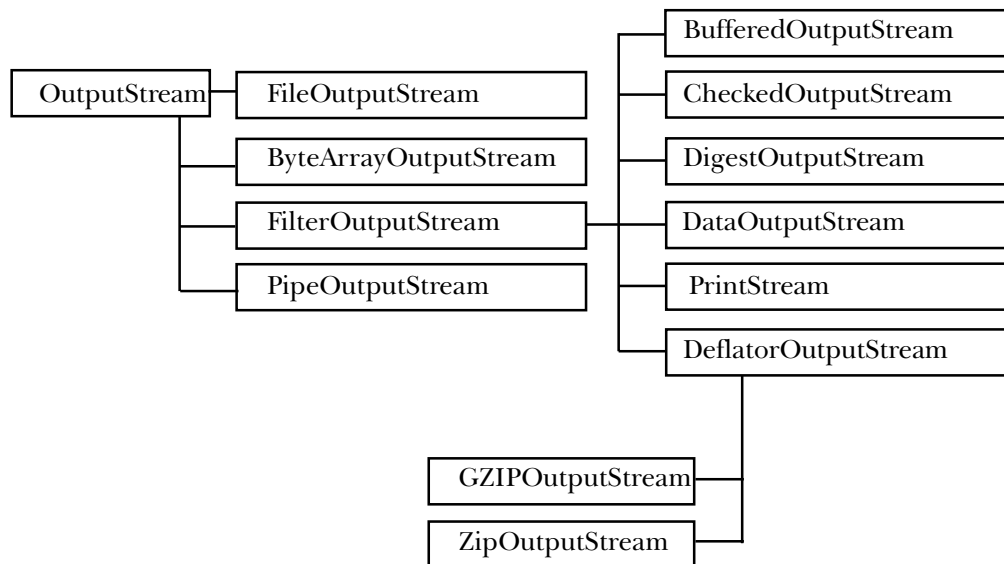


Figure 12-2 Subclasses of the **OutputStream** class

The classes inherited from the class **OutputStream** provided by the **java.io** package are used for different purposes.

The class **OutputStream** defines the methods of writing bytes or arrays of bytes to the stream and flushing out the stream. An output stream is automatically opened when you create it. You can explicitly close an output stream using the **close()** method or let it be closed implicitly when the object finds nothing in the data source. The methods provided by the class **OutputStream** are discussed next.

The **write(int b)** Method

This method writes the specified byte to the output stream. Generally, the functioning of this method is to write one (1) to the output stream.

The write(byte[] b) Method

This method writes the **b** bytes from the specified location or the byte array to the output stream. Here, **b** is the buffer array, which stores data to write.

The write(byte[] byt, int start, int len) Method

In the above method, **byt** is the data in bytes, which you want to write into the source stream. The **start** is the integer value that indicates the position in the buffer array **byt**, where you want to write data, and **len** is the number of bytes to write.

The **write** method of the class **OutputStream** writes the **len** bytes from the specified byte array, starting from the **start** position of the output stream. The general functioning of **write(byt, start, len)** is that some of the bytes in the array **byt** are written to the output stream. The order of writing bytes in the output stream is that the first byte is written as **byt[start]**, second as **byt[start+1]**, and this process continues till the last byte, which is written as **byt[start + len - 1]**.

If **byt** is null, the exception **NullPointerException** is thrown. The method **read** returns zero, when the length of **byt** is zero. If **start** or **len** is negative or both **start** and **len** are greater than the length of the buffer array **byt**, it will throw the exception **IndexOutOfBoundsException**.

The flush Method

The **flush** method forces the buffered bytes to be written to the output stream. Calling the **flush** method indicates that the previously written bytes have been buffered by the output stream and such bytes must be written immediately to their destination.

The close Method

This method of the class **OutputStream** is used to close the output stream and release the system resources used by it.

**Note**

It is important to close the output files because sometimes the buffer does not get flushed out completely.

The Character Stream Classes

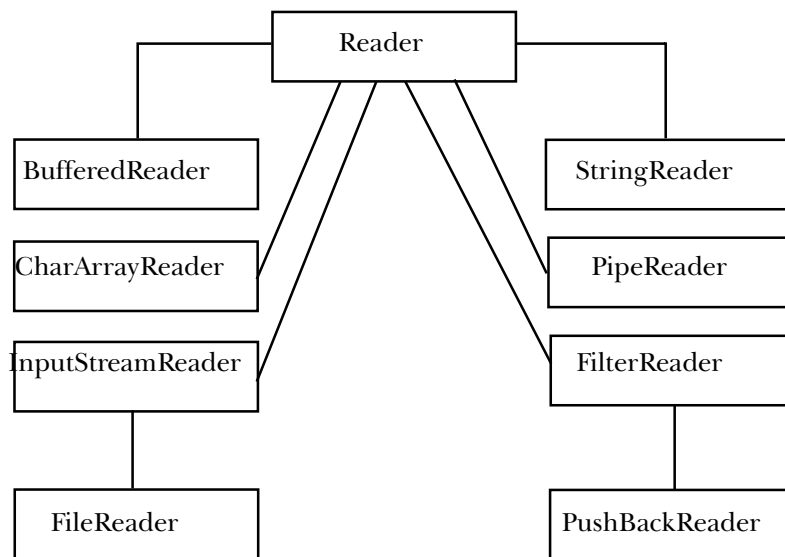
The **Character Stream** classes are used to read and write 16-bit Unicode characters. The functionality of these classes is similar to those of the Byte Stream classes. The advantage of using the **Character Stream** classes is that they do not follow a specific character, so they are not character dependent. Like the Byte Stream classes, the **Character Stream** classes also contain the following two classes:

1. The Reader Stream classes
2. The Writer Stream classes

Reader Stream Classes

The Reader Stream classes are used to read characters from files. The classes belonging to the **Reader Stream** classes are very similar to the **Input Stream** classes with the only difference that the **Input Stream** classes use bytes, whereas the Reader Stream classes use characters. In fact, both the classes use the same methods.

The hierarchy of the **Reader Stream** classes is given in Figure 12-3.

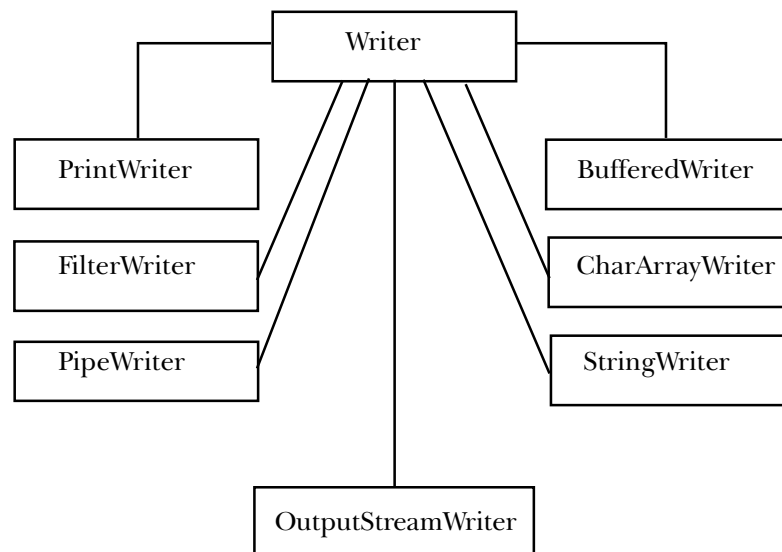


*Figure 12-3 The hierarchy of the **Reader Stream** classes*

Writer Stream Classes

The **Writer Stream** classes are used to write characters to files. The **Writer Stream** classes are similar to the **Output Stream** classes with the only difference that the **Output Stream** classes use bytes, whereas the **Writer Stream** classes use characters. Infact, both the classes use the same methods.

The hierarchy of the **Writer Stream** classes is given in Figure 12-4.



*Figure 12-4 The hierarchy of the **Writer Stream** classes*

The File Class

The **File** class is used to create files and directories in Java. It belongs to the **java.io** package. This class performs various operations related to file handling. The **File** class can be used for creating, opening, closing, and deleting a file. You can also get the name and size of a file with the help of the **File** class.

Naming Conventions for Creating a File

There are some naming conventions that must be followed while creating a file. These conventions are as follows:

1. The name of file should be unique and of string type.
2. The name of file can be divided in two parts. For example, test.text and test.dat.
3. Before using any file, you should know its purpose, whether it is meant for writing, reading, or both.
4. Data type of the file operation, whether it will be in bytes or characters.

The following example illustrates how to create a file in Java with the help of the **java.io** package:

Example 1

Write a program to create a file in Java using the **java.io** package.

The following program will create a text file on the current directory of a project.

```
//Write a program to create a file in Java      1
import java.io.*;                               2
public class NewFile                             3
{                                                 4
    public static void main(String[] args) throws IOException 5
    {                                             6
        File nf;                                7
        nf=new File("NewFile.txt");             8
        if(!nf.exists())                        9
        {                                       10
            nf.createNewFile();                 11
            System.out.println("A file with the name \"NewFile.txt\" is created in your current
            directory");                         12
        }                                       13
        else                                    14
            System.out.println("The file with the name \"NewFile.txt\" already exists"); 15
        }                                       16
    }                                           17
```

Explanation

Line 7

File nf;

This line declares an object **nf** of the class **File**.

Line 8

nf=new File("NewFile.txt");

This line creates the object **nf**, and a file with the name *NewFile.txt* is passed as a string argument to the class **File**.

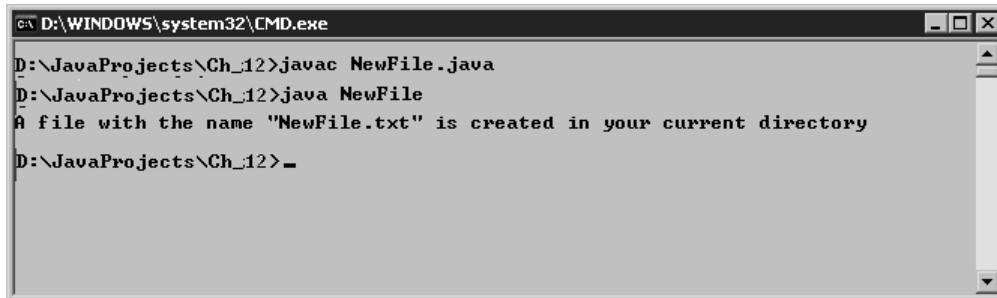
Lines 9 to 15

if(!nf.exists())

```
{
nf.createNewFile();
System.out.println("A file with the name \"NewFile.txt\" is created in your current
directory");
}
else
System.out.println("The file with the name \"NewFile.txt\" already exists");
```

These block of lines indicate that if the file with the name *NewFile.txt* does not exist in the current directory of the project, it will be created in the current directory. In case, the above file already exists, the control will pass to line 15 and the message **The file with the name NewFile.txt already exists** will be displayed. The property **exists()** checks whether the file *NewFile.txt* exists or not in the current directory. Whereas, the method **createNewFile()** creates a new file.

The output will be displayed on the screen as follows:



```

C:\D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_12>javac NewFile.java
D:\JavaProjects\Ch_12>java NewFile
A file with the name "NewFile.txt" is created in your current directory
D:\JavaProjects\Ch_12>_
  
```

Reading and Writing a Character File

There are basically two classes that are used to read and write character files, the **FileReader** and **FileWriter** classes. Go through the next example to understand the concept of reading and writing files with characters.

The following example illustrates the concept of reading and writing a file with characters:

Example 2

Write a program to illustrate the concept of reading and writing a file using the **FileReader** and **FileWriter** classes.

The following program will illustrate the concept of reading and writing a file with the help of the **FileReader** and **FileWriter** classes.

```

//Write a program to open an existing file, read it, and create another file with
the same content as the existing file.                                     1
import java.io.*;                                                         2
class Characters                                                            3
{                                                                           4
    public static void main(String args[ ])                               5
    {                                                                       6
        File OldFile=new File("OldFile.txt");                             7
        File NewFile=new File("NewFile.txt");                             8
        FileReader OldF=null;                                              9
        FileWriter NewF=null;                                             10
        try                                                                11
        {                                                                  12
            OldF=new FileReader(OldFile);                                   13
            NewF=new FileWriter(NewFile);                                   14
            int ch;                                                         15
            while((ch=OldF.read())!=-1)                                     16
            {                                                                17
                NewF.write(ch);                                             18
            }
        }
    }
}
  
```

```
} 19
System.out.println("A file with the name \"NewFile.txt\" is created and written from the
file \"OldFile.txt\" in your current directory."); 20
} 21
catch(IOException e) 22
{ 23
System.out.println(e); 24
System.exit(-1); 25
} 26
finally 27
{ 28
try 29
{ 30
OldF.close(); 31
NewF.close(); 32
} 33
catch(IOException e) {} 34
} 35
} 36
} 37
```

Before running this program, rename the existing file *NewFile.txt* to *OldFile.txt* in your current project directory. Also, write some contents in the file.

Explanation

Line 7

File OldFile=new File("OldFile.txt");

This line indicates that an object **OldFile** of the class **File** has been created and the existing file with the name *OldFile.txt* is passed as a string argument to the class **File**.

Line 8

File NewFile=new File("NewFile.txt");

This line indicates that another object **NewFile** of the class **File** has been created and a new file, which has to be created with the name *NewFile.txt*, is passed as a string argument to the class **File**.

Line 9

FileReader OldF=null;

This line indicates that an object with the name **OldF** of the class **FileReader** has been declared and initialized with the null value.

Line 10

FileWriter NewF=null;

This line indicates that an object with the name **NewF** of the class **FileWriter** has been declared and initialized with the null value.

Line 13

OldF=new FileReader(OldFile);

This line indicates that the object **OldF** has been created and the object **OldFile** is passed as an argument to the class **FileReader**.

Line 14

NewF=new FileWriter(NewFile);

This line indicates that the object **NewF** has been created and the object **NewFile** is passed as an argument to the class **FileWriter**.

Lines 16 to 19

```
while((ch=OldF.read())!=-1)  
{  
    NewF.write(ch);  
}
```

These lines indicate that first the existing file *OldFile.txt* will be read and then the contents of this file will be written in the *NewFile.txt* file. The process of reading and writing the file will be on the basis of the characters and it will be continued till the control reaches to the end of the file.

Line 22

catch(IOException e)

This line indicates that an I/O exception will be raised, if any error is found in the I/O handling of the file.

Line 31

OldF.close();

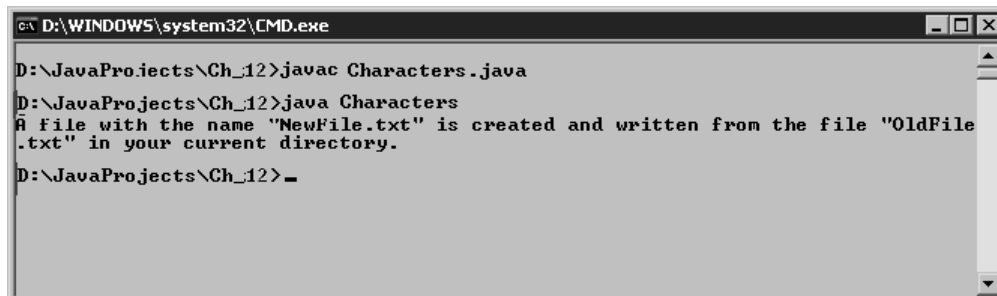
This line will close the file object **OldF** for reading.

Line 32

NewF.close();

This line will close the file object **NewF** for writing.

The output of the program will be displayed on the screen as follows:



```
D:\WINDOWS\system32\CMD.exe  
  
D:\JavaProjects\Ch_12>javac Characters.java  
D:\JavaProjects\Ch_12>java Characters  
A file with the name "NewFile.txt" is created and written from the file "OldFile  
.txt" in your current directory.  
D:\JavaProjects\Ch_12>_
```

Reading and Writing a File with Bytes

There are two classes that can be used for reading and writing a byte file. These classes are **FileInputStream** and **FileOutputStream**.

The following example illustrates the concept of reading and writing a file with bytes:

Example 3

Write a program to illustrate the concept of reading and writing a file with bytes.

The following program will illustrate the concept of reading and writing a file with bytes:

```
//Write a program that will create a new file and write some byte contents in it. 1
import java.io.*; 2
class Bytes 3
{ 4
    public static void main(String args[]) 5
    { 6
        byte states []={'C','A','L','I','F','O','R','N','I','A','\t','F','L','O','R','I','D','A'}; 7
        FileOutputStream NewFile=null; 8
        try 9
        { 10
            NewFile=new FileOutputStream("states.txt"); 11
            NewFile.write(states); 12
            System.out.println("The byte file with the name \"states.txt\" has been created in your 13
            current directory") 14
            NewFile.close(); 15
        } 16
        catch(IOException e) 17
        { 18
            System.out.println(e); 19
            System.exit(-1); 20
        } 21
    } 22
```

Explanation

Line 7

byte states []={'C','A','L','I','F','O','R','N','I','A','\t','F','L','O','R','I','D','A'};

This line indicates that an array **states** of the byte data type has been declared and initialized with some letters.

Line 8

FileOutputStream NewFile=null;

This line indicates that an object named **NewFile** of the class **FileOutputStream** has been declared and initialized with the **null** value.

Line 11

NewFile=new FileOutputStream("states.txt");

This line indicates that the object **NewFile** has been created and the file *states.txt* is passed as an argument to the constructor of the class **FileOutputStream**.

Line 12

NewFile.write(states);

This line will write the contents of the variable **states** to the object **NewFile** with the help of the method **write** method.

Line 14

NewFile.close();

This line will close the object **NewFile** for writing.

Line 16

catch(IOException e)

This line indicates that an I/O exception will be raised, if any error is found in I/O handling of the file.

The output of the program will be displayed on the screen as follows:



```
D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_12>javac Bytes.java
D:\JavaProjects\Ch_12>java Bytes
The byte file with the name "states.txt" has been created in your current directory
D:\JavaProjects\Ch_12>
```

Random Access Files

Random access means ability to access any location within a file and then read or write the data to that location in the file. The **java.io** package provides the class **RandomAccessFile** that can be used to read and write the text, bytes, or primitive Java data types of the file at any location. It treats class as a collection of records.

The **RandomAccessFile** class provides a pointer to a file that behaves like an index and indicates the location from where the read or write operation will start. This class implements both the interfaces **DataInput** and **DataOutput** to read and write the data into files.

The following code of Java is used to create an instance of the class **RandomAccessFile** to read and write the file:

```
RandomAccessFile Ran_File = new RandomAccessFile(File_name, "r");
```

The above code creates an instance of the class **RandomAccessFile**. The constructor **RandomAccessFile()** takes two arguments: First is the file name that is used to read and write the data, and second is the operation mode that can be read (r) or read-write (rw) mode. The above code is used to open the file in the read mode (r).

The following code is used to open the file in the read-write mode:

```
RandomAccessFile Ran_File = new RandomAccessFile(File_name, "rw");
```

The **RandomAccessFile** constructor checks the existence of the file that you passed as an argument. If the file does not exist, it will throw an **IOException**. Also, if an attempt is made to read the end of the file, the **read** method will throw the exception **EOFException** (which is a part of the **IOException**).

The class **RandomAccessFile** provides various methods for different operations. These methods are as follows:

```
Close()
getChannel()
getFD()
getFilePointer()
length()
read()
read(byte [] b)
read(byte [] b, int off, int len)
write(byte [] b)
write(byte [] b, int off, int len)
write(int b)
seek(long pos)
setLength(long newLength)
skipBytes()
```

The following example illustrates the concept of writing a file using the class **RandomAccessFile** and its methods:

Example 4

Write a program that will illustrate the concept of writing a file using the class **RandomAccessFile**.

The following program will illustrate the concept of writing a file using the class **RandomAccessFile**.

```
//Write a program that will open the existing file in the read-write mode
and write bytes in it.
import java.io.* ;
public class RandAccessFile
{
public static void main(String[] args)
{
BufferedReader inF = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter File name : ");
String str = inF.readLine();
```

```

File file = new File(str);                                10
if(!file.exists())                                       11
{                                                         12
System.out.println("File does not exist.");              13
System.exit(0);                                          14
}                                                         15
try                                                       16
{                                                         17
RandomAccessFile rndFile = new RandomAccessFile(file,"rw"); 18
rndFile.seek(file.length());                             19
rndFile.writeBytes("www.cadcim.com");                    20
rndFile.writeBytes("The random access means to go any location within the file."); 21
rndFile.close();                                         22
System.out.println("Write Successfully");                23
}                                                         24
catch(IOException e)                                     25
{                                                         26
System.out.println(e.getMessage());                      27
}                                                         28
}                                                         29
}                                                         30

```

Explanation

Line 7

BufferedReader inF = new BufferedReader(new InputStreamReader(System.in));

This line indicates that an object **inF** of the **BufferedReader** class been created and the instance of the class **InputStreamReader** is passed as an argument. The argument **System.in** of the constructor of the class **InputStreamReader** is used to read the user's input.

Line 8

System.out.print("Enter File name : ");

This line will prompt a message **Enter File name**.

Line 9

String str = in.readLine();

This line declares the variable **str** as **String** and assigns the value entered by the user (file name) to it.

Line 10

File file = new File(str);

This line indicates that an object **file** of the class **File** has been created and initialized by passing the file name as string (**str**) to the constructor of the class **File**.

Lines 11 to 15

if(!file.exists())

{

System.out.println("File does not exist.");


```
System.exit(0);  
}
```

These lines check the existence of the specified file. If the file does not exist, they will print a message **File does not exist.**, and the program will exit without executing the rest of the codes.

Line 18

```
RandomAccessFile rndFile = new RandomAccessFile(file, "rw");
```

This line indicates that an object **rndFile** of the class **RandomAccessFile** has been created and it is initialized by passing the object **file** and the mode **rw** as String (**r** for read and **rw** for read-write) to the constructor of the class **RandomAccessFile**.

Line 19

```
rndFile.seek(file.length());
```

This line will set the file-pointer to the end of the file. Here, the **file.length()** function will return the length of the file, which is passed as an argument to the method **seek()**. The **seek()** method sets the file-pointer to the position indicated by the value returned by **file.length()**.

Lines 20 to 23

```
rndFile.writeBytes("www.cadcim.com");  
rndFile.writeBytes("The random access means to go any location within the file.");  
rndFile.close();  
System.out.println("Write Successfully");
```

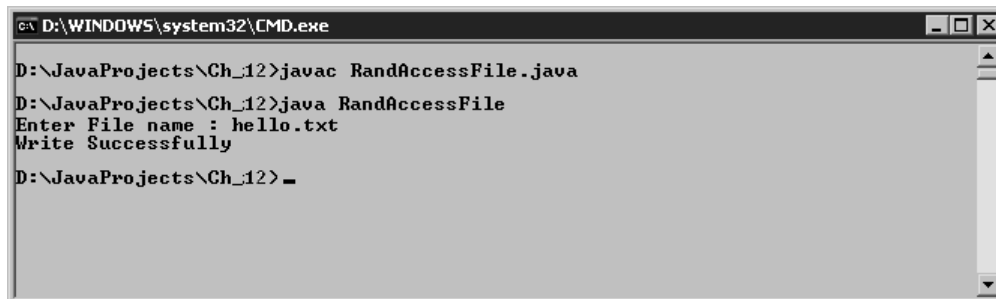
These lines will write bytes to the file. The **writeBytes()** method of the class **RandomAccessFile** takes string as an argument and writes it to the file. Make sure the file is opened in the write mode. The **close()** method of the class **RandomAccessFile** will close the random access file stream and set the resources used by the program free.

Lines 25 to 28

```
catch(IOException e)  
{  
System.out.println(e.getMessage());  
}
```

These lines will handle the error, which will occur due to the I/O operation. Here, the function **e.getMessage()** returns the text message regarding the error.

The output of the program will be displayed on the screen as follows:



The following example illustrates the concept of reading a file using the class **RandomAccessFile** and its methods:

Example 5

Write a program to illustrate the concept of reading a file using the class **RandomAccessFile**.

The following program will illustrate the concept of reading a using with the class **RandomAccessFile**:

```
//Write a program that will open the existing file in the read mode and write bytes in it. 1
import java.io.*; 2
public class File_read 3
{ 4
    public static void main(String[] args) 5
    { 6
        BufferedReader inF = new BufferedReader(new InputStreamReader(System.in)); 7
        System.out.print("Enter File name : "); 8
        String strFile = inF.readLine(); 9
        File file = new File(strFile); 10
        if(!file.exists()) 11
        { 12
            System.out.println("File does not exist."); 13
            System.exit(0); 14
        } 15
        try 16
        { 17
            RandomAccessFile rndFile = new RandomAccessFile(file,"r"); 18
            int ctrl=(int)rndFile.length(); 19
            System.out.println("Length: " + ctrl); 20
            rndFile.seek(0); 21
            for(int ct = 0; ct < ctrl; ct++) 22
            { 23
                byte b = rndFile.readByte(); 24
                System.out.print((char)b); 25
            } 26
            rndFile.close(); 27
        } 28
        catch(IOException e) 29
        { 30
            System.out.println(e.getMessage()); 31
        } 32
    } 33
```

Explanation

Line 7

BufferedReader inF = new BufferedReader(new InputStreamReader(System.in));

This line indicates that an object **inF** of the **BufferedReader** class been created and the instance of the class **InputStreamReader** is passed as an argument. The argument **System.in** of the constructor of the class **InputStreamReader** is used to read the user's input.

Line 8

```
System.out.print("Enter File name : ");
```

This line will prompt a message **Enter File name**.

Line 9

```
String strFile = inF.readLine();
```

This line declares the variable **strFile** as **String** and assigns the value entered by the user (file name) to it.

Line 10

```
File file = new File(strFile);
```

This line indicates that an object **file** of the class **File** has been created and it is initialized by passing the file name as string (**strFile**) to the constructor of the class **File**.

Lines 11 to 15

```
if(!file.exists())
```

```
{
```

```
System.out.println("File does not exist.");
```

```
System.exit(0);
```

```
}
```

These lines check the existence of the file. If the file does not exist, it will print a message **File does not exist**, and the program will exit without executing the rest of the codes.

Line 18

```
RandomAccessFile rand = new RandomAccessFile(file,"r");
```

This line indicates that an object **rand** of the class **RandomAccessFile** has been created and it is initialized by passing the object **file** and mode **r** as string(**r** for read and **rw** for read-write) to the constructor of the class **RandomAccessFile**.

Lines 19

```
int ctrl=(int)rand.length();
```

These lines indicate that the variable **ctrl** is declared as **int**. Also, the variable **ctrl** has been assigned with a value returned by **randFile.length()**. Here, **randFile.length()** returns the length of the file that you assign at the declaration of the **randFile** object (see line 18).

Line 20

```
System.out.println("Length: " + ctrl);
```

This line will print **Length:** and the value of **ctrl**.

Line 21

```
randFile.seek(0);
```

In this line, the **seek()** method of the class **RandomAccessFile** sets the file-pointer to the beginning of the file, at which the next read or write event occurs.

Lines 22 to 26

```
for(int ct = 0; ct < ctrl; ct++)  
{  
    byte b = rndFile.readByte();  
    System.out.print((char)b);  
}
```

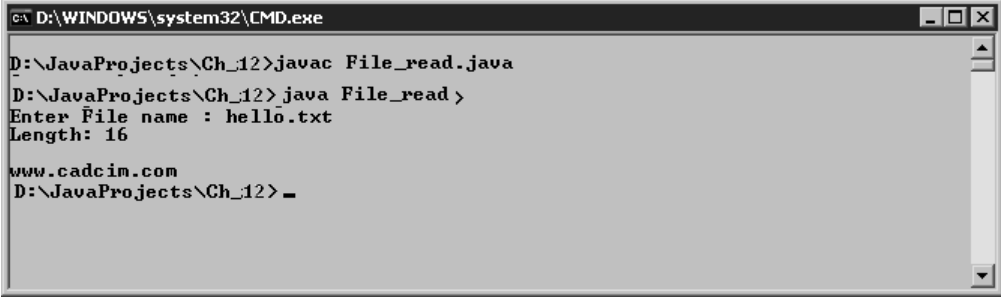
These lines will read bytes from the file. The method **`rndFile.readBytes()`** will read bytes from the existing file, which is passed as an argument (check line 18). The **`for`** loop will be execute till **`rndFile.readBytes()`** reads bytes from the file, and **`(char)b`** will convert bytes to character.

Lines 29 to 32

```
catch(IOException e)  
{  
    System.out.println(e.getMessage());  
}
```

These lines will handle the error, which occurs during to the I/O operation. Here, the method **`e.getMessage`** will return the text message regarding the error.

The output of the program will be displayed on the screen as follows:



```
C:\ D:\WINDOWS\system32\CMD.exe  
  
D:\JavaProjects\Ch_12>javac File_read.java  
D:\JavaProjects\Ch_12>java File_read >  
Enter File name : hello.txt  
Length: 16  
  
www.cadcam.com  
D:\JavaProjects\Ch_12> _
```

Self-Evaluation Test

Answer the following questions and then compare them to those given at the end of this chapter:

1. The term **stream** indicates the _____ of data.
2. The **Byte Stream** classes perform the input/output operations on _____.
3. The names of the classes that are used for reading and writing byte files are _____ and _____.
4. The class **InputStream** is inherited from the class _____ class.
5. The **OutputStream** classes are used to write the bytes to the _____ location.
6. The **Reader Stream** classes are used to read _____ from files.
7. The _____ method is used to mark the current position in the input stream.
8. The class _____ extends the class **FilterInputStream**.
9. The **Character Stream** classes contain two classes: _____ and _____.
10. There are basically two classes for reading and writing character files, _____ and _____ classes.

Review Questions

Answer the following questions:

1. List different types of the **Stream** classes.
2. Write different sources that can be read by the class **InputStream**.
3. What is the **reset** method? Explain.
4. Define the **OutputStream** classes.
5. Create a file with the help of the class **File**.

Exercises

Exercise 1

Write a program to read the bytes from existing file.

Exercise 2

Write a program to append the bytes in the existing file using the class **RandomAccessFile**.

Answers to Self-Evaluation Test

1. flow, 2. bytes, 3. **FileInputStream**, **FileOutputStream**, 4. **Object**, 5. memory, 6. characters, 7. mark, 8. **DataInputStream**, 9. Reader Stream, Writer Stream, 10. **FileReader**, **FileWriter**