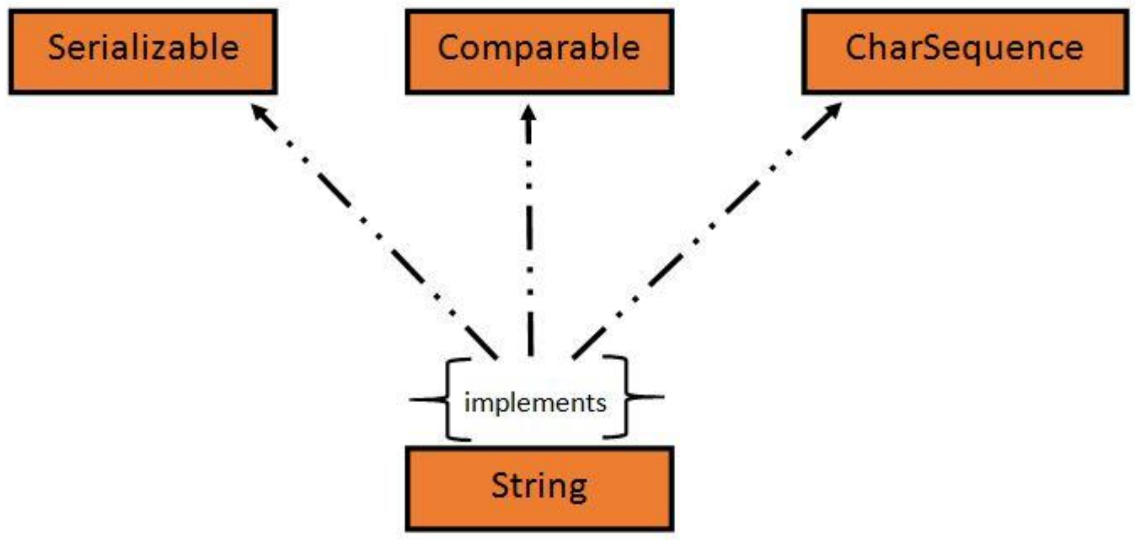
UNIT-5

Java String Handling

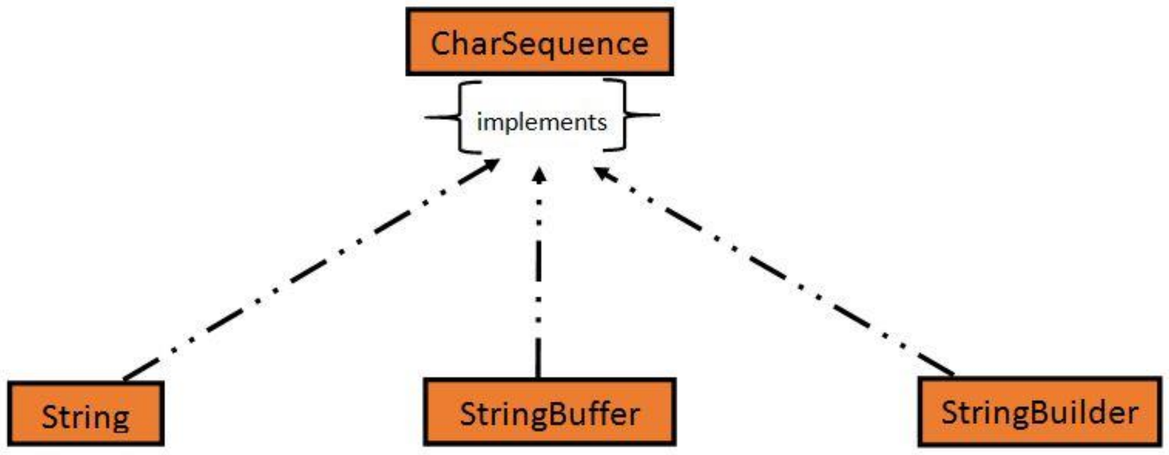
String is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.



In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.



Class string:

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

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

Creating Strings

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

String greeting = "Hello world!";

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

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

```
public class StringDemo {

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

This will produce the following result −

Output

```
hello.
```

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

String Length

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

The following program is an example of **length()**, method String class.

Example

```
public class StringDemo {

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

This will produce the following result −

Output

```
String Length is : 17
```

Concatenating Strings

The String class includes a method for concatenating two strings −

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

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

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

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

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

which results in −

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

Let us look at the following example −

Example

```
public class StringDemo {

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

This will produce the following result −

Output

```
Dot saw I was Tod
```

Creating Format Strings

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

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

Example

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

You can write −

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

String Methods

Here is the list of methods supported by String class –

Sr.No.	**Method & Description**
1	char charAt(int index) Returns the character at the specified index.
2	int compareTo(Object o) Compares this String to another Object.

3	int compareTo(String anotherString) Compares two strings lexicographically.
4	int compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
5	String concat(String str) Concatenates the specified string to the end of this string.
6	boolean contentEquals(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	static String copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.
8	static String copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
9	boolean endsWith(String suffix) Tests if this string ends with the specified suffix.
10	boolean equals(Object anObject) Compares this string to the specified object.
11	boolean equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
12	byte[] getBytes() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

13	byte[] getBytes(String charsetName) Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	void getChars(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.
32	String[] split(String regex) Splits this string around matches of the given regular expression.

33	String[] split(String regex, int limit) Splits this string around matches of the given regular expression.
34	boolean startsWith(String prefix) Tests if this string starts with the specified prefix.
35	boolean startsWith(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
36	CharSequence subSequence(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
37	String substring(int beginIndex) Returns a new string that is a substring of this string.
38	String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
39	char[] toCharArray() Converts this string to a new character array.
40	String toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale.
41	String toLowerCase(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
42	String toString() This object (which is already a string!) is itself returned.

43	String toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale.
44	String toUpperCase(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
45	String trim() Returns a copy of the string, with leading and trailing whitespace omitted.
46	static String valueOf(primitive data type x) Returns the string representation of the passed data type argument.

Character Extraction:

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Several are examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( ):

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

char ch;

ch = "abc".charAt(1);

assigns the value **b** to **ch**.

getChars( ):

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

The following program demonstrates **getChars( )**:

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This   is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new   char[end - start];
s.getChars(start, end, buf, 0);
 System.out.println(buf);
}
}
```

getBytes( ):

There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( ):

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

String Comparison:

The **String** class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form: boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals( )** and **equalsIgnoreCase( )**:

```
// Demonstrate equals() and equalsIgnoreCase().

class equalsDemo {


public static void main(String args[]) { String s1 = "Hello";

String s2 = "Hello"; String s3 = "Good-bye"; String s4 = "HELLO";

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));

System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));

System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));

}

}
```

The output from the program is shown here:

```
Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true
```

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int *startIndex*, String *str2*,

int *str2StartIndex*, int *numChars*)

boolean regionMatches(boolean *ignoreCase*,

int *startIndex*, String *str2*,

int *str2StartIndex*, int *numChars*

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )

String defines two methods that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*) boolean endsWith(String *str*)

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

"Foobar".endsWith("bar")

and

"Foobar".startsWith("Foo")

are both **true**.

A second form of **startsWith( )**, shown here, lets you specify a starting point: boolean startsWith(String *str*, int *startIndex*)

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

"Foobar".startsWith("bar", 3)

returns **true**.

equals( ) Versus ==

It is important to understand that the **equals( )** method and the **==** operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

// equals() vs ==

```
class EqualsNotEqualTo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = new String(s1);

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));

}

}
```

The variable **s1** refers to the **String** instance created by **"Hello"**. The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true

Hello == Hello -> false
```

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The method **compareTo( )** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements. It has this general form:

int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here

Here is a sample program that sorts an array of strings. The program uses **compareTo( )** to determine sort ordering for a bubble sort:

```
 // A bubble sort for Strings.

class SortString {

static String arr[] = {

"Now", "is", "the", "time", "for", "all", "good", "men", "to", "come", "to", "the", "aid", "of", "their", "country"

};
public static void main(String args[]) { for(int j = 0; j < arr.length; j++) {

for(int i = j + 1; i < arr.length; i++) { if(arr[i].compareTo(arr[j]) < 0) {

String t = arr[j];

arr[j] = arr[i]; arr[i] = t;

}

}

System.out.println(arr[j]);

}

}

}
```

The output of this program is the list of words:

```
Now aid all come country for good

is men of the the their time to

to
```

As you can see from the output of this example, **compareTo( )** takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

int compareToIgnoreCase(String *str*)

This method returns the same results as **compareTo( )**, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

indexOf( )	Searches for the first occurrence of a character or substring.
lastIndexOf( )	Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

To search for the first occurrence of a character, use int indexOf(int *ch*)

To search for the last occurrence of a character, use int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String *str*) int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*) int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*) int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " + "to come to the aid of their country.";
System.out.println(s); System.out.println("indexOf(t) = " +
s.indexOf('t')); System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t')); System.out.println("indexOf(the) = " +
s.indexOf("the")); System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the")); System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10)); System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60)); System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10)); System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
}
}
```

To search for the last occurrence of a character, use int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use int indexOf(String *str*) int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*) int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*) int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().

class indexOfDemo {

public static void main(String args[]) {


String s = "Now is the time for all good men " + "to come to the aid of their country."; System.out.println(s);

 System.out.println("indexOf(t) = " +s.indexOf('t')); System.out.println("lastIndexOf(t) = "
+ s.lastIndexOf('t')); System.out.println("indexOf(the) = " +s.indexOf("the"));
System.out.println("lastIndexOf(the) = " +s.lastIndexOf("the")); System.out.println("indexOf(t, 10) = "
+s.indexOf('t', 10)); System.out.println("lastIndexOf(t, 60) = " +s.lastIndexOf('t', 60));

 System.out.println("indexOf(the, 10) = " +s.indexOf("the", 10));

System.out.println("lastIndexOf(the, 60) = " +s.lastIndexOf("the", 60));

}

}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country. indexOf(t) = 7

lastIndexOf(t) = 65

indexOf(the) = 7

lastIndexOf(the) = 55

indexOf(t, 10) = 11

lastIndexOf(t, 60) = 55

indexOf(the, 10) = 44
```

lastIndexOf(the, 60) = 55

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
 public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was"; String result = ""; int i;
do { // replace all matching substrings
System.out.println(org);
i = org.indexOf(search); if(i != -1)
 {
 result = org.substring(0, i); result = result + sub;result = result + org.substring(i + search.length()); org = result;
}
} while(i != -1);
}
}
```

The output from this program is shown here:

```
This is a test. This is, too.
 Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

You can concatenate two strings using **concat( )**, shown here: String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( ) Here is an example:

```
String s = "     Hello World     ".trim();
```

This puts the string "Hello World" into **s**.

The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state’s capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;
class UseTrim {
public static void main(String args[]) throws IOException
{

// create a BufferedReader using System.
in BufferedReader br = new
BufferedReader(new InputStreamReader(System.in)); String str;
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do {
str = br.readLine();
str = str.trim(); // remove whitespace
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
else if(str.equals("Missouri"))
 System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
System.out.println("Capital is Olympia.");
// ...
} while(!str.equals("stop"));
}
}
```

Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java’s built-in types so that each type can be converted properly into a string. **valueOf( )** is also overloaded for type

Object, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

static String valueOf(double *num*) static String valueOf(long *num*) static String valueOf(Object *ob*) static String valueOf(char *chars*[ ])

valueOf( ) is called when a string representation of some other type of data is needed—for example, during concatenation operations. You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass to **valueOf( )** will return the result of a call to the object’s **toString( )** method. In fact, you could just call **toString( )** directly and get the same result.

For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type. For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array. There is a special version of **valueOf( )** that allows you to specify a subset of a **char** array. It has this general form:

static String valueOf(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *chars* is the array that holds the characters, *startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( ) StringBuffer(int *size*) StringBuffer(String *str*)

StringBuffer(CharSequence *chars*)

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

length( ) and capacity( )

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity can be found through the **capacity( )** method. They have the following general forms:

int length( ) int capacity( )

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
 System.out.println("length = " + sb.length()); System.out.println("capacity = " + sb.capacity());
}
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello length = 5 capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity( )

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a

StringBuffer. **ensureCapacity( )** has this general form:

void ensureCapacity(int *minCapacity*)

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength( )

To set the length of the string within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

void setLength(int *len*)

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength( )** with a value less than the current value returned by **length( )**, then the characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength( )** to shorten a **StringBuffer**.

charAt( ) and setCharAt( )

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

char charAt(int *where*)

void setCharAt(int *where*, char *ch*)

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt( )** and **setCharAt( )**:

```
// Demonstrate charAt() and setCharAt().

class setCharAtDemo {

public static void main(String args[]) {

StringBuffer sb = new StringBuffer("Hello"); System.out.println("buffer before = " + sb);


System.out.println("charAt(1) before = " + sb.charAt(1));

sb.setCharAt(1, 'i'); sb.setLength(2);

System.out.println("buffer after = " + sb); System.out.println("charAt(1) after = " + sb.charAt(1));

}

}
```

Here is the output generated by this program:

```
buffer before = Hello

charAt(1) before = e
```

```
buffer after = Hi

charAt(1) after = i
```

getChars( )

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append( )

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*) StringBuffer append(int *num*) StringBuffer append(Object *obj*)

The string representation of each parameter is obtained, often by calling **String.valueOf( )**. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append( )**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().

class appendDemo {

public static void main(String args[]) {

String s;

int a = 42;

StringBuffer sb = new StringBuffer(40);

s = sb.append("a = ").append(a).append("!").toString(); System.out.println(s);

}

}
```

The output of this example is shown here:

```
a = 42!
```

insert( )

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **String**s, **Object**s, and **CharSequence**s. Like **append( )**, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*) StringBuffer insert(int *index*, char *ch*) StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking

StringBuffer object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) { StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like "); System.out.println(sb);
}
}
```

The output of this example is shown here:

```
I like Java!
```

reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here: StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) { StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

Here is the output produced by the program:

```
abcdef fedcba
```

delete( ) and deleteCharAt( )

You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*) StringBuffer deleteCharAt(int *loc*)

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete( )** and **deleteCharAt( )** methods:

```
// Demonstrate delete() and deleteCharAt()

class deleteDemo {

public static void main(String args[]) {

StringBuffer sb = new StringBuffer("This is a test.");

sb.delete(4, 7); System.out.println("After delete: " + sb);

sb.deleteCharAt(0);

System.out.println("After deleteCharAt: " + sb);

}

}
```

The following output is produced:

```
After delete: This a test.

After deleteCharAt: his a test.
```

replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()

class replaceDemo {

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

```
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was"); System.out.println("After replace: " + sb);
}
}
```

Here is the output:

```
After replace: This was a test.
```

substring( )

You can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

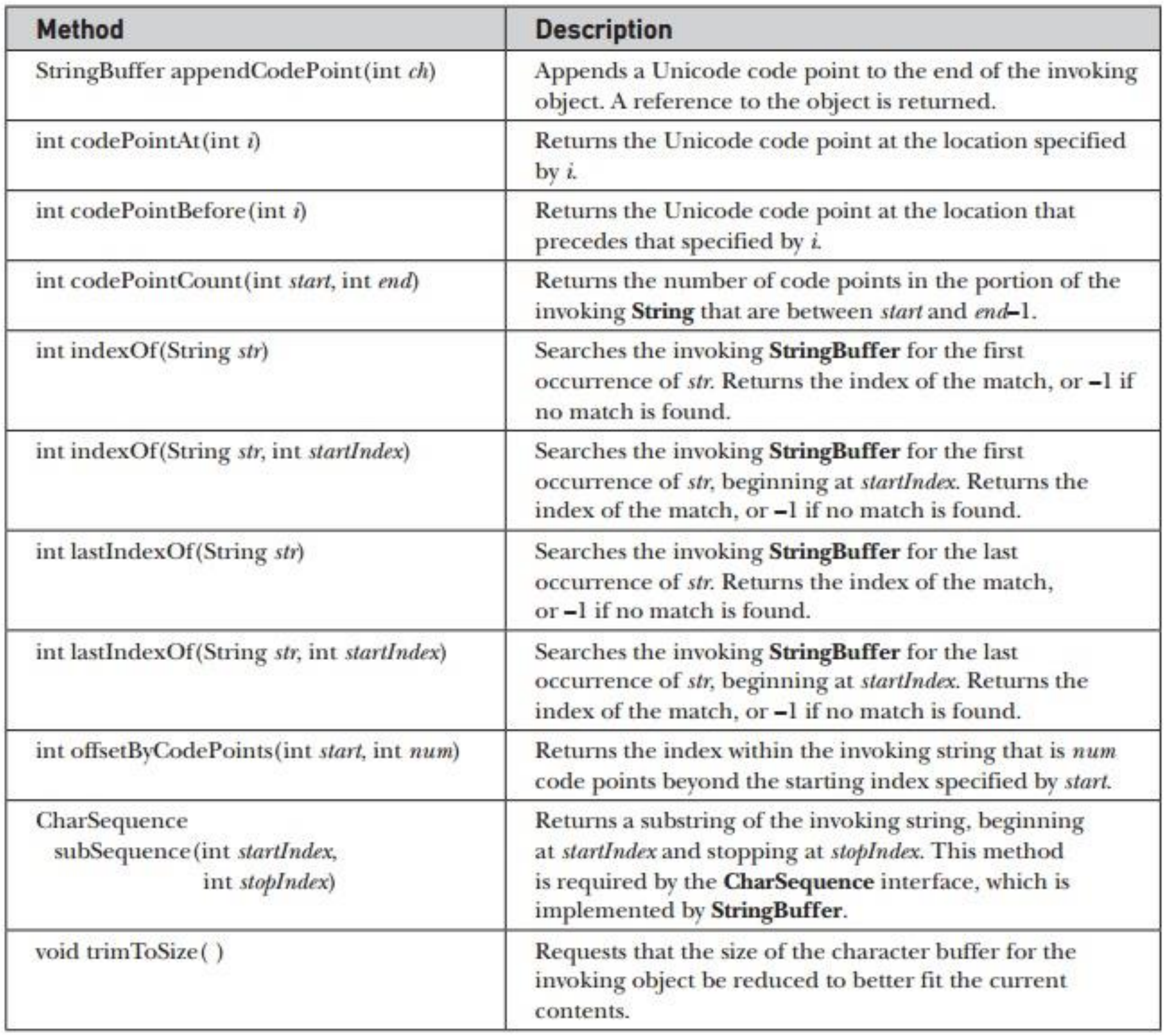
String substring(int *startIndex*)

String substring(int *startIndex*, int *endIndex*)

The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object. The second form returns the substring that starts at startIndex and runs through endIndex–1. These methods work just like those defined for String that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies several others, including those summarized in the following table:



Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **StringBuffer**.
void trimToSize( )	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

```
class IndexOfDemo {
public static void main(String args[]) { StringBuffer sb = new StringBuffer("one two one"); int i;
i = sb.indexOf("one"); System.out.println("First index: " + i);
i = sb.lastIndexOf("one"); System.out.println("Last index: " + i);
}
}
```

The output is shown here:

```
 First index: 0    Last index:
```

StringBuilder

Introduced by JDK 5, **StringBuilder** is a relatively recent addition to Java's string handling capabilities. **StringBuilder** is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

Multi-threading:

Introduction

Unlike many other programming languages, java supports multithreaded programming. The multithreaded program contains two or more ***parts*** that can run concurrently. These parts are called –**Threads**‖**.** Each Thread defines separate path of execution. Thus, Multithreading is specialized form of **Multitasking**.

However, there are two distinct types of multitasking: **process based and thread-based**. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A ***process*** is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the ***scheduler***.

In a *thread-based* multitasking environment, ***the thread is the smallest unit of dispatchable code***. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the **"big picture,"** and thread-based multitasking handles the **details**. Multitasking threads require less overhead than multitasking processes. Processes are heavy weight tasks that require their own separate address space.

Difference between Multiprocessing and Multithreading

Process-Based Multitasking	Thread-Based Multitasking
This deals with "Big Picture"	This deals with Details
These are Heavyweight tasks	These are Lightweight tasks
Inter-process communication is expensive and limited	Inter-Thread communication is inexpensive.
Context switching from one process to another is costly in terms of memory	Context switching is low cost in terms of memory, because they run on the same address space
This is not under the control of Java. Controlled by Operating System.	This is controlled by Java

Life Cycle of a Thread

During the life time of the thread, there are many states it can enter. They include the following:

- Newborn state
- Runnable State
- Running State
- Blocked state
- Dead state

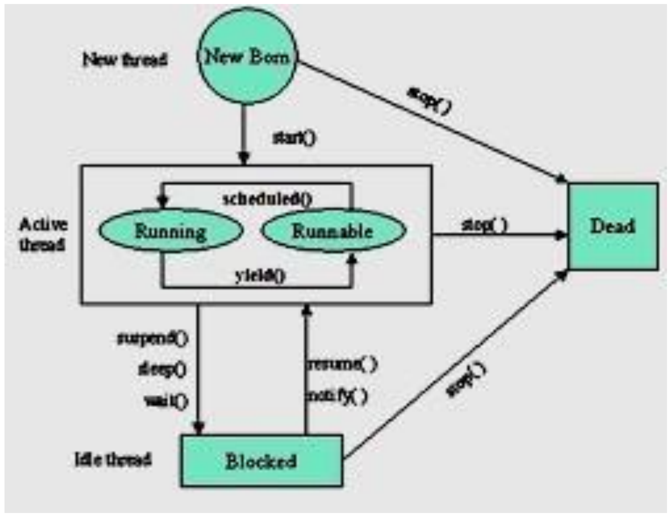


Fig 1: Life Cycle of Thread

A thread can always in any one of the five states. It can move from one state to other via variety of ways as shown in the fig.

Newborn State: When we create a thread it is said to be in the new born state. At this state we can do the following:

- Schedule it for running using the start() method.
- Kill it using stop() method.

Runnable State: A runnable state means that a thread is ready for execution and waiting for the availability of the processor. That is the thread has joined the queue of the threads for execution. If all the threads have equal priority, then they are given time slots for execution in the round rabin fashion, first-come-first-serve manner. The thread that relinquishes the control will join the queue at the end and again waits for its turn. This is known as time ***slicing***.

Running state: Running state means that the processor has given its time to the thread for it execution. The thread runs until it ***relinquishes*** the control or it is preempted by the other higher priority thread. As shown in the Fig 1. a running thread can be preempted using the suspen(), or wait(), or sleep() methods.

Blocked state: A thread is said to be in the blocked state when it is prevented from entering into runnable state and subsequently the running state.

Dead state: Every thread has a life cycle. A running thread ends its life when it has completed execution. It is a natural death. However we also can kill the thread by sending the stop() message to it at any time.

The methods of the Thread class are as follow:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() methodon the thread.

3. **public void sleep(long miliseconds):** Causes the currently executing thread tosleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specifiedmiliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executingthread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarilypause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has beeninterrupted.

Multithreading in Java

Every program that we have been writing has at least one thread, that is, the "**main**" thread. Whenever a program starts executing, the JVM is responsible for creating the main thread and calling "main()" method. Along with this main thread, some other threads are also running to carryout the tasks such as "**finalization**" and "**garbage collection**". The thread can either die naturally or be forced to die.

- Thread dies naturally when it exits from the "run()" method.
- Thread can be forced to die by calling "interrupt()" method.

java.lang.Thread package

Creation of Thread in java is very simple task. There is a class called "Thread", which belongs to the "java.lang.Thread" package. This package contains one interface also called "**Runnable**". Both these contain a common method called "**run()**" which is the heart of the thread. The run() method would have the following syntax:

Syntax:

```
public void run()

{

        //statement for implementing the thread
```

Thread Constructors:

- Thread ()-without arguments, default constructor
- Thread(String str)- Thread contains name given as argument

The Main Thread

Every java program has a thread called "main" thread. When the program execution starts, the JVM creates "main" Thread and calls the "main()" method from within that thread. Along with this JVM also creates other threads for the purpose of the Housekeeping task such as "garbage" collection. The "main" thread Spawns the other Threads. These spawned threads are called "Child Threads". The main thread is always is the last thread to finish execution. We, as Programmer can also take control of the main thread, using the method "**currentThread()**". The main thread can be controlled by this method. We can also change the name of the Thread using the method "**setName(String name)**".

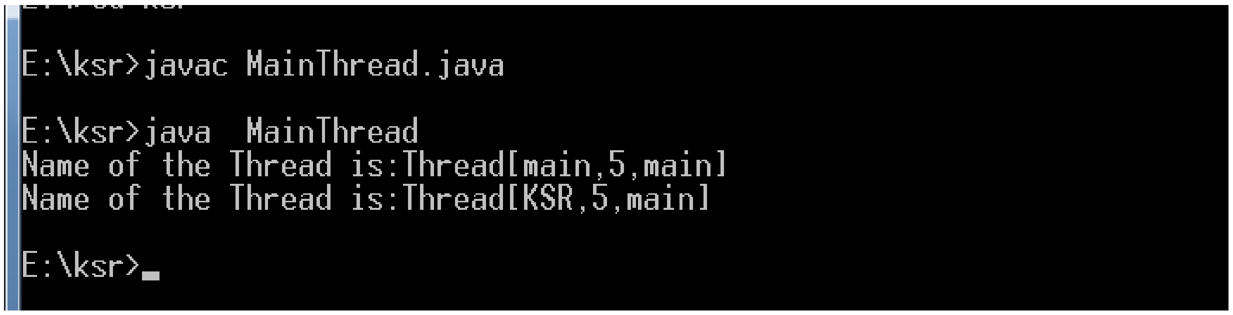
Example Program:

MainThread.java

```
class MainThread
{
        public static void main(String args[])
        {
        Thread t=Thread.currentThread();
        System.out.println("Name of the Thread is:"+t);
        t.setName("KSR");
        System.out.println("Name of the Thread is:"+t);
        }
}
```

Note: If we want name of the Thread alone, we can write t.getName().

Output:



```
E:\ksr>javac MainThread.java

E:\ksr>java  MainThread
Name of the Thread is:Thread[main,5,main]
Name of the Thread is:Thread[KSR,5,main]

E:\ksr>_
```

Creation of Threads

Creating the threads in the Java is simple. The threads can be implemented in the form of object that contains a method "**run()**". The "**run()**" method is the heart and soul of any thread. It makes up the entire body of the thread and is the only method in which the thread behavior can be implemented. There are two ways to create thread.

1. Declare a class that **extends** the **Thread** class and override the **run()** method.
2. Declare a class that implements the **Runnable** interface which contains the **run()** method

1. *Creating Thread using The Thread Class*

We can make our thread by extending the Thread class of java.lang.Thread class. Thisgives us access to all the methods of the Thread. It includes the following steps:

I. Declare the class as extending the Thread class.
II. Override the "**run()**" method that is responsible for running the thread.
III. Create a thread and call the "**start()**" method to instantiate the Thread Execution.

Declaring the class

TheThread class can be declared as follows:

```
class MyThread extends Thread
{
        -------------------------
        ------------------------
        ------------------------
        ------------------------
}
```

Overriding the method run()

The run() is the method of the Thread. We can override this as follows:

```
public void run()
{
        -----------------
        -----------------
        -----------------
```

Starting the new Thread

To actually to create and run an instance of the thread class, we must write the following:

```
MyThread a=new MyThread(); // creating the Thread
a.start();                          // Starting the Thread
```

Example program:

```
import java.io.*; import java.lang.*;
```

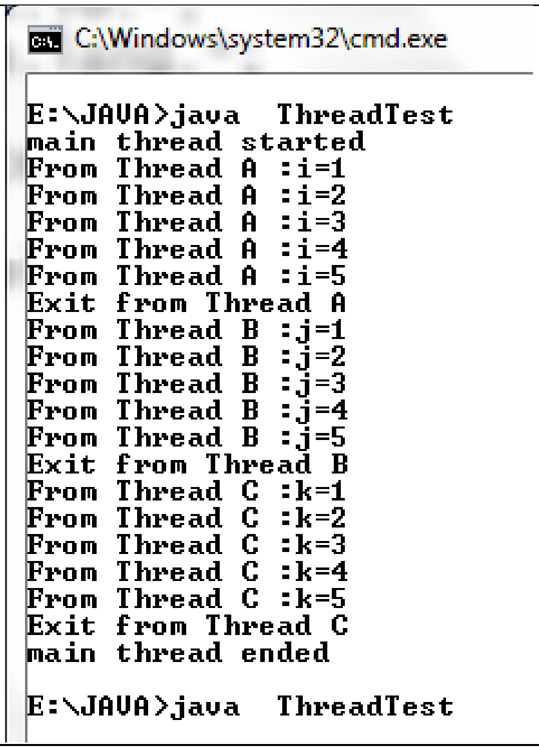
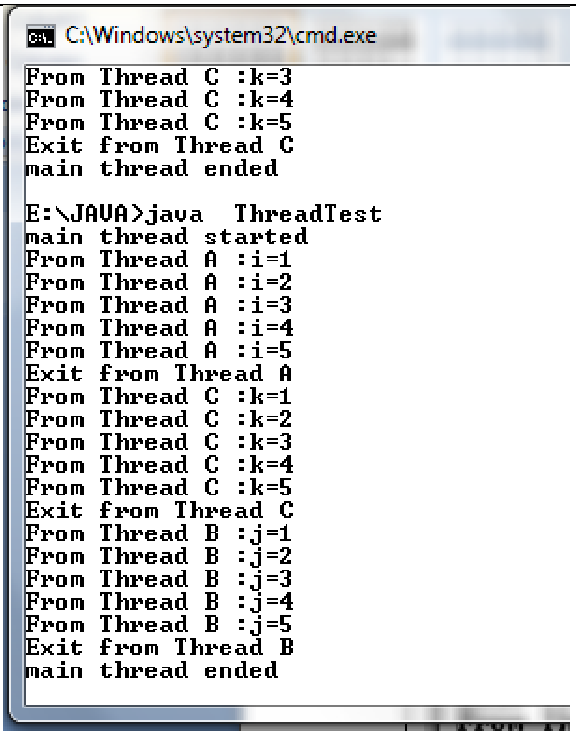
<u>ThreadTest.java</u>

```
class A extends Thread

{

        public void run()

        {

                for(int i=1;i<=5;i++)

                {

                        System.out.println("From Thread A :i="+i);

                }

                System.out.println("Exit from Thread A");

        }

}

class B extends Thread

{

        public void run()

        {

                for(int j=1;j<=5;j++)

                {

                        System.out.println("From Thread B :j="+j);

                }
```

```
                    System.out.println("Exit from Thread B");
          }
}
class C extends Thread
{
          public void run()
          {
                    for(int k=1;k<=5;k++)
                    {
                              System.out.println("From Thread C :k="+k);
                    }
                    System.out.println("Exit from Thread C");
          }
}
class ThreadTest
{
          public static void main(String args[]) {
System.out.println("main thread started");A a=new A();
```

```
                    a.start();

                    B b=new B();
                    b.start();

                    C c=new C();
                    c.start();
```

output: First Run

First Run	Second Run
	

C:\Windows\system32\cmd.exe

```
E:\JAVA>java  ThreadTest
main thread started
From Thread A :i=1
From Thread A :i=2
From Thread A :i=3
From Thread A :i=4
From Thread A :i=5
Exit from Thread A
From Thread B :j=1
From Thread B :j=2
From Thread B :j=3
From Thread B :j=4
From Thread B :j=5
Exit from Thread B
From Thread C :k=1
From Thread C :k=2
From Thread C :k=3
From Thread C :k=4
From Thread C :k=5
Exit from Thread C
main thread ended

E:\JAVA>java  ThreadTest
```

C:\Windows\system32\cmd.exe

```
From Thread C :k=3
From Thread C :k=4
From Thread C :k=5
Exit from Thread C
main thread ended

E:\JAVA>java  ThreadTest
main thread started
From Thread A :i=1
From Thread A :i=2
From Thread A :i=3
From Thread A :i=4
From Thread A :i=5
Exit from Thread A
From Thread C :k=1
From Thread C :k=2
From Thread C :k=3
From Thread C :k=4
From Thread C :k=5
Exit from Thread C
From Thread B :j=1
From Thread B :j=2
From Thread B :j=3
From Thread B :j=4
From Thread B :j=5
Exit from Thread B
main thread ended
```

Note: If you observe here, first and second Runs produce different Results. This depends on the Processor availability, speed of the processor, Scheduling algorithms and priority of the threads. If you run this in two different systems they produce different outputs.

2. *Creating the Thread using Runnable Interface*

The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

I. Declare a class as implementing the **Runnable** interface
II. Implement the **run()** method
III. Create a **Thread** by defining an object that is instantiated from this "runnable" class as the target of the thread
IV. Call the thread's **start()** method to run the thread.

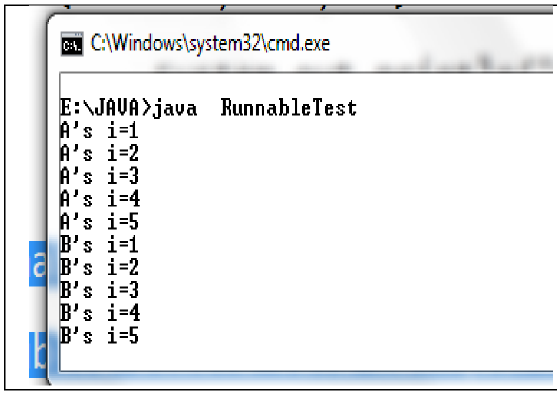
Example program:

Runnable.java

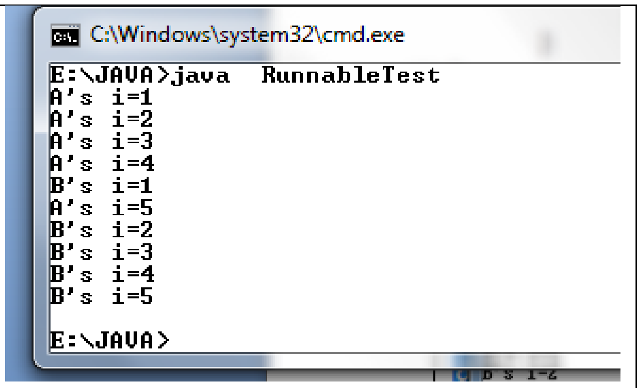
Output: Threads A and B execution by running the above program two times. (You may see a different sequence of Output, every time you run this program)

```
class A implements Runnable
{
        public void run()
        {
        for(int i=1;i<=5;i++)
        {
                System.out.println("A's i="+i);
        }
        }
}
class B implements Runnable
{
        public void run()
        {
        for(int i=1;i<=5;i++)
        {
                System.out.println("B's i="+i);
        }
        }
}
class RunnableTest
{       public static void main(String args[])
        {
                A a=new
                A();B
                b=new B();
                Thread t1=new Thread(a); //providing runnable object as argument
                Thread t2=new Thread(b);
                t1.start();
                t2.start();
        }
}
```

First Run	Second Run



```
E:\JAVA>java  RunnableTest
A's i=1
A's i=2
A's i=3
A's i=4
A's i=5
B's i=1
B's i=2
B's i=3
B's i=4
B's i=5
```



```
E:\JAVA>java  RunnableTest
A's i=1
A's i=2
A's i=3
A's i=4
B's i=1
A's i=5
B's i=2
B's i=3
B's i=4
B's i=5

E:\JAVA>
```

Which way of creating Thread is better?

Answer: We may get one question, that is which way of creating thread is better. The Answer is, if you want use others methods of the thread, then use Thread super class. If you are interested in only run()

Advantage of the Multithreading

- It enables you to write very efficient programs that maximizes the CPU utilization and reduces the idle time.
- Most I/O devices such as network ports, disk drives or the keyboard are much slower than CPU
- A program will spend much of it time just send and receive the information to or from the devices, which in turn wastes the CPU valuable time.
- By using the multithreading, your program can perform another task during this idle time.
- For example, while one part of the program is sending a file over the internet, another part can read the input from the keyboard, while other part can buffer the next block to send.
- It is possible to run two or more threads in multiprocessor or multi core systems simultaneously.

When a Thread is ended: *isAlive() and join()* methods

It is often very important to know which thread is ended. This helps to prevent the main from terminating before the child Thread is terminating. To address this problem "Thread" class provides two methods: **1) Thread.isAlive() 2) Thread.join()**.

The general form of the "isAlive()" method is as follows:

final boolean isAlive();

This method returns the either "**TRUE**" or "**FALSE**" . It returns "TRUE" if the thread is alive, returns "FALSE" otherwise.

General form of run() method

public void join();

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( ).** This method forces the currently running thread to finish it task first.

This method also has another version as follow:

Public void run(milliseconds);

Here, time milliseconds are used to specify the deadline within which the thread has to finish its task, context-switching is done to transfer control otherwise.

Example program: displaying two tables (2 and 3 table)

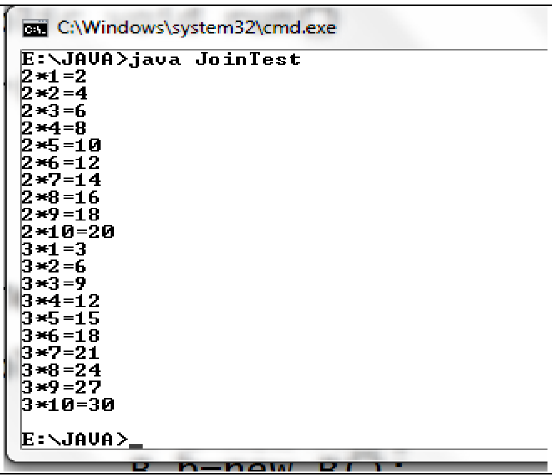
Without join() method

```
class A implements Runnable
{
        public void run()
        {
        for(int i=1;i<=10;i++)
        {

        System.out.println("2*"+i+"="+(2*i));
        }
        }
}
class B implements Runnable
{
        public void run()
        {
        for(int j=1;j<=10;j++)
        {

        System.out.println("3*"+j+"="+(3*j));
        }
        }
}
class JoinTest
{
        public static void main(String args[])
        {
                A a=new A();
                B b=new B();
                Thread      t1=new      Thread(a);
//providing runnable object as argument
                Thread t2=new Thread(b);
                t1.start();
                t2.start();
        }
}
```

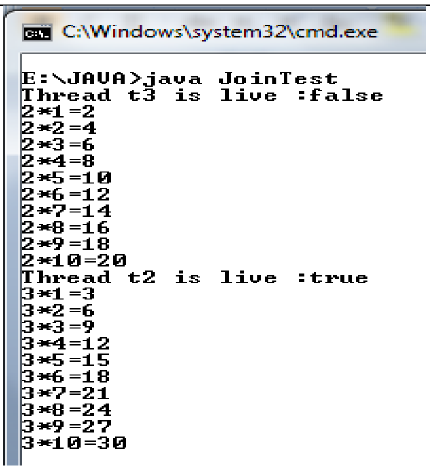
With join() method

```
class A implements Runnable
{
        public void run()
        {
        for(int i=1;i<=10;i++)
        {

        System.out.println("2*"+i+"="+(2*i));
        }
        }
}
class B implements Runnable
{
        public void run()
        {
        for(int j=1;j<=10;j++)
        {

        System.out.println("3*"+j+"="+(3*j));
        }
        }
}
class JoinTest
{
public static void main(String args[])
{
A a=new A();
B b=new B();
Thread  t1=new  Thread(a);   //providing  runnable
object as argument
Thread t2=new Thread(b);
Thread t3=new Thread(); //new thread t3 is created,
but note starte
//calling isAlive() method
System.out.println("Thread t3 is live :"+t3.isAlive());
t1.start();
//calling join() method
try
        {
        t1.join();
        }
catch(InterruptedException ie)
        {
        }
t2.start();
System.out.println("Thread t2 is live :"+t2.isAlive());
}
}
```

Without join() output	With join() output



```
E:\JAVA>java JoinTest
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
3*1=3
3*2=6
3*3=9
3*4=12
3*5=15
3*6=18
3*7=21
3*8=24
3*9=27
3*10=30

E:\JAVA>_
```



```
E:\JAVA>java JoinTest
Thread t3 is live :false
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
Thread t2 is live :true
3*1=3
3*2=6
3*3=9
3*4=12
3*5=15
3*6=18
3*7=21
3*8=24
3*9=27
3*10=30
```

<u>Daemon Thread</u>

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

Properties:

- They cannot prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution
- If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
- It is an utmost low priority thread.
- ***When is a user thread is finished, the JVM automatically terminates the Daemon threads.***

Methods:

1. **void setDaemon(boolean status):** This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.
   Syntax:

   public final void setDaemon(boolean on)

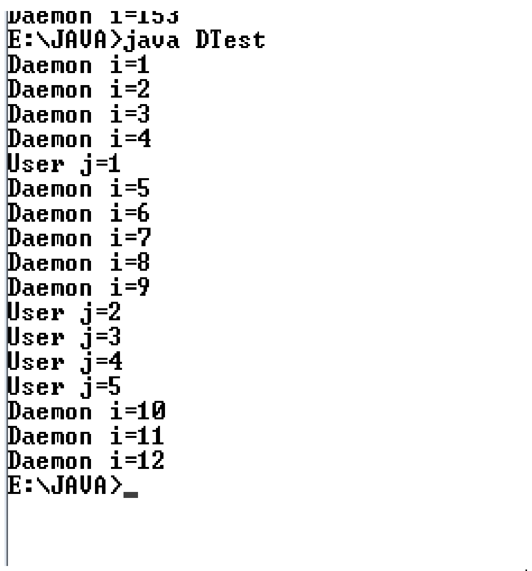
2. ***boolean isDaemon():***
   This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.
   Syntax:

   public final boolean isDaemon()returns:
   true or false

DTest.java

```
class DT extends Thread
{
	public void run()
	{

		for(int i=1;i<=1000;i++)
		System.out.println("Daemon i="+i);

	}
}
class UD extends Thread
{
	public void run()
	{
		for (int j=1;j<=5;j++)
		System.out.println("User j="+j);
	}
}
class DTest
{
	public static void main(String args[])
	{
	DT d=new DT();
	d.setDaemon(true);
	UD u=new UD();
	d.start();
	u.start();

	}
}
```

Output:



```
Daemon i=153
E:\JAVA>java DTest
Daemon i=1
Daemon i=2
Daemon i=3
Daemon i=4
User j=1
Daemon i=5
Daemon i=6
Daemon i=7
Daemon i=8
Daemon i=9
User j=2
User j=3
User j=4
User j=5
Daemon i=10
Daemon i=11
Daemon i=12
E:\JAVA>_
```

The Thread Priorities

Thread priorities are used by the thread ***scheduler*** to decide when and which thread should be allowed to run. In theory, **higher-priority** threads get more CPU time than **lower-priority** threads. In practice, the amount of CPU time that a thread gets often depends on **several factors** besides its priority. (For example, how an operating system implements ***multitasking*** can affect the relative availability of CPU time.) A higher-priority thread can also **preempts** (stops) a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread **resumes** (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must bewithin the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which iscurrently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,shown here:

final int getPriority( )

Example Program: **PTest.java**

```
class PThread1 extends Thread

{

        public void run()

        {

                System.out.println(" Child 1 is started");

        }

}

class PThread2 extends Thread

{

        public void run()

        {

                System.out.println(" Child 2 is started");

        }

}

class PThread3 extends Thread
```

```
class PTest

{
```

```
    public static void main(String args[])
    //setting the priorities to the thread using the setPriority() method

            PThread1 pt1=new PThread1();
                    pt1.setPriority(1);

            PThread2 pt2=new PThread2();
                    pt2.setPriority(9);

            PThread3 pt3=new PThread3();
                    pt3.setPriority(6);

            pt1.start();

            pt2.start();
}
```

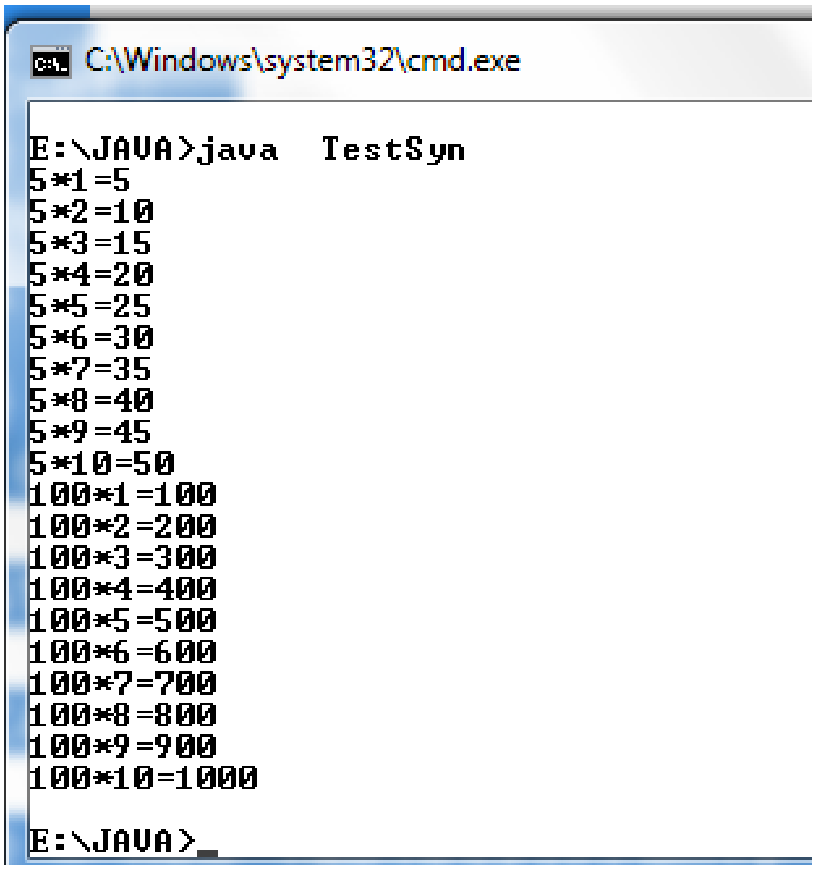
Note: Of course, the exact output produced by this program depends on the speed of your CPU and the number of other tasks running in the system. When this same program is run under a non preemptive system, different results will be obtained.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization***.

Key to synchronization is the concept of the **monitor** (also called a ***semaphore***). A *monitor* is an object that is used as a mutually exclusive lock, or ***mutex***. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have ***entered*** the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor. These other threads are said to be ***waiting*** for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

Let us try to understand the problem without synchronization. Here, in the following example two threads are accessing the same resource (object) to print the Table. The Table class contains one method, ***printTable(int)***, which actually prints the table. We are creating two Threads, Thread1 and Thread2, which are using the same instance of the Table Resource (object), to print the table. When one thread is using the resource, no other thread is allowed to access the same resource Table to print the table.



```
E:\JAVA>java  TestSyn
5*1=5
5*2=10
5*3=15
5*4=20
5*5=25
5*6=30
5*7=35
5*8=40
5*9=45
5*10=50
100*1=100
100*2=200
100*3=300
100*4=400
100*5=500
100*6=600
100*7=700
100*8=800
100*9=900
100*10=1000

E:\JAVA>_
```

In the above output it can be observed that when Thread1 is accessing the Table object, Thread2 is not allowed to access it. Thread1 preempts the Thread2 from accessing the printTable() method.

Note:

1. This way of communications between the threads competing for same resource is called **implicit communication.**
2. This has one disadvantage due to polling. The polling wastes the CPU time. To save the CPU time, it is preferred to go to the **inter-thread communication.**

Inter-Thread Communication

If two or more Threads are communicating with each other, it is called "inter thread" communication. Using the synchronized method, two or more threads can communicate indirectly. Through, synchronized method, each ***thread always competes for the resource***. This way of competing is called **polling**. The polling wastes the much of the CPU valuable time. The better solution to this problem is, just notify other threads for the resource, when the current thread has finished its task, meanwhile other threads will be doing some useful work. This is **explicit communication** between the threads.

Java addresses this polling problem, using via **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify( )** or **notifyAll( )** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Sun recommends that calls to **wait( )** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Example program for producer and consumer problem:class Q

```
{
        int n;
        boolean valueSet = false;//flag
        synchronized int get()
        {
        while(!valueSet)
        try {
                wait();
        }
         catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
        } //end of the get() method
        synchronized void put(int n)
        {
        while(valueSet)
        try {
                wait();
        }
        catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();

        }//end of the put method
} //end of the class Q


class Producer implements Runnable
{
Q q;
Producer(Q q)
{
this.q = q;
new Thread(this, "Producer").start();
}
public void run()

{
int i = 0;
        while(true)
        {
                q.put(i++);
        }
}
}//end of Producer

class Consumer implements Runnable
{
        Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();
}
public void run()
```

```
{
        while(true)
        {
                q.get();
        }
}
}//end of Consumer


class PCFixed
{
public static void main(String args[])
{
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
}
}
```

Suspending, Blocking and StoppingThreads

Whenever we want stop a thread we can stop from running using "**stop()**" method of thread class. It's general form will be as follows:

Thread.stop();

This method causes a thread to move from **running** to **dead** state. A thread will also move to dead state automatically when it reaches the end of its method.

Blocking Thread

A thread can be temporarily suspended or blocked from entering into the runnable and running state by using the following methods:

sleep()	—blocked for specified time
suspend()	---------blocked until further orders
wait()	--blocked until certain condition occurs

These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invoked in the case of **suspend()**, and the **notify()** method is called in the case of **wait()**.

Example program:

The following program demonstrates these methods:

// Using suspend() and resume().

TestThread.java

```
//(1) declare a class
class AThread extends Thread
{
        //(2) override run() method
        public void run()
        {
                try{
                for(int i=1;i<=10;i++)
                {
                System.out.println("A Thread i ="+i);
                Thread.sleep(1000);
                }
                }
                catch(Exception e)
                {
                }
        }
}
//(1) declare a class
class BThread extends Thread
{
        //(2) override run() method
        public void run()
        {
                try{
                for(int j=1;j<=10;j++)
                {
                System.out.println("B Thread j ="+j);
                Thread.sleep(1000);
                }
                }
                catch(Exception e)
                {
                }
        }
}
class TestThread
{
        public static void main(String args[])
        {
                //(3) create object from above class and call start() method
                AThread a=new AThread();
                BThread b=new BThread();
                //call start() method
                a.start();
                b.start();
```

```
try
{
        Thread.sleep(1000);
        System.out.println("Thread a is
        suspending");a.suspend(); //suspending
        Thread.sleep(1000);
        System.out.println("Thread a is Resuming");
        a.resume(); // resuming

        b.stop();  //stoping

}
catch(Exception e)
{
```

Thread Exceptions

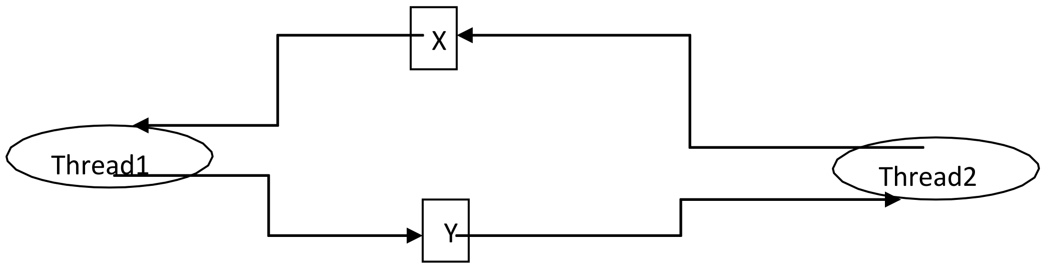
Note that a call to the ***sleep()*** method is always enclosed in try/ catch block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception the program will not compile. The ***join()*** method also generates an exception called ***"InterruptedException"***.

It's general form will be as follows

```
try
{
        Thread.sleep(1000);
}
cathc(Exception e)
{       --------
        ----------
}
```

Deadlock

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called **deadlock.**



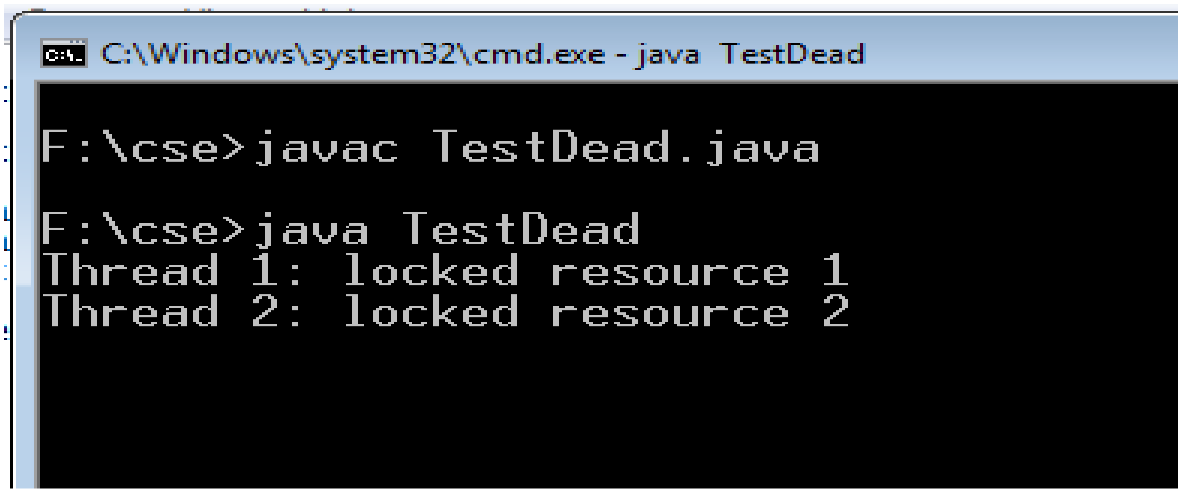
Example program:

```
public class TestDead
{
```

TestDead.java

```
  public static void main(String[] args)
{
    final String resource1 = "John Gardner";
    final String resource2 = "James Gosling";
    // t1 tries to lock resource1 then resource2

    Thread t1 = new Thread()
            {
            public void run()
            { //locking the resource
        synchronized (resource1)
            {
           System.out.println("Thread 1: locked resource 1");

           try {
                     Thread.sleep(100);
             }
catch (Exception e)
              {
                                    System.out.println(e);
                        }

          synchronized (resource2)
              {
           System.out.println("Thread 1: locked resource 2");
          }
       }
     } //end of run()
   }; //end of t1

   // t2 tries to lock resource2
   then resource1Thread t2 =
   new Thread()
   {
     public void run()
     {
       synchronized (resource2)
       {
         System.out.println("Thread 2: locked

         resource 2");try { Thread.sleep(100);}

         catch (Exception e) {} synchronized

         (resource1)

         {
           System.out.println("Thread 2: locked resource 1");
         }
      }
     }//end of run()
   }; //end of t2


   t1.start();
   t2.start();
  }
}
```

Output:



```
F:\cse>javac TestDead.java

F:\cse>java TestDead
Thread 1: locked resource 1
Thread 2: locked resource 2
```

JDBC – Introduction

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

- Core JAVA Programming
- SQL or MySQL Database

JDBC Architecture

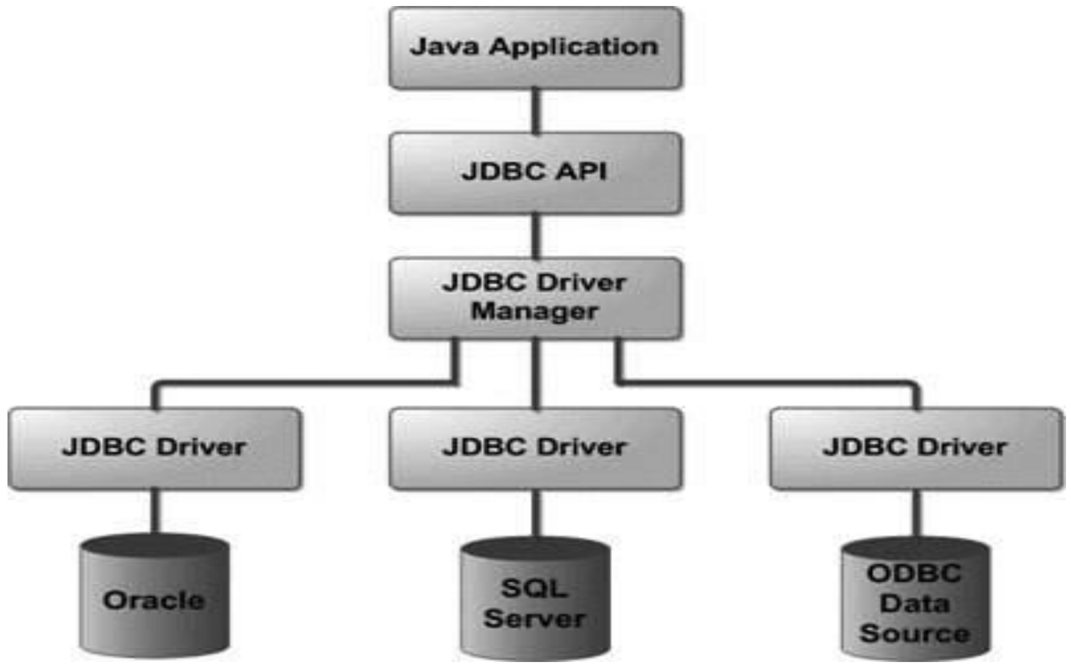
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API** − This provides the application-to-JDBC Manager connection.
- **JDBC Driver API** − This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager** − This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver** − This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection** − This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement** − You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet** − These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException** − This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

Installing the Driver and Configuring the CLASSPATH

Once you have extracted the distribution archive, you can install the driver by placing mysql-connector-java-***version***-bin.jar in your classpath, either by adding the full path to it to your CLASSPATH environment variable, or by directly specifying it with the command line switch -cp when starting the JVM.
To use the driver with the JDBC DriverManager, use com.mysql.jdbc.Driver as the class that implements java.sql.Driver.
You can set the CLASSPATH environment variable under Unix, Linux, or macOS either locally for a user within their .profile, .login or other login file. You can also set it globally by editing the global /etc/profile file.
For example, add the Connector/J driver to your CLASSPATH using one of the following forms, depending on your command shell:

```
# Bourne-compatible shell (sh, ksh, bash, zsh):
shell> export CLASSPATH=/path/mysql-connector-java-ver-bin.jar:$CLASSPATH

# C shell (csh, tcsh):
shell> setenv CLASSPATH /path/mysql-connector-java-ver-bin.jar:$CLASSPATH
```

For Windows platforms, you set the environment variable through the System Control Panel.

To use MySQL Connector/J with an application server such as GlassFish, Tomcat, or JBoss, read your vendor's documentation for more information on how to configure third-party class libraries, as most application servers ignore the CLASSPATH environment variable. For configuration examples for some J2EE application servers, see Chapter 7, *Connection Pooling with Connector/J*, Section 8.2, “Configuring Load Balancing with Connector/J”, and Section 8.4, “Advanced Load-balancing and Failover Configuration”. However, the authoritative source for JDBC connection pool configuration information for your particular application server is the documentation for that application server.
If you are developing servlets or JSPs, and your application server is J2EE-compliant, you can put the driver's .jar file in the WEB-INF/lib subdirectory of your webapp, as this is a standard location for third party class libraries in J2EE web applications.
You can also use the MysqlDataSource or MysqlConnectionPoolDataSource classes in the com.mysql.jdbc.jdbc2.optional package, if your J2EE application server supports or requires them. Starting with Connector/J 5.0.0, the javax.sql.XADataSource interface is implemented using the com.mysql.jdbc.jdbc2.optional.MysqlXADataSource class, which supports XA distributed transactions when used in combination with MySQL server version 5.0 and later.

The various MysqlDataSource classes support the following parameters (through standard set mutators):

- user
- password
- serverName (see the previous section about failover hosts)
- databaseName
- port

JDBC - Environment Setup

Install Java

Java SE is available for download for free. To download click here, please download a version compatible with your operating system.

Follow the instructions to download Java, and run the **.exe** to install Java on your machine. Once you have installed Java on your machine, you would need to set environment variables to point to correct installation directories.

Setting Up the Path for Windows 2000/XP

Assuming you have installed Java in c:\Program Files\java\jdk directory −

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now, edit the 'Path' variable and add the path to the Java executable directory at the end of it. For example, if the path is currently set to C:\Windows\System32, then edit it the following way

```
C:\Windows\System32;c:\Program Files\java\jdk\bin
```

Setting Up the Path for Windows 95/98/ME

Assuming you have installed Java in c:\Program Files\java\jdk directory −

- Edit the 'C:\autoexec.bat' file and add the following line at the end −

```
SET PATH = %PATH%;C:\Program Files\java\jdk\bin
```

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

For example, if you use bash as your shell, then you would add the following line at the end of your **.bashrc** −

```
export PATH = /path/to/java:$PATH'
```

You automatically get both JDBC packages **java.sql** and **javax.sql,** when you install J2SE Development Kit.

Install Database

The most important thing you will need, of course is an actual running database with a table that you can query and modify.

Install a database that is most suitable for you. You can have plenty of choices and most common are −

- **MySQL DB** − MySQL is an open source database. You can download it from MySQL Official Site. We recommend downloading the full Windows installation.

  In addition, download and install MySQL Administrator as well as MySQL Query Browser. These are GUI based tools that will make your development much easier.

  Finally, download and unzip MySQL Connector/J (the MySQL JDBC driver) in a convenient directory. For the purpose of this tutorial we will assume that you have installed the driver at C:\Program Files\MySQL\mysql-connector-java-5.1.8.

  Accordingly, set CLASSPATH variable to C:\Program Files\MySQL\mysql-connector-java-5.1.8\mysql-connector-java-5.1.8-bin.jar. Your driver version may vary based on your installation.

Set Database Credential

When we install MySQL database, its administrator ID is set to **root** and it gives provision to set a password of your choice.

Using root ID and password you can either create another user ID and password, or you can use root ID and password for your JDBC application.

There are various database operations like database creation and deletion, which would need administrator ID and password.

For rest of the JDBC tutorial, we would use MySQL Database with **guest** as ID and **guest123** as password.

If you do not have sufficient privilege to create new users, then you can ask your Database Administrator (DBA) to create a user ID and password for you.

Create Database

To create the **TUTORIALSPOINT** database, use the following steps −

Step 1

Open a **Command Prompt** and change to the installation directory as follows −

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

Note − The path to **mysqld.exe** may vary depending on the install location of MySQL on your system. You can also check documentation on how to start and stop your database server.

Step 2

Start the database server by executing the following command, if it is already not running.

```
C:\Program Files\MySQL\bin>mysqld
C:\Program Files\MySQL\bin>
```

Step 3

Create the **TUTORIALSPOINT** database by executing the following command −

```
C:\Program Files\MySQL\bin> mysqladmin create TUTORIALSPOINT -u guest -p
Enter password: ********
C:\Program Files\MySQL\bin>
```

Create Table

To create the **Employees** table in TUTORIALSPOINT database, use the following steps −

Step 1

Open a **Command Prompt** and change to the installation directory as follows −

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

Step 2

Login to the database as follows −

```
C:\Program Files\MySQL\bin>mysql -u guest -p
Enter password: ********
mysql>
```

Step 3

Create the table **Employees** as follows −

```
mysql> use TUTORIALSPOINT;
mysql> create table Employees
   -> (
   -> id int not null,
   -> age int not null,
   -> first varchar (255),
   -> last varchar (255)
   -> );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

Create Data Records

Finally you create few records in Employee table as follows −

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
```

```
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

Creating JDBC Application

There are following six steps involved in building a JDBC application −

- **Import the packages** − Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set** − Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment** − Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in the previous chapter.

Copy and paste the following example in FirstExample.java, compile and run as follows −

```
import java.sql.*;

public class FirstExample {
   static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
   static final String USER = "guest";
   static final String PASS = "guest123";
   static final String QUERY = "SELECT id, first, last, age FROM Employees";

   public static void main(String[] args) {
      // Open a connection
      try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(QUERY);) {
         // Extract data from result set
         while (rs.next()) {
            // Retrieve by column name
            System.out.print("ID: " + rs.getInt("id"));
```

```
          System.out.print(", Age: " + rs.getInt("age"));
          System.out.print(", First: " + rs.getString("first"));
          System.out.println(", Last: " + rs.getString("last"));
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
   }
}
```

Now let us compile the above example as follows −

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces the following result −

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

JDBC - Database Connections

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps −

- **Import JDBC Packages** − Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver** − This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation** − This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object** − Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code −

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver −

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows −

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
   System.exit(2);
catch(InstantiationException ex) {
   System.out.println("Error: unable to instantiate driver!");
   System.exit(3);
}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver −

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods −

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password* −

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be −

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows −

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL −

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form −

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows −

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object −

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code −

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows −

```
conn.close();
```

JDBC - Result Sets

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods** − Used to move the cursor around.
- **Get methods** − Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods** − Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet −

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

```
try {
   Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
}
```

```
catch(Exception ex) {
   ....
}
finally {
   ....
}
```

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including −

S.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException** Moves the cursor to the next row. This method returns false if there are no more rows in

	the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

For a better understanding, let us study Navigate - Example Code.

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

For a better understanding, let us study Viewing - Example Code.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods −

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()** Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()** Refreshes the data in the result set to reflect any recent changes in the database.

4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

JDBC - Batch Processing

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The **addBatch()** method of *Statement*, *PreparedStatement*, and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

- Create a Statement object using either *createStatement()* methods.
- Set auto-commit to false using *setAutoCommit()*.
- Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
- Execute all the SQL statements using *executeBatch()* method on created statement object.
- Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object −

```
// Create statement object
Statement stmt = conn.createStatement();
```

```
// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
             "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

For a better understanding, let us study the Batching - Example Code.

Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

1. Create SQL statements with placeholders.
2. Create PrepareStatement object using either *prepareStatement()* methods.
3. Set auto-commit to false using *setAutoCommit()*.
4. Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object −

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(?, ?, ?, ?)";

// Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);
```

```
// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.
//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

JDBC –Transactions

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions −

- To increase performance.
- To maintain the integrity of business processes.
- To use distributed transactions.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit −

```
conn.setAutoCommit(false);
```

Commit & Rollback

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows −

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code −

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

For a better understanding, let us study the Commit - Example Code.

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints −

- **setSavepoint(String savepointName)** − Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName)** − Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees " +
                "VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   // If there is no error, commit the changes.
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.