CHP-17
Java Methods

A Javamethod is a collection of statements that are grouped together to perform an operation. When you call the System.out.println method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating a Method:

In general, a method has the following syntax:

```
modifier returnValueType methodName(list of parameters){
// Method body;
}
```

A method definition consists of a method header and a method body. Here are all the parts of a method:

- **Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.
- **Return Type:** A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void**.
- **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.
- **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method Body:** The method body contains a collection of statements that define what the method does.

CHAPTER

Note: In certain other languages, methods are referred to as procedures and functions. A method with a nonvoid return value type is called a function; a method with a void return value type is called a procedure.

Example:

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** Return the max between two numbers */
public static int max(int num1,int num2){
int result;
if(num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Calling a Method:

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not.
When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.
If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30,40);
```

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

Example:

Following is the example to demonstrate how to define a method and how to call it:

```
public class TestMax{
/** Main method */
public static void main(String[] args){
```

```
int i =5;
int j =2;
int k = max(i, j);
System.out.println("The maximum between "+ i +
" and "+ j +" is "+ k);
}
/** Return the max between two numbers */
public static int max(int num1,int num2){
int result;
if(num1 > num2)
result = num1;
else
result = num2;
return result;
}
}
```

This would produce the following result:

```
The maximum between 5and2is5
```

This program contains the main method and the max method. The main method is just like any other method except that it is invoked by the JVM.
The main method's header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter is an array of String.

The void Keyword:

This section shows how to declare and invoke a void method. Following example gives a program that declares a method named printGrade and invokes it to print the grade for a given score.

Example:

```
public class TestVoidMethod{
public static void main(String[] args){
printGrade(78.5);
}
public static void printGrade(double score){
if(score >=90.0){
System.out.println('A');
}
elseif(score >=80.0){
System.out.println('B');
}
elseif(score >=70.0){
System.out.println('C');
}
elseif(score >=60.0){
System.out.println('D');
}
else{
System.out.println('F');
}
```

```
}
}
```

This would produce the following result:

```
C
```

Here the printGrade method is a void method. It does not return any value. A call to a void method must be a statement. So, it is invoked as a statement in line 3 in the main method. This statement is like any Java statement terminated with a semicolon.

Passing Parameters by Values:

When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.
For example, the following method prints a message n times:

```
public static void nPrintln(String message,int n){
for(int i =0; i < n; i++)
System.out.println(message);
}
```

Here, you can use nPrintln("Hello", 3) to print "Hello" three times. The nPrintln("Hello", 3) statement passes the actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times. However, the statement nPrintln(3, "Hello") would be wrong.
When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.
For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing the value of x to y.

Example:

Following is a program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The swap method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

```
public class TestPassByValue{
public static void main(String[] args){
int num1 =1;
int num2 =2;
System.out.println("Before swap method, num1 is "+num1 +" and num2 is "+ num2);
// Invoke the swap method
swap(num1, num2);
System.out.println("After swap method, num1 is "+num1 +" and num2 is "+ num2);
}
/** Method to swap two variables */
public static void swap(int n1,int n2){
System.out.println("\tInside the swap method");
System.out.println("\t\tBefore swapping n1 is "+ n1+" n2 is "+ n2);
// Swap n1 with n2
int temp = n1;
```

```
n1 = n2;
n2 = temp;
System.out.println("\t\tAfter swapping n1 is "+ n1+" n2 is "+ n2);
}
}
```

This would produce the following result:

```
Before swap method, num1 is1and num2 is2
Inside the swap method
Before swapping n1 is1 n2 is2
```

```
After swapping n1 is2 n2 is1
After swap method, num1 is1and num2 is2
```

Overloading Methods:

The max method that was used earlier works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1,double num2){
if(num1 > num2)
return num1;
else
return num2;
}
```

If you call max with int parameters, the max method that expects int parameters will be invoked; if you call max with double parameters, the max method that expects double parameters will be invoked. This is referred to as **method overloading**; that is, two methods have the same name but different parameter lists within one class.
The Java compiler determines which method is used based on the method signature. Overloading methods can make programs clearer and more readable. Methods that perform closely related tasks should be given the same name.
Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the compiler cannot determine the most specific match. This is referred to as ambiguous invocation.

The Scope of Variables:

The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a local variable.
The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.
A parameter is actually a local variable. The scope of a method parameter covers the entire method.
A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable as shown below:

You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

Using Command---Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).
A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy.they are stored as strings in the String array passed to main( ).

Example:

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine{
public static void main(String args[]){
for(int i=0; i<args.length; i++){
System.out.println("args["+ i +"]: "+args[i]);
}
}
}
```

Try executing this program as shown here:

```
java CommandLine this is a command line 200-100
```

This would produce the following result:

```
args[0]:this
args[1]:is
args[2]: a
args[3]: command
args[4]: line
```

```
args[5]:200
args[6]:-100
```

The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.
All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass{
int x;
// Following is the constructor
MyClass(){
x =10;
}
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo{
public static void main(String args[]){
MyClass t1 =new MyClass();
MyClass t2 =new MyClass();
System.out.println(t1.x +" "+ t2.x);
}
}
```

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass{
int x;
// Following is the constructor
MyClass(int i ){
x = i;
}
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo{
```

```
public static void main(String args[]){
MyClass t1 =new MyClass(10);
MyClass t2 =new MyClass(20);
System.out.println(t1.x +" "+ t2.x);
}
}
```

This would produce the following result:

```
1020
```

Variable Arguments(var---args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example:

```
public class VarargsDemo{
public static void main(String args[]){
// Call method with variable args
printMax(34,3,3,2,56.5);
printMax(new double[]{1,2,3});
}
public static void printMax(double... numbers){
if(numbers.length ==0){
System.out.println("No argument passed");
return;
}
double result = numbers[0];
for(int i =1; i < numbers.length; i++)
if(numbers[i]> result)
result = numbers[i];
System.out.println("The max value is "+ result);
}
}
```

This would produce the following result:

```
The max value is 56.5
The max value is 3.0
```

The finalize( ) Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.
For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.
Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.
The finalize( ) method has this general form:

```
protected void finalize()
{
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.
This means that you cannot know whenor even iffinalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

CHP-18
Java Streams, 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.
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.
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();
}
}
```

CHAPTER

```
}
}
```

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 (having unicode characters) 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 finalize()throws 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(int r)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

ByteArrayInputStream

The ByteArrayInputStream class allows a buffer in the memory to be used as an InputStream. The input source is a byte array. There are following forms of constructors to create ByteArrayInputStream objects
Takes a byte array as the parameter:

```
ByteArrayInputStream bArray = new ByteArrayInputStream(byte [] a);
```

Another form takes an array of bytes, and two ints, where **off** is the first byte to be read and **len** is the number of bytes to be read.

```
ByteArrayInputStream bArray = new ByteArrayInputStream(byte []a,
int off,
int len)
```

Once you have *ByteArrayInputStream* object in hand then there is a list of helper methods which can be used to read the stream or to do other operations on the stream.
SN Methods with Description
1
public int read()
This method reads the next byte of data from the InputStream. Returns an int as the next byte of
data. If it is end of file then it returns -1.
2
public int read(byte[] r, int off, int len)
This method reads upto **len** number of bytes starting from **off** from the input stream into an array.
Returns the total number of bytes read. If end of file -1 will be returned.
3
public int available()
Gives the number of bytes that can be read from this file input stream. Returns an int that gives the
number of bytes to be read.
4
public void mark(int read)
This sets the current marked position in the stream. The parameter gives the maximum limit of
bytes that can be read before the marked position becomes invalid.
5 **public long skip(long n)**
Skips n number of bytes from the stream. This returns the actual number of bytes skipped.

Example:

Following is the example to demonstrate ByteArrayInputStream and ByteArrayOutputStream

```
import java.io.*;
public class ByteStreamTest {
public static void main(String args[])throws IOException {
ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
while( bOutput.size()!= 10 ) {
// Gets the inputs from the user
bOutput.write(System.in.read());
}
byte b [] = bOutput.toByteArray();
System.out.println("Print the content");
for(int x= 0 ; x < b.length; x++) {
// printing the characters
System.out.print((char)b[x] + " ");
}
System.out.println(" ");
int c;
ByteArrayInputStream bInput = new ByteArrayInputStream(b);
System.out.println("Converting characters to Upper case " );
for(int y = 0 ; y < 1; y++ ) {
while(( c= bInput.read())!= -1) {
System.out.println(Character.toUpperCase((char)c));
}
bInput.reset();
}
}
}
```

Here is the sample run of the above program:

```
asdfghjkly
Print the content
a s d f g h j k l y
Converting characters to Upper case
A
S
D
F
G
H
J
K
L
Y
```

DataInputStream

The DataInputStream is used in the context of DataOutputStream and can be used to read primitives.
Following is the constructor to create an InputStream:

```
InputStream in = DataInputStream(InputStream in);
```

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

SN Methods with Description

1

public final int read(byte[] r, int off, int len)throws IOException

Reads up to len bytes of data from the input stream into an array of bytes. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.

2

Public final int read(byte [] b)throws IOException

Reads some bytes from the inputstream an stores in to the byte array. Returns the total number of bytes read into the buffer otherwise -1 if it is end of file.

3

(a) public final Boolean readBooolean()throws IOException,
(b) public final byte readByte()throws IOException,
(c) public final short readShort()throws IOException
(d) public final Int readInt()throws IOException

These methods will read the bytes from the contained InputStream. Returns the next two bytes of the InputStream as the specific primitive type.

4

public String readLine() throws IOException

Reads the next line of text from the input stream. It reads successive bytes, converting each byte separately into a character, until it encounters a line terminator or end of file; the characters read are then returned as a String.

Example:

Following is the example to demonstrate DataInputStream and DataInputStream. This example reads 5 lines given in a file test.txt and convert those lines into capital letters and finally copies them into another file test1.txt.

```
import java.io.*;
public class Test{
public static void main(String args[])throws IOException{
DataInputStream d = new DataInputStream(new
FileInputStream("test.txt"));
DataOutputStream out = new DataOutputStream(new
FileOutputStream("test1.txt"));
String count;
while((count = d.readLine()) != null){
String u = count.toUpperCase();
System.out.println(u);
out.writeBytes(u + " ,");
}
d.close();
out.close();
}
}
```

Here is the sample run of the above program:

```
THIS IS TEST 1 ,
THIS IS TEST 2 ,
THIS IS TEST 3 ,
THIS IS TEST 4 ,
THIS IS TEST 5 ,
```

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 finalize()throws 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(int w)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

ByteArrayOutputStream

The ByteArrayOutputStream class stream creates a buffer in memory and all the data sent to the stream is stored in the buffer. There are following forms of constructors to create ByteArrayOutputStream objects
Following constructor creates a buffer of 32 byte:

```
OutputStream bOut = new ByteArrayOutputStream()
```

Following constructor creates a buffer of size int a:

```
OutputStream bOut = new ByteArrayOutputStream(int a)
```

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

SN Methods with Description

1
public void reset()
This method resets the number of valid bytes of the byte array output stream to zero, so all the accumulated output in the stream will be discarded.
2
public byte[] toByteArray()
This method creates a newly allocated Byte array. Its size would be the current size of the output stream and the contents of the buffer will be copied into it. Returns the current contents of the output stream as a byte array.
3
public String toString()
Converts the buffer content into a string. Translation will be done according to the default character encoding. Returns the String translated from the buffer's content.
4 **public void write(int w)**
Writes the specified array to the output stream.
5 **public void write(byte []b, int of, int len)**
Writes len number of bytes starting from offset off to the stream.
6 **public void writeTo(OutputStream outSt)**
Writes the entire content of this Stream to the specified stream argument.

Example:

Following is the example to demonstrate ByteArrayOutputStream and ByteArrayOutputStream

```
import java.io.*;
public class ByteStreamTest {
public static void main(String args[])throws IOException {
ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
while( bOutput.size()!= 10 ) {
// Gets the inputs from the user
bOutput.write(System.in.read());
}
byte b [] = bOutput.toByteArray();
System.out.println("Print the content");
for(int x= 0 ; x < b.length; x++) {
//printing the characters
System.out.print((char)b[x] + " ");
}
System.out.println(" ");
int c;
ByteArrayOutputStream bInput = new ByteArrayOutputStream(b);
System.out.println("Converting characters to Upper case " );
for(int y = 0 ; y < 1; y++ ) {
while(( c= bInput.read())!= -1) {
System.out.println(Character.toUpperCase((char)c));
}
bInput.reset();
}
```

```
}
}
```

Here is the sample run of the above program:

```
asdfghjkly
Print the content
a s d f g h j k l y
Converting characters to Upper case
A
S
D
F
G
H
J
K
```

L
Y

DataOutputStream

The DataOutputStream stream let you write the primitives to an output source.
Following is the constructor to create a DataOutputStream.

```
DataOutputStream out = DataOutputStream(OutputStream out);
```

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

SN Methods with Description

1 **public final void write(byte[] w, int off, int len)throws IOException**
Writes len bytes from the specified byte array starting at point off , to the underlying stream.

2
Public final int write(byte [] b)throws IOException
Writes the current number of bytes written to this data output stream. Returns the total number of bytes write into the buffer.

3
(a) public final void writeBooolean()throws IOException,
(b) public final void writeByte()throws IOException,
(c) public final void writeShort()throws IOException
(d) public final void writeInt()throws IOException
These methods will write the specific primitive type data into the output stream as bytes.

4 **Public void flush()throws IOException**
Flushes the data output stream.

5
public final void writeBytes(String s) throws IOException
Writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits.

Example:

Following is the example to demonstrate DataInputStream and DataOutputStream. This example reads 5 lines given in a file test.txt and converts those lines into capital letters and finally copies them into another file test1.txt.

```
import java.io.*;
public class Test{
public static void main(String args[])throws IOException{
DataInputStream d = new DataInputStream(new
FileInputStream("test.txt"));
DataOutputStream out = new DataOutputStream(new
FileOutputStream("test1.txt"));
String count;
while((count = d.readLine()) != null){
String u = count.toUpperCase();
System.out.println(u);
out.writeBytes(u + " ,");
}
d.close();
out.close();
}
}
```

Here is the sample run of the above program:

```
THIS IS TEST 1 ,
THIS IS TEST 2 ,
THIS IS TEST 3 ,
THIS IS TEST 4 ,
THIS IS TEST 5 ,
```

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

File Class

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc.

The File object represents the actual file/directory on the disk. There are following constructors to create a File object:

Following syntax creates a new File instance from a parent abstract pathname and a child pathname string.

```
File(File parent, String child);
```

Following syntax creates a new File instance by converting the given pathname string into an abstract pathname.

```
File(String pathname)
```

Following syntax creates a new File instance from a parent pathname string and a child pathname string.

```
File(String parent, String child)
```

Following syntax creates a new File instance by converting the given file: URI into an abstract pathname.

```
File(URI uri)
```

Once you have *File* object in hand then there is a list of helper methods which can be used manipulate the files.

SN Methods with Description

1 **public String getName()**

Returns the name of the file or directory denoted by this abstract pathname.

2

public String getParent()

Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.

3

public File getParentFile()

Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.

4 **public String getPath()**

Converts this abstract pathname into a pathname string.

5

public boolean isAbsolute()

Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise

6 **public String getAbsolutePath()**
Returns the absolute pathname string of this abstract pathname.
7
public boolean canRead()
Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8
public boolean canWrite()
Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9
public boolean exists()
Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise
10
public boolean isDirectory()
Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
11
public boolean isFile()
Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.
12
public long lastModified()
Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.
13
public long length()
Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
14
public boolean createNewFile() throws IOException
Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
15
public boolean delete()
Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.
16
public void deleteOnExit()
Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
17
public String[] list()
Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
18 **public String[] list(FilenameFilter filter)**
Returns an array of strings naming the files and directories in the directory denoted by this abstract

pathname that satisfy the specified filter.
20 **public File[] listFiles()**
Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
21
public File[] listFiles(FileFilter filter)
Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

22
public boolean mkdir()
Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.
23
public boolean mkdirs()
Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
24
public boolean renameTo(File dest)
Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.
25
public boolean setLastModified(long time)
Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise.
26
public boolean setReadOnly()
Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.
27
public static File createTempFile(String prefix, String suffix, File directory) throws IOException
Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.
28
public static File createTempFile(String prefix, String suffix) throws IOException
Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.
29
public int compareTo(File pathname)
Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
30
public int compareTo(Object o)
Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
31
public boolean equals(Object obj)
Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.
32
public String toString()
Returns the pathname string of this abstract pathname. This is just the string returned by the getPath() method.

Example:

Following is the example to demonstrate File object:

```
package com.tutorialspoint;
import java.io.File;
public class FileDemo {
public static void main(String[] args) {
File f = null;
String[] strs = {"test1.txt", "test2.txt"};
try{
// for each string in string array
for(String s:strs )
{
```

```
// create new file
f= new File(s);
// true if the file is executable
boolean bool = f.canExecute();
// find the absolute path
String a = f.getAbsolutePath();
// prints absolute path
System.out.print(a);
// prints
System.out.println(" is executable: "+ bool);
}
}catch(Exception e){
// if any I/O error occurs
e.printStackTrace();
}
}
}
```

Consider there is an executable file test1.txt and another file test2.txt is non executable in current directory, Let us compile and run the above program, this will produce the following result:

```
test1.txt is executable: true
test2.txt is executable: false
```

FileReader Class

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.
This class has several constructors to create required objects.
Following syntax creates a new FileReader, given the File to read from.

```
FileReader(File file)
```

Following syntax creates a new FileReader, given the FileDescriptor to read from.

```
FileReader(FileDescriptor fd)
```

Following syntax creates a new FileReader, given the name of the file to read from.

```
FileReader(String fileName)
```

Once you have *FileReader* object in hand then there is a list of helper methods which can be used manipulate the files.

SN Methods with Description

1 **public int read() throws IOException**

Reads a single character. Returns an int, which represents the character read.

2 **public int read(char [] c, int offset, int len)**

Reads characters into an array. Returns the number of characters read.

Example:

Following is the example to demonstrate class:

```
import java.io.*;
public class FileRead{
public static void main(String args[])throws IOException{
File file = new File("Hello1.txt");
// creates the file
file.createNewFile();
// creates a FileWriter Object
FileWriter writer = new FileWriter(file);
// Writes the content to the file
writer.write("This\n is\n an\n example\n");
writer.flush();
writer.close();
//Creates a FileReader Object
FileReader fr = new FileReader(file);
char [] a = new char[50];
fr.read(a); // reads the content to the array
for(char c : a)
System.out.print(c); //prints the characters one by one
fr.close();
}
}
```
This would produce the following result:
```
This
is
an
example
```
FileWriter Class
This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters.

This class has several constructors to create required objects.
Following syntax creates a FileWriter object given a File object.
```
FileWriter(File file)
```
Following syntax creates a FileWriter object given a File object.
```
FileWriter(File file, boolean append)
```
Following syntax creates a FileWriter object associated with a file descriptor.
```
FileWriter(FileDescriptor fd)
```
Following syntax creates a FileWriter object given a file name.
```
FileWriter(String fileName)
```
Following syntax creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.
```
FileWriter(String fileName, boolean append)
```
Once you have *FileWriter* object in hand, then there is a list of helper methods, which can be used manipulate the files.

SN	Methods with Description
1	**public void write(int c) throws IOException** Writes a single character.
2	**public void write(char [] c, int offset, int len)** Writes a portion of an array of characters starting from offset and with a length of len.
3	**public void write(String s, int offset, int len)** Write a portion of a String starting from offset and with a length of len.

Example:
Following is the example to demonstrate class:
```
import java.io.*;
public class FileRead{
public static void main(String args[])throws IOException{
File file = new File("Hello1.txt");
// creates the file
file.createNewFile();
// creates a FileWriter Object
FileWriter writer = new FileWriter(file);
// Writes the content to the file
writer.write("This\n is\n an\n example\n");
writer.flush();
writer.close();
//Creates a FileReader Object
FileReader fr = new FileReader(file);
```

```
char [] a = new char[50];
fr.read(a); // reads the content to the array
for(char c : a)
System.out.print(c); //prints the characters one by one
fr.close();
}
}
```
This would produce the following result:
```
This
is
an
example
```

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.clas
```