JAVA - FILES AND I/O

The java.io package contains nearly every class you might ever need to perform input and output I/O in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. there are two kinds of Streams

- **InPutStream:** The InputStream is used to read data from a source.
- **OutPutStream:** the OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException
   {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content:

```
This is test for copy file.
```

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , **FileReader** and **FileWriter.**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file *havingunicodecharacters* into an output file:

```
import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException
   {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content:

```
This is test for copy file.
```

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware if C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT

and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader** to read standard input stream until the user types a "q":

```
import java.io.*;

public class ReadConsole {
   public static void main(String args[]) throws IOException
   {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

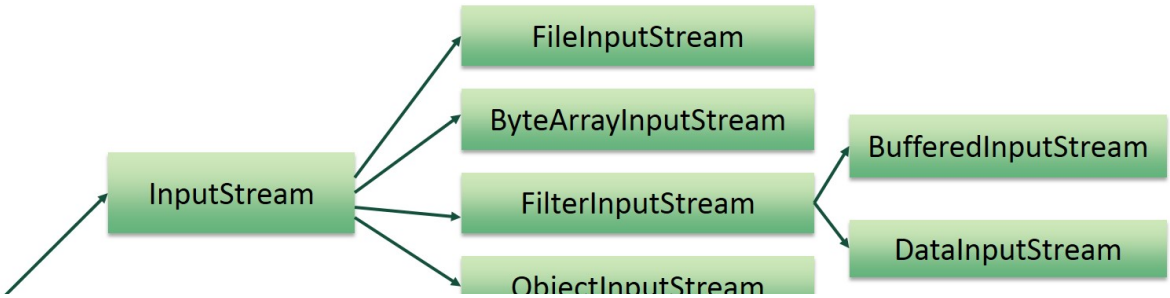
Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

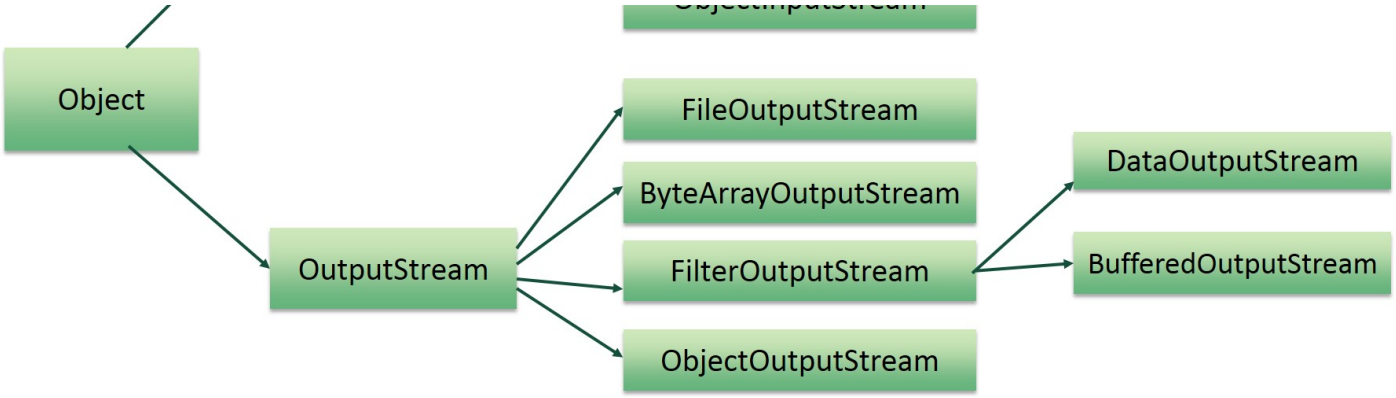
```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.





The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalizethrows IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read*intr*throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**public int read*byte*[]*r* throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	**public int available throws IOException{}**

Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links:

- ByteArrayInputStream
- DataInputStream

FileOutputStream:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException
2	**protected void finalizethrows IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write*intw*throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write*byte[]w*** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

   try{
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories:

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
   public static void main(String args[]) {
```

```
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // Create directory now.
      d.mkdirs();
   }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash / on a Windows version of Java, the path will still resolve correctly.

Listing Directories:

You can use **list** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;

public class ReadDir {
   public static void main(String[] args) {

      File file = null;
      String[] paths;

      try{
         // create new file object
         file = new File("/tmp");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths)
         {
            // prints filename and directory name
            System.out.println(path);
         }
      }catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

This would produce following result based on the directories and files available in your **/tmp** directory:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

JAVA - STRINGS CLASS

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings:

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

This would produce the following result:

```
hello.
```

Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

String Length:

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length method, which returns the number of characters contained in the string object.

Below given program is an example of **length** , method String class.

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

This would produce the following result:

```
String Length is : 17
```

Concatenating Strings:

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat method with string literals, as in:

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in:

```
"Hello," + " world" + "!"
```

which results in:

```
"Hello, world!"
```

Let us look at the following example:

```
public class StringDemo {

   public static void main(String args[]) {
      String string1 = "saw I was ";
      System.out.println("Dot " + string1 + "Tod");
   }
}
```

This would produce the following result:

```
Dot saw I was Tod
```

Creating Format Strings:

You have printf and format methods to print output with formatted numbers. The String class has an equivalent class method, format, that returns a String object rather than a PrintStream object.

Using String's static format method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of:

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

you can write:

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String Methods:

Here is the list of methods supported by String class:

SN	Methods with Description
1	char charAt*intindex* Returns the character at the specified index.

2

int compareTo*Objecto*

Compares this String to another Object.

3

int compareTo*StringanotherString*

Compares two strings lexicographically.

4

int compareToIgnoreCase*Stringstr*

Compares two strings lexicographically, ignoring case differences.

5

String concat*Stringstr*

Concatenates the specified string to the end of this string.

6

boolean contentEquals*StringBuffersb*

Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.

7

static String copyValueOf*char[]data*

Returns a String that represents the character sequence in the array specified.

8

static String copyValueOf*char[]data, intoffset, intcount*

Returns a String that represents the character sequence in the array specified.

9

boolean endsWith*Stringsuffix*

Tests if this string ends with the specified suffix.

10

boolean equals*ObjectanObject*

Compares this string to the specified object.

11

boolean equalsIgnoreCase*StringanotherString*

Compares this String to another String, ignoring case considerations.

12

byte getBytes

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

13

byte[] getBytes(String charsetName

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

14

void getChars*intsrcBegin, intsrcEnd, char[]dst, intdstBegin*

Copies characters from this string into the destination character array.

15

int hashCode

Returns a hash code for this string.

16

int indexOf*intch*

Returns the index within this string of the first occurrence of the specified character.

17

int indexOf*intch, intfromIndex*

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

18

int indexOf*Stringstr*

Returns the index within this string of the first occurrence of the specified substring.

19

int indexOf*Stringstr, intfromIndex*

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index

20

String intern

Returns a canonical representation for the string object.

21

int lastIndexOf*intch*

Returns the index within this string of the last occurrence of the specified character.

22

int lastIndexOf*intch, intfromIndex*

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.

23

int lastIndexOf*Stringstr*

Returns the index within this string of the rightmost occurrence of the specified substring.

24

int lastIndexOf*Stringstr, intfromIndex*

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25

int length

Returns the length of this string.

26

boolean matches*Stringregex*

Tells whether or not this string matches the given regular expression.

27

boolean regionMatches*booleanignoreCase, inttoffset, Stringother, intooffset, intlen*

Tests if two string regions are equal.

28

boolean regionMatches*inttoffset, Stringother, intooffset, intlen*

Tests if two string regions are equal

29

String replace*charoldChar, charnewChar*

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

30

String replaceAll(String regex, String replacement

Replaces each substring of this string that matches the given regular expression with the given replacement.

31

String replaceFirst*Stringregex, Stringreplacement*

Replaces the first substring of this string that matches the given regular expression with the given replacement.

32

String[] split*Stringregex*

Splits this string around matches of the given regular expression.

33

String[] split*Stringregex, intlimit*

Splits this string around matches of the given regular expression.

34

boolean startsWith*Stringprefix*

Tests if this string starts with the specified prefix.

35

boolean startsWith*Stringprefix, inttoffset*

Tests if this string starts with the specified prefix beginning a specified index.

36

CharSequence subSequence*intbeginIndex, intendIndex*

Returns a new character sequence that is a subsequence of this sequence.

37

String substring*intbeginIndex*

Returns a new string that is a substring of this string.

38

String substring*intbeginIndex, intendIndex*

Returns a new string that is a substring of this string.

39

char[] toCharArray

Converts this string to a new character array.

40

String toLowerCase

Converts all of the characters in this String to lower case using the rules of the default locale.

41

String toLowerCase*Localelocale*

Converts all of the characters in this String to lower case using the rules of the given Locale.

42

String toString

This object *whichisalreadyastring*! is itself returned.

43

String toUpperCase

Converts all of the characters in this String to upper case using the rules of the default locale.

44

String toUpperCase*Localelocale*

Converts all of the characters in this String to upper case using the rules of the given Locale.

45

String trim

Returns a copy of the string, with leading and trailing whitespace omitted.

46

static String valueOf*primitivedatatypex*

Returns the string representation of the passed data type argument.