ABOUT THE TUTORIAL

Java Tutorial

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial gives a complete understanding ofJava.

This reference will take you through simple and practical approach while learning Java Programming language.

Audience

This reference has been prepared for the beginners to help them understand the basic to advanced concepts related to Java Programming language.

Prerequisites

Before you start doing practice with various types of examples given in this reference, I'm making an assumption that you are already aware about what is a computer program and what is a computer programming language?

Copyright & Disclaimer Notice

- **Multithreaded**: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

- **Interpreted**:Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and lightweight process.

- **High Performance**: With the use of Just-In-Time compilers, Java enables high performance.

- **Distributed**:Java is designed for the distributed environment of the internet.

- **Dynamic**: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools you will need:

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You also will need the following softwares:

- Linux 7.1 or Windows 95/98/2000/XP operating system.

- Java JDK 5

- Microsoft Notepad or any other text editor

This tutorial will provide the necessary skills to create GUI, networking, and Web applications using Java.

What is Next?

Next chapter will guide you to where you can obtain Java and its documentation. Finally, it instructs you on how to install Java and prepare an environment to develop Java applications.

Popular Java Editors:

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following:

- **Notepad:** On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans:**Is a Java IDE that is open-source and free which can be downloaded fromhttp://www.netbeans.org/index.html.
- **Eclipse:** Is also a Java IDE developed by the eclipse open-source community and can be downloaded from http://www.eclipse.org/.

What is Next?

Next chapter will teach you how to write and run your first Java program and some of the important basic syntaxes in Java needed for developing applications.

Classes in Java:

A class is a blue print from which individual objects are created.

A sample of a class is given below:

```
public class Dog{
String breed;
int age;
String color;

void barking(){
}

void hungry(){
}

void sleeping(){
}
}
```

A class can contain any of the following variable types.

- **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:** Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Below mentioned are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors:

When discussing about classes, one of the most important subtopic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Example of a constructor is given below:

```
public class Puppy{
public Puppy(){
}

public Puppy(String name){
// This constructor has one parameter, name.
}
}
```

CHAPTER

5

Java Basic Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100, byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.

- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s= 10000, short r = -20000

int:

- int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: int a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various types of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce the following result:

```
Puppy age is: 7
```


Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test{
   public void pupAge(){
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce the following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
          ^
1 error
```

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.

CHAPTER 7

Java Modifier Types

Modifiers arekeywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

1. Java Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public

Example:

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version ="1.5.1";

boolean processOrder(){
return true;
}
```

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example:

The following parent class uses protected access control, to allow its child class override*openSpeaker()* method:

```
class AudioPlayer{
protected boolean openSpeaker(Speaker sp){
// implementation details
}
}

class StreamingAudioPlayer{
boolean openSpeaker(Speaker sp){
// implementation details
}
}
```


Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, that's why we used*protected* modifier.

Access Control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

2. Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

```
public class className {
// ...
}
private boolean myFlag;
static final double weeks =9.5;
protected static final int BOXWIDTH =42;
public static void main(String[] arguments){
```

CHAPTER

8

Java Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Example

The following simple example program demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test{

public static void main(String args[]){
int a =10;
int b =20;
int c =25;
int d =25;
System.out.println("a + b = "+(a + b));
System.out.println("a - b = "+(a - b));
System.out.println("a * b = "+(a * b));
System.out.println("b / a = "+(b / a));
System.out.println("b % a = "+(b % a));
System.out.println("c % a = "+(c % a));
System.out.println("a++   = "+(a++));
System.out.println("b--   = "+(a--));
// Check the difference in d++ and ++d
System.out.println("d++   = "+(d++));
System.out.println("++d   = "+(++d));
}
}
```

This would produce the following result:

```
a + b =30
a - b =-10
a * b =200
b / a =2
b % a =0
c % a =5
a++=10
b--=11
d++=25
++d   =27
```


The Relational Operators:

There are following relational operators supported by Java language:

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value	(A >= B) is not true.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>>>>><<	Left to right
Relational	>>= <<=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

What is Next?

Next chapter would explain about loop control in Java programming. The chapter will describe various types of loops and how these loops can be used in Java program development and for what purposes they are being used.

```
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The sine of %.1f degrees is
%.4f%n",degrees,Math.sin(radians));

}
}
```

This produces the following result:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
```

cos()

Description:

The method returns the cosine of the specified double value.

Syntax:

```
double cos(double d)
```

Parameters:

Here is the detail of parameters:

- **d** -- A double data types.

Return Value:

- This method Returns the cosine of the specified double value.

Example:

```
public class Test{

public static void main(String args[]){
double degrees =45.0;
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The cosine of %.1f degrees is %.4f%n",
                        degrees,Math.cos(radians));

}
}
```

This produces the following result:

Here is the detail of parameters:

- **d** -- A double data types

Return Value:

- This method Returns the arccosine of the specified double value.

Example:

```
public class Test{

public static void main(String args[]){
double degrees =45.0;
double radians =Math.toRadians(degrees);

System.out.format("The value of pi is %.4f%n",Math.PI);
System.out.format("The arccosine of %.4f is %.4f degrees %n",
Math.cos(radians),
Math.toDegrees(Math.acos(Math.sin(radians))));


}
}
```

This produces the following result:

```
The value of pi is 3.1416
The arccosine of 0.7071 is 45.0000 degrees
```


atan()

Description:

The method returns the arctangent of the specified double value.

Syntax:

```
double atan(double d)
```

Parameters:

Here is the detail of parameters:

- **d** -- A double data types

Return Value :

- This method Returns the arctangent of the specified double value.

- NA

Return Value:

- This method returns a double

Example:

```
public class Test{

public static void main(String args[]){
System.out.println(Math.random());
System.out.println(Math.random());
}
}
```

This produces the following result:

```
0.16763945061451657
0.400551253762343
```

Note: Above result would vary every time you would call random() method.

What is Next?

In the next section, we will be going through the Character class in Java. You will be learning how to use object Characters and primitive data type char in Java.

The newline character (\n) has been used frequently in this tutorial in System.out.println() statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences:

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example:

If you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes:

```
public class Test{

public static void main(String args[]){
System.out.println("She said \"Hello!\" to me.");
}
}
```


This would produce the following result:

```
She said "Hello!" to me.
```

Character Methods:

Here is the list of the important instance methods that all the subclasses of the Character class implement:

SN	Methods with Description
1	isLetter() Determines whether the specified char value is a letter.
2	isDigit() Determines whether the specified char value is a digit.
3	isWhitespace() Determines whether the specified char value is white space.
4	isUpperCase() Determines whether the specified char value is uppercase.

Example:

```
public class Test{

public static void main(String args[]){
System.out.println(Character.isUpperCase('c'));
System.out.println(Character.isUpperCase('C'));
System.out.println(Character.isUpperCase('\n'));
System.out.println(Character.isUpperCase('\t'));
}
}
```

This produces the following result:

```
false
true
false
false
```

isLowerCase()

Description:

The method determines whether the specified char value is lowercase.

Syntax:

```
boolean isLowerCase(char ch)
```

Parameters:

Here is the detail of parameters:

- **ch** -- Primitive character type

Return Value:

- This method Returns true if passed character is really an lowercase.

Example:

```
public class Test{

public static void main(String args[]){
System.out.println(Character.isLowerCase('c'));
System.out.println(Character.isLowerCase('C'));
System.out.println(Character.isLowerCase('\n'));
System.out.println(Character.isLowerCase('\t'));
}
}
```

14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
15	int hashCode() Returns a hash code for this string.
16	int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
17	int indexOf(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	int indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
19	int indexOf(String str, int fromIndex) 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(int ch) Returns the index within this string of the last occurrence of the specified character.
22	int lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	int lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
24	int lastIndexOf(String str, int fromIndex) 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(String regex) Tells whether or not this string matches the given regular expression.
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
28	boolean regionMatches(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
29	String replace(char oldChar, char newChar) 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(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.

```
Str2=Str1.getBytes("ISO-8859-1");
System.out.println("Returned  Value "+Str2);
}catch(UnsupportedEncodingException e){
System.out.println("Unsupported character set");
}
}
}
```

This produces the following result:

```
Returned  Value [B@192d342
Returned  Value [B@15ff48b
Returned  Value [B@1b90b39
```

byte[] getBytes(String charsetName)

Description:

This method has following two forms:

- **getBytes(String charsetName):** Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
- **getBytes():** Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

Syntax:

Here is the syntax of this method:

```
public byte[] getBytes(String charsetName)
throws UnsupportedEncodingException

or

public byte[] getBytes()
```

Parameters:

Here is the detail of parameters:

- **charsetName** -- the name of a supported charset.

Return Value:

- This method returns the resultant byte array

Example:

```
import java.io.*;

public class Test{
```

- This method returns a hash code value for this object.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");
System.out.println("Hashcode for Str :"+Str.hashCode());
}
}
```

This produces the following result:

```
Hashcode for Str :1186874997
```

int indexOf(int ch)

Description:

This method has following different variants:

- **public int indexOf(int ch):** Returns the index within this string of the first occurrence of the specified character or -1 if the character doesn't occur.
- **public int indexOf(int ch, int fromIndex):** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1 if the character does not occur.
- **int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
- **int indexOf(String str, int fromIndex):** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. If it does not occur, -1 is returned.

Syntax:

Here is the syntax of this method:

```
public int indexOf(int ch )

or

public int indexOf(int ch,int fromIndex)

or

int indexOf(String str)

or

int indexOf(String str,int fromIndex)
```

Parameters:

Here is the detail of parameters:

```
public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.replace('o','T'));

System.out.print("Return Value :");
System.out.println(Str.replace('l','D'));
}
}
```

This produces the following result:

```
Return Value :WelcTme tT TutTrialspTint.cTm
Return Value :WeDcome to TutoriaDspoint.com
```

String replaceAll(String regex, String replacement)

Description:

This method replaces each substring of this string that matches the given regular expression with the given replacement.

Syntax:

Here is the syntax of this method:

```
public String replaceAll(String regex,String replacement)
```

Parameters:

Here is the detail of parameters:

- **regex** -- the regular expression to which this string is to be matched.
- **replacement** -- the string which would replace found expression.

Return Value:

- This method returns the resulting String.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
```

Return Value:

- It returns the array of strings computed by splitting this string around matches of the given regular expression.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome-to-Tutorialspoint.com");

System.out.println("Return Value :");
for(String retval:Str.split("-",2)){
System.out.println(retval);
}
System.out.println("");
System.out.println("Return Value :");
for(String retval:Str.split("-",3)){
System.out.println(retval);
}
System.out.println("");
System.out.println("Return Value :");
for(String retval:Str.split("-",0)){
System.out.println(retval);
}
System.out.println("");
System.out.println("Return Value :");
for(String retval:Str.split("-")){
System.out.println(retval);
}
}
}
```


This produces the following result:

```
Return Value :
Welcome
to-Tutorialspoint.com

Return Value :
Welcome
to
Tutorialspoint.com

Return Value:
Welcome
to
Tutorialspoint.com

Return Value :
Welcome
to
Tutorialspoint.com
```

```
Return Value :true
Return Value :false
Return Value :true
```

CharSequence subSequence(int beginIndex, int endIndex)

Description:

This method returns a new character sequence that is a subsequence of this sequence.

Syntax:

Here is the syntax of this method:

```
public CharSequence subSequence(int beginIndex,int endIndex)
```

Parameters:

Here is the detail of parameters:

- **beginIndex** -- the begin index, inclusive.
- **endIndex** -- the end index, exclusive.

Return Value:

- This method returns the specified subsequence.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.subSequence(0,10));

System.out.print("Return Value :");
System.out.println(Str.subSequence(10,15));
}
}
```

This produces the following result:

```
Return Value :Welcome to
Return Value : Tuto
```

String substring(int beginIndex, int endIndex)

Description:

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex - 1 if second argument is given.

Syntax:

Here is the syntax of this method:

```
public String substring(int beginIndex)

or

public String substring(int beginIndex,int endIndex)
```

Parameters:

Here is the detail of parameters:

- **beginIndex** -- the begin index, inclusive.
- **endIndex** -- the end index, exclusive.

Return Value:

- The specified substring.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.substring(10));

System.out.print("Return Value :");
System.out.println(Str.substring(10,15));
}
}
```

This produces the following result:

```
Return Value : Tutorialspoint.com
Return Value : Tuto
```

char[] toCharArray()

Description:

This method converts this string to a new character array.

Syntax:

Here is the syntax of this method:

```
public char[] toCharArray()
```

Parameters:

Here is the detail of parameters:

- **NA**

Return Value:

- It returns a newly allocated character array, whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.toCharArray());
}
}
```

This produces the following result:

```
Return Value :Welcome to Tutorialspoint.com
```

String toLowerCase()

Description:

This method has two variants. First variant converts all of the characters in this String to lower case using the rules of the given Locale. This is equivalent to calling toLowerCase(Locale.getDefault()).

Second variant takes locale as an argument to be used while converting into lower case.

```
public static void main(String args[]){
String Str=new String("Welcome to Tutorialspoint.com");

System.out.print("Return Value :");
System.out.println(Str.toUpperCase());
}
}
```

This produces the following result:

```
Return Value :WELCOME TO TUTORIALSPOINT.COM
```

String trim()

Description:

This method returns a copy of the string, with leading and trailing whitespace omitted.

Syntax:

Here is the syntax of this method:

```
publicString trim()
```

Parameters:

Here is the detail of parameters:

- **NA**

Return Value:

- It returns a copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

Example:

```
import java.io.*;

public class Test{
public static void main(String args[]){
String Str=new String("   Welcome to Tutorialspoint.com   ");

System.out.print("Return Value :");
System.out.println(Str.trim());
}
}
```

This produces the following result:

```
Return Value :Welcome to Tutorialspoint.com
```

1	**public int start()** Returns the start index of the previous match.
2	**public int start(int group)** Returns the start index of the subsequence captured by the given group during the previous match operation.
3	**public int end()** Returns the offset after the last character matched.
4	**public int end(int group)** Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods:

Study methods review the input string and return a Boolean indicating whether or not the pattern is found:

SN	Methods with Description
1	**public boolean lookingAt()** Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	**public boolean find()** Attempts to find the next subsequence of the input sequence that matches the pattern.
3	**public boolean find(int start** Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	**public boolean matches(** Attempts to match the entire region against the pattern.

Replacement Methods:

Replacement methods are useful methods for replacing text in an input string:

SN	Methods with Description
1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** Implements a non-terminal append-and-replace step.
2	**public StringBuffer appendTail(StringBuffer sb)** Implements a terminal append-and-replace step.
3	**public String replaceAll(String replacement)** Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	**public String replaceFirst(String replacement)** Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	**public static String quoteReplacement(String s)** Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement s in the appendReplacement method of the Matcher class.

The start and end Methods:

Following is the example that counts the number of times the word "cats" appears in the input string:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
   private static final String REGEX ="\\bcat\\b";
   private static final String INPUT ="cat cat cat cattie cat";

public static void main(String args[]){
Pattern p =Pattern.compile(REGEX);
Matcher m = p.matcher(INPUT);// get a matcher object
int count =0;

while(m.find()){
       count++;
System.out.println("Match number "+count);
System.out.println("start(): "+m.start());
System.out.println("end(): "+m.end());
}
}
}
```

This would produce the following result:

```
Match number 1
start():0
end():3
Match number 2
start():4
end():7
Match number 3
start():8
end():11
Match number 4
start():19
end():22
```


You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

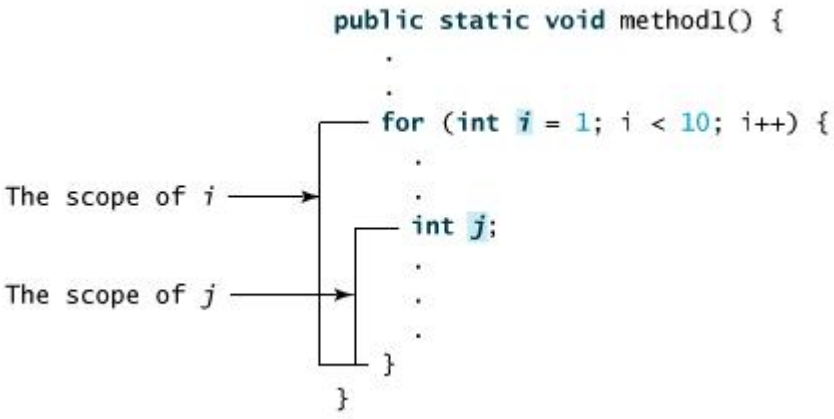
The *matches* and *lookingAt* Methods:

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
```



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
```

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.

	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

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();
      }
```

```
+" with salary "+ salary);
}
public double getSalary()
{
return salary;
}
public void setSalary(double newSalary)
{
if(newSalary >=0.0)
{
  salary = newSalary;
}
}
public double computePay()
{
System.out.println("Computing salary pay for "+ getName());
return salary/52;
}
}
```

Here, we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
public static void main(String[] args)
{
Salary s =new Salary("Mohd Mohtashim","Ambehta, UP",
3,3600.00);
Employee e =new Salary("John Adams","Boston, MA",
2,2400.00);

System.out.println("Call mailCheck using Salary reference --");
  s.mailCheck();
System.out.println("\n Call mailCheck usingEmployee reference--");
  e.mailCheck();
}
}
```


This would produce the following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to JohnAdams with salary 2400.
```

Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.

The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body.

```
public static void main(String[] args){
Map m1 =new HashMap();
  m1.put("Zara","8");
  m1.put("Mahnaz","31");
  m1.put("Ayan","12");
  m1.put("Daisy","14");
System.out.println();
System.out.println(" Map Elements");
System.out.print("\t"+ m1);
}
}
```

This would produce the following result:

```
MapElements
{Mahnaz=31,Ayan=12,Daisy=14,Zara=8}
```

The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.

The specific meaning of keys in regard to hashtables is totally dependent on the usage of the hashtable and the data it contains.

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 reengineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hashtable. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

The Hashtable defines four constructors. The first version is the default constructor:

```
Hashtable()
```

The second version creates a hashtable that has an initial size specified by size:

```
Hashtable(int size)
```

The third version creates a hashtable that has an initial size specified by size and a fill ratio specified by fillRatio.

This ratio must be between 0.0 and 1.0, and it determines how full the hashtable can be before it is resized upward.

```
Hashtable(int size,float fillRatio)
```

The fourth version creates a hashtable that is initialized with the elements in m.

The capacity of the hashtable is set to twice the number of elements in m. The default load factor of 0.75 is used.

```
Hashtable(Map m)
```

Apart from the methods defined by Map interface, Hashtable defines the following methods:

```
System.out.println();

// look for state not in list -- specify default
  str = capitals.getProperty("Florida","Not Found");
System.out.println("The capital of Florida is "+ str +".");
}
}
```

This would produce the following result:

```
The capital of Missouri is JeffersonCity.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is NotFound.
```

9	TreeSet Implements a set stored in a tree. Extends AbstractSet.
10	**AbstractMap** Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents.

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util have been discussed in previous tutorial:

SN	Classes with Description
1	Vector This implements a dynamic array. It's similar to ArrayList, but with some differences.
2	Stack Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3	Dictionary Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4	Hashtable Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5	Properties Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6	BitSet A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

The Collection Algorithms:

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

	Returns the next element. Throws NoSuchElementException if there is not a next element.
3	**void remove( )** Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

The Methods Declared by ListIterator:

SN	Methods with Description
1	**void add(Object obj)** Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	**boolean hasNext( )** Returns true if there is a next element. Otherwise, returns false.
3	**boolean hasPrevious( )** Returns true if there is a previous element. Otherwise, returns false.
4	**Object next( )** Returns the next element. A NoSuchElementException is thrown if there is not a next element.
5	**int nextIndex( )** Returns the index of the next element. If there is not a next element, returns the size of the list.
6	**Object previous( )** Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
7	**int previousIndex( )** Returns the index of the previous element. If there is not a previous element, returns -1.
8	**void remove( )** Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9	**void set(Object obj)** Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Example:

Here is an example demonstrating both Iterator and ListIterator. It uses an ArrayList object, but the general principles apply to any type of collection.

Of course, ListIterator is available only to those collections that implement the List interface.

```
import java.util.*;

public class IteratorDemo {

   public static void main(String args[]) {
      // Create an array list
      ArrayList al = new ArrayList();
      // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
```

CHAPTER

30

Java Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:


```
public final void writeObject(Object x)throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject()throws IOException,
ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

```
public class Employeeimplements java.io.Serializable
{
public String name;
public String address;
public transient int SSN;
public int number;
public void mailCheck()
{
System.out.println("Mailing a check to "+ name+" "+ address);
}
```

CHAPTER

31

Java Networking

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connectionless protocol that allows for packets of data to be transmitted between applications.

This tutorial gives good understanding on the following two subjects:

- **Socket Programming**: This is most widely used concept in Networking and it has been explained in very detail.
- **URL Processing**: This would be covered separately. Click here to learn about URL Processing in Java language.

Url Processing

URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.

This section shows you how to write Java programs that communicate with a URL. A URL can be broken down into parts, as follows:

```
protocol://host:port/path?query#ref
```

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.

The following is a URL to a Web page whose protocol is HTTP:

```
http://www.amrood.com/index.htm?language=en#j2se
```

Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

URL Class Methods:

The **java.net.URL** class represents a URL and has complete set of methods to manipulate URL in Java.

The URL class has several constructors for creating URLs, including the following:

SN	Methods with Description
1	**public URL(String protocol, String host, int port, String file) throws MalformedURLException.** Creates a URL by putting together the given parts.
2	**public URL(String protocol, String host, String file) throws MalformedURLException** Identical to the previous constructor, except that the default port for the given protocol is used.
3	**public URL(String url) throws MalformedURLException** Creates a URL from the given String
4	**public URL(URL context, String url) throws MalformedURLException** Creates a URL by parsing the together the URL and String arguments

The URL class contains many methods for accessing the various parts of the URL being represented.

Some of the methods in the URL class include the following:

SN	Methods with Description
1	**public String getPath()** Returns the path of the URL.
2	**public String getQuery()** Returns the query part of the URL.
3	**public String getAuthority()** Returns the authority of the URL.
4	**public int getPort()** Returns the port of the URL.
5	**public int getDefaultPort()** Returns the default port for the protocol of the URL.
6	**public String getProtocol()** Returns the protocol of the URL.
7	**public String getHost()** Returns the host of the URL.
8	**public String getHost()** Returns the host of the URL.
9	**public String getFile()** Returns the filename of the URL.
10	**public String getRef()** Returns the reference part of the URL.
11	**public URLConnection openConnection() throws IOException** Opens a connection to the URL, allowing a client to communicate with the resource.

```
{
public static void main(String[] args)
{
try
{
URL url =new URL(args[0]);
URLConnection urlConnection = url.openConnection();
HttpURLConnection connection =null;
if(urlConnection instanceof HttpURLConnection)
{
   connection =(HttpURLConnection) urlConnection;
}
else
{
System.out.println("Please enter an HTTP URL.");
return;
}
BufferedReader in=new BufferedReader(
new InputStreamReader(connection.getInputStream()));
String urlString ="";
String current;
while((current =in.readLine())!=null)
{
urlString += current;
}
System.out.println(urlString);
}catch(IOException e)
{
   e.printStackTrace();
}
}
}
```


A sample run of the thid program would produce the following result:

```
$ java URLConnDemo http://www.amrood.com

.....a complete HTML content of home page of amrood.com.....
```

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.

- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.

- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

SN	Methods with Description
1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on
4	**public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

SN	Methods with Description
1	**public int getLocalPort()** Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	**public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.
3	**public void setSoTimeout(int timeout)** Sets the time-out value for how long the server socket waits for a client during the accept().

```
// Part two is attachment
messageBodyPart =new MimeBodyPart();
String filename ="file.txt";
DataSource source =new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Send the complete message parts
message.setContent(multipart );

// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
}catch(MessagingException mex){
mex.printStackTrace();
}
}
}
```

Compile and run this program to send an HTML e-mail:

```
$ java SendFileEmail
Sent message successfully....
```


User Authentication Part:

If it is required to provide user ID and Password to the e-mail server for authentication purpose, then you can set these properties as follows:

```
props.setProperty("mail.user","myuser");
props.setProperty("mail.password","mypwd");
```

Rest of the e-mail sending mechanism would remain as explained above.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                      + " guesses " + guess);
          counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                      + " in " + counter + " guesses.**");
   }
}
```


Following is the main program which makes use of above defined classes:

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
```

```
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(bye);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

          thread4.start();
      System.out.println("main() is ending...");
   }
}
```


This would produce the following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```

Major Java Multithreading Concepts:

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

- What is thread synchronization?
- Handling threads inter communication
- Handling thread deadlock
- Major thread operations

```
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
   void suspend() {
      suspended = true;
   }
   synchronized void resume() {
      suspended = false;
       notify();
   }
}

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

      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();

      try {
         Thread.sleep(1000);
         R1.suspend();
         System.out.println("Suspending first thread");
         Thread.sleep(1000);
         R1.resume();
         System.out.println("Resuming first thread");
         R2.suspend();
         System.out.println("Suspending thread Two");
         Thread.sleep(1000);
         R2.resume();
         System.out.println("Resuming thread Two");
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      }
      try {
         System.out.println("Waiting for threads to finish.");
         R1.t.join();
         R2.t.join();
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      }
      System.out.println("Main thread exiting.");
   }
}
```


Here is the output produced by the above program:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 10
Running Thread-2
Thread: Thread-2, 10
Thread: Thread-1, 9
```

```
Thread: Thread-2, 9
Thread: Thread-1, 8
Thread: Thread-2, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Suspending thread Two
Thread: Thread-1, 6
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```

```
clip.loop();
}
}
publicvoid stop()
{
if(clip !=null)
{
   clip.stop();
}
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<appletcode="ImageDemo.class"width="0"height="0">
<paramname="audio"value="test.wav">
</applet>
<hr>
</html>
```

You can use your test.wav at your PC to test the above example.

		include I exclude
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods	@serialData data-description
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description
{@value}	When {@value} is used in the doc comment of a static field, it displays the value of that constant:	{@value package.class#field}
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text

Example:

Following program uses few of the important tags available for documentation comments. You can make use of other tags based on your requirements.

The documentation about the AddNum class will be produced in HTML file AddNum.html but same time a master file with a name index.html will also be created.


```
import java.io.*;

/**
* <h1>Add Two Numbers!</h1>
* The AddNum program implements an application that
* simply adds two given integer numbers and Prints
* the output on the screen.
* <p>
* <b>Note:</b> Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class AddNum {
   /**
   * This method is used to add two integers. This is
   * a the simplest form of a class method, just to
   * show the usage of various javadoc Tags.
   * @param numA This is the first paramter to addNum method
   * @param numB  This is the second parameter to addNum method
   * @return int This returns sum of numA and numB.
   */
   public int addNum(int numA, int numB) {
      return numA + numB;
   }

   /**
   * This is the main method which makes use of addNum method.
   * @param args Unused.
   * @return Nothing.
```

CHAPTER

36

Java Library Classes

This tutorial would cover package **java.lang,** which provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Here is the list of classes of ackage **java.lang**. These classes are very important to know for a Java programmer. Click a class link to know more detail about that class. For a further drill, you can refer standard Java documentation.

SN	Methods with Description
1	Boolean Boolean
2	Byte The Byte class wraps a value of primitive type byte in an object.
3	Character The Character class wraps a value of the primitive type char in an object.
4	Class Instances of the class Class represent classes and interfaces in a running Java application.
5	ClassLoader A class loader is an object that is responsible for loading classes.
6	Compiler The Compiler class is provided to support Java-to-native-code compilers and related services.
7	Double The Double class wraps a value of the primitive type double in an object.
8	Float The Float class wraps a value of primitive type float in an object.
9	Integer The Integer class wraps a value of the primitive type int in an object.
10	Long The Long class wraps a value of the primitive type long in an object.
11	Math The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.