# UNIT-5

## I. String Handling in Java:

1. Introduction
2. Interface Char Sequence
3. Class String
4. Methods for Extracting Characters from Strings
5. Methods for Comparison of Strings
6. Methods for Modifying Strings
7. Methods for Searching Strings
8. Data Conversion and Miscellaneous Methods
9. Class String Buffer
10. Class String Builder.

## II. Multithreaded Programming:

1. Introduction
2. Need for Multiple Threads
3. Thread Class
4. Main Thread- Creation of New Threads
5. Thread States
6. Thread Priority-Synchronization
7. Deadlock and Race Situations
8. Inter-thread Communication - Suspending
9. Resuming and Stopping of Threads.

## III. Java Database Connectivity:

1. Introduction
2. JDBC Architecture
3. Installing MySQL and MySQL Connector
4. JDBC Environment Setup
5. Establishing JDBC Database Connections
6. ResultSet Interface
7. Creating JDBC Application
8. JDBC Batch Processing
9. JDBC Transaction Management

# I. String Handling in Java:

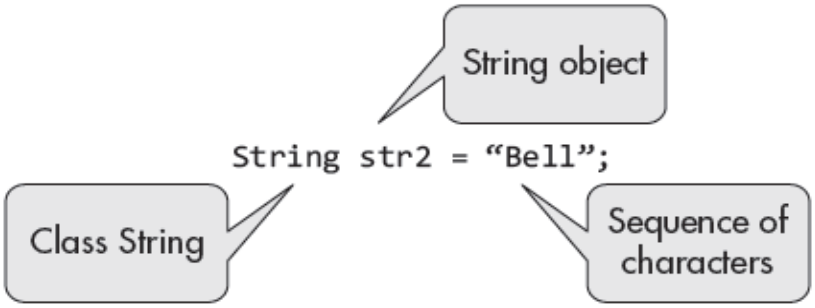
## 1. Introduction

- Strings are treated differently in Java compared to C language;
- In the latter case, it represents an array of characters terminated by null character.
- **In Java, a string is an object of a class**, and **there is no automatic appending of null character by the system.**
- In Java, there are three classes that can create strings and process them with nearly similar methods.
  - (i) class String
  - (ii) class StringBuffer
  - (iii) class StringBuilder
- All the three classes are part of java.lang package.
- All the three classes have several constructors that can be used for constructing strings.
- In the case of **String** class, an object may be created as

String str1 = "abcd";

Here :

- String is a Predefined class of **`java.lang`** package
- Str1 is an object not a variable.
- "abcd" is string literal




- The aforementioned two declarations are equivalent to the following:

```
char s1[] = {'a', 'b', 'c', 'd'};
char s2[] = {'B', 'e', 'l', 'l'};
```

**Storage of Strings**

- The memory allocated to a Java program is divided into two segments:
  - (i) Stack
  - (ii) Heap
- The objects of class String have a special storage facility, which is not available to objects of other two String classes or to objects of any other

class.

- The variables are stored on heap, whereas the program is stored on stack.
- Within the heap, there is a memory segment called 'String constant pool'.
- The String class objects can be created in two different ways:

```
String strx = "abcd";
String strz = new String("abcd");
```

**Example: StringTest.java**

```
public class StringTest
{
   public static void main(String args[])
   {
      String strx = "abcd"; //Object stored in pool
      String stry = "abcd";  // only one "abcd" exists in the pool
      String strz = new String("abcd"); //new object
      String str1 = new String("abcd"); // new object
      String s2 = new String(); // empty string
      String s1 =""; //empty string - no space in the string

     System.out.println("Are reference of strx and stry same?  " +
(strx == stry));
     System.out.println("Are reference of strx and strz same?  " +
(strx == strz));
     System.out.println("Are reference of str1 and strz same?  " +
(str1 == strz));
     System.out.println("Are reference of s1 and s2 same?  " + (s1
== s2));
     System.out.println("Are strings in strx and strz equal?  " +
strx.equals(strz));
   }
}
```

Output:

```
C:\>javac StringTest.java

C:\>java StringTest
Are reference of strx and stry same?  true
Are reference of strx and strz same?  false
Are reference of str1 and strz same?  false
Are reference of s1 and s2 same?  false
Are strings in strx and strz equal?  true
```

**Immutability**

**i. String class**

- **The string objects created by class String are immutable**.
- By immutable implies that once an object is created, its value or contents cannot be changed.
- Neither the characters in the String object once created nor their case (upper or lower) can be changed.
- New String objects can always be created and assigned to older String object references.
- Thus, when you change the content of a string by defining a new string, the old and new remain in the memory.
- The immutable objects are **thread safe** and so are the String objects.

**ii. StringBuffer class**

- **The objects created by class StringBuffer are mutable.**
- These are stored on the heap segment of memory outside the String constant pool.
- The contents of StringBuffer strings may be changed without creating new objects.
- The methods of StringBuffer are synchronized, and hence, **they are thread safe**.

**iii. StringBuilder class**

- **The objects of class StringBuilder are also mutable but are not thread safe.**
- The operations are fast as compared to StringBuffer and there is no memory loss as is the case with String class.
- The class StringBuilder has the same methods as the class StringBuffer.
- Therefore, if multithreads are not being used, then StringBuilder class should be used to avoid memory loss.

**Properties of String, StringBuffer, and StringBuilder objects**

**Table 12.1** Properties of String, StringBuffer, and StringBuilder objects

| *Property* | *String* | *StringBuffer* | *StringBuilder* |
|---|---|---|---|
| Storage area | Constant string pool and heap | Heap | Heap |
| Object mutability | Not mutable | Mutable | Mutable |
| Thread safety | Safe | Safe | Not safe |
| Performance | Fast | Slow | Fast |

## 2. Interface CharSequence

- It is an interface in java.lang package.
- It is implemented by several classes including the classes String, StringBuffer, and StringBuilder.

It has the following four methods.

(i) char charAt(int index): The method returns character value at specified index value.

(ii) int length(): This method returns the length of this (invoking) character sequence.

(iii) CharSequence subsequence(int startIndex, endIndex):
The method returns a subsequence from start index to end index of this sequence Throws IndexOutOfBoundsException.

(iv) String to String(): The method returns a string containing characters of the sequence in the same.

**Example: CharSq.java**

```
public class CharSeq
{
   public static void main(String args[])
   {
      CharSequence csq1 = "ABCD";
      System.out.println("Second  letter  in  csq1  =  "  +
csq1.charAt(1));
      System.out.println("Length of csq1 = " + csq1.length());

      CharSequence csq2 = new StringBuffer("XYZ12345");
      CharSequence csq3 = new StringBuffer("Amrit");
      CharSequence  sq  = csq2.subSequence(2,6);

      System.out.println("The csq3 = " + csq3);
      System.out.println("The csq2 = " + csq2);
System.out.println("The sub sequence(2,6) of csq2  sq = " + sq);
   }
}
```

```
Output:

C:\>javac CharSeq.java

C:\>java CharSeq
Second letter in csq1 = B
Length of csq1 = 4
The csq3 = Amrit
The csq2 = XYZ12345
The sub sequence(2,6) of csq2  sq = Z123
```

## 3. Class String

- The class String is used to represent strings in Java. It is declared as

```
public final class String
extends Object
implements serializable, comparable<String>, charSequence
```

- The String class is final, it cannot be extended.
- Following are the constructors of class String:

**1. *String()***

- This constructor creates a string without any characters. See the following examples.

```
String str = new String();
String str1 = "";
```

**2. *String (byte* [] *barray)***

- It constructs a new string by decoding the specified byte[] barray by using a computer's default character set. The following code
- constructs str2 = "ABCDE".

```
byte []bray = new byte[]{65, 66, 67, 68, 69};
String str2 = new String(bray);
```

**Example: StringArray.java**

```
public class StringArray
{
   public static void main(String args[])
   {
      byte []bray = new byte[]{65, 66, 67, 68, 69};
      String str2 = new String(bray);

      System.out.println("str2 =" + str2);
   }
}
```

```
Output:
C:\>javac StringArray.java

C:\>java StringArray
str2 =ABCDE
```

**3. *String (byte* [] *barray, Charset specifiedset)***

- It constructs a new string by decoding the specified byte array (bray) by using specified character set.
- Some of the Charsets supported by Java are UTF8, UTF16, UTF32, and ASCII.
- These may be written in lower case such as utf8, utf16, utf32, and ascii.

**Example: StringUnicode.java**

```
public class StringUnicode
{
   public static void main(String args[])
   {
      byte []unicode = new byte[]{'\u0041','\u0042','\u0043'};
      String str = new String(unicode);

      System.out.println("str =" + str);
   }
}
```

**Output:**

```
C:\>javac StringUnicode.java

C:\>java StringUnicode
str =ABC
```

**4. *String(byte*[] *bray, int offset, int length, String charsetName)***

- The constructor constructs String object by decoding the specified part of byte array using specified Charset.
- For example

  `String str4 = new String(bray,1, 3, “ascii”);`

**Example: StringArray2.java**

```
public class StringArray2
{
   public static void main(String args[]) throws Exception
   {
      byte []bray = new byte[]{65, 66, 67, 68, 69};
      String str = new String(bray,1,3,"ascii");

      System.out.println("str =" + str);
   }
}

Output:
C:\>javac StringArray2.java

C:\>java StringArray2
str =BCD
```

**5. *String (byte[] barray, string charsetName}***

- The constructor constructs a string by decoding the byte array using specified Charset.
  `String str3 = new String(bray, "UTF8");`
- The method throws UnsupportedEncodingException if the given charset is not supported.

**Example: StringUnicode2.java**

```
public class StringUnicode2
{
   public static void main(String args[]) throws Exception
   {
      byte []unicode = new byte[]{'\u0041','\u0042','\u0043'};
      String str = new String(unicode, "UTF8");

      System.out.println("str =" + str);
   }
}
C:\>javac StringUnicode2.java

C:\>java StringUnicode2
str =ABC
```

## 4. Methods for Extracting Characters from Strings

**Table 12.2** Methods for extraction of characters and substrings

| Method | Description |
|---|---|
| char charAt(int n) | Extracts a character at a specified index position |
| void getChars(args) | Extracts more than one character at a time |
| byte[] getBytes() | Encodes a given string into a sequence of bytes and returns an array of bytes |
| int length() | Finds out the length of a given string |
| char toCharArray() | Converts the string into an array of characters and returns the array |

The signatures of the first two methods are as:

1. ***char charAt (int n),*** where *n* is a positive number, which gives the location of a character in a string.

- It returns the character at the specified index location.

- Thus, in String 'Delhi' the index value of 'D' is 0, index value of 'e' is 1, 'h' is 3, and so on.

***2. void getChars (int SourceStartindex, int SourceEndindex, char targetarray[], int targetindexstart)***

- This method takes characters from a string and deposits them in another string.
- int *SourceStartindex* and int Source*Endindex* relate to the source string.
- char *targetarray*[] relates to the array that receives the string characters. The int *targetindexstart* is the index value of target array.

**Example: MethodsChar.java**

```
public class MethodsChar
{
   public static void main(String args[])
   {
      String str1 = "Delhi";
      String str2 = "Vijayawada";
      String str = "I am going to Delhi and from there to Mumbai";

      int begin =14;
      int end =19;
      char aSTR[] = new char[end -begin];
      str.getChars(begin, end, aSTR, 0);

      System.out.println("Length of string str1 =" + str1.length());
      System.out.println("Fourth Character in Delhi = " +
str1.charAt(3));
      System.out.println("Fifth Character in Vijayawada = " +
str2.charAt(4));
      System.out.print("aSTR = ");
      System.out.println(aSTR);
   }
}
Output:
C:\>javac MethodsChar.java

C:\>java MethodsChar
Length of string str1 =5
Fourth Character in Delhi = h
Fifth Character in Vijayawada = y
aSTR = Delhi
```

## 5. Methods for Comparison of Strings

**Table 12.3** Methods for comparison of strings

| *Method* | *Description* |
|---|---|
| `int compareTo (String str)` | Compares the invoking string with argument string lexicographically. It returns a value less than 0, 0, or more than 0 if the invoking string is either less than, equal, or more than argument string, respectively. |
| `int compareToIgnoreCase (String str)` | Compares the invoking string with argument string lexicographically ignoring the case differences. The outputs are as described for `compareTo`. |
| `boolean equals()` | Compares the characters inside the two strings. Returns true if the match occurs, otherwise false. It is different from operator == in which case only the references of the two strings are compared. |
| `boolean equalsIgnoreCase()` | Compares the strings ignoring the case of letters |
| `boolean regionMatches(String str)` | Compares a specified region in one string with a specified region in another string. Returns true if the two match, otherwise returns false. |
| `endsWith(String str)` | Returns true if a string ends with the given substring |
| `startsWith(String str)` | Returns true if a string starts with the given substring |

- The operator == simply compares the references. E.g.

  if (str1==str2)
- The contents are compared by the method equals() as

  if (str1.equals(str3))

**Example: MethodCompare.java** **compareTo() method**

```
public class MethodCompare
{
   public static void main(String args[])
   {
      String str1 = "AA";
      String str2 = "ZZ";
      String str3 = new String(str1);

      System.out.println("str2.compareTo(str1) =" +
str2.compareTo(str1));
      System.out.println("str1.compareTo(str2) =" +
str1.compareTo(str2));
      System.out.println("str3.compareTo(str1) =" +
str3.compareTo(str1));
   }
}
```

Output:

```
C:\>javac MethodCompare.java

C:\>java MethodCompare
str2.compareTo(str1) =25
str1.compareTo(str2) =-25
str3.compareTo(str1) =0
```

**Example: StringMethods.java - equals() and length() methods**

```
public class StringMethods
{
   public static void main(String args[])
   {
      String str1 = "AA";
      String str2 = "ZZ";
      String str3 = new String(str1);

      System.out.println("str3 = " + str3);

      System.out.println("str2.equals(str1) =" + str2.equals(str1));

      System.out.println("str1.equals(str2) =" + str1.equals(str2));
      System.out.println("str3.equals(str1) =" + str3.equals(str1));

      System.out.println("Length of str2 =" + str2.length());
   }
}
```

Output:

```
C:\>javac StringMethods.java

C:\>java StringMethods
str3 = AA
str2.equals(str1) =false
str1.equals(str2) =false
str3.equals(str1) =true
Length of str2 =2
```

## 6. Methods for Modifying Strings

- The ways to modify strings are as follows:
- 1. Define a new string.
- 2. Create a new string out of substring of an existing string.
- 3. Create a new string by adding two or more substrings.
- 4. Create a new string by replacing a substring of an existing string.
- 5. Create a new string by trimming the existing string.

**Table 12.4** Methods for modifying strings

| *Method* | *Description* |
|---|---|
| `substring(int indexbegin)` | Creates new string as a substring of existing string starting from *indexbegin* to end of string |
| `substring (int indexbegin, int indexend)` | Creates new string that has characters from *indexbegin* to *indexend* of the existing string |
| `String concat (String S)` | Creates a new string by concating the string S with the existing string. It returns a new String object that contains characters from both the String objects |
| `String replace(char current, char replacement)` | Returns string in which all occurrences of a character are replaced with another character |
| `String replace(Char Sequencecurrent, Char Sequence replacement)` | Returns string in which all occurrences of a sequence of characters is replaced with another sequence of characters |
| `String trim()` | Trims/removes the white spaces at the beginning and the end of current string to create a new string. Original string is obtained if there is no white space characters at the beginning or end of string |

Example: ModifyString.java -replace() and substring() methods

```
public class ModifyString
{
   public static void main(String args[])
   {
      String str1 = "Belhi";
      String mstr1 = str1.replace('B', 'D');
      System.out.println("Before Modification str1 = " + str1);
      System.out.println("Modified string mstr1 = " + mstr1);

      String str2 = "       WELCOME      ";
      System.out.println("str2 =" + str2);
      String mstr2 = str2.trim();
      System.out.println("mstr2 =" + mstr2);

      String str3 = "I am going to Delhi and from there to Mumbai";

      String mstr3 = str3.substring(0,19);
      System.out.println("mstr3 =" +  mstr3);

      String mstr4 = str3.substring(19);
      System.out.println("mstr4 =" +  mstr4);
   }
}
```

Output:

```
C:\ >javac ModifyString.java

C:\ >java ModifyString
Before Modification str1 = Belhi
Modified string mstr1 = Delhi
str2 =       WELCOME
mstr2 =WELCOME
mstr3 =I am going to Delhi
mstr4 = and from there to Mumbai
```

## 7. Methods for Searching Strings `indexOf()`

(i) *`int indexOf(int character/substring)`* - searches for the first occurrence of a specified character in the string.

(ii) `int indexOf(int` *`character/substring`*`, int` *`index`*`)` - searches for the first occurrence of a specified character or substring and the search starts from the specified index value, that is, the second argument.

**Example: StringSearch.java**

```
public class SearchString
{
      public static void main (String args[])
      {
            String str1 = "Help the needed ones";
            String str2 = "One has to take care of one's health
oneself";
            System.out.println("The index of \"e\" in the String str1
is at index = " + str1.indexOf('e'));
            System.out.println ("The last index of \"e\" in str1 is
at index = " + str1.lastIndexOf('e'));
            System.out.println ("The last occurrence  \"of\" in
String str2 is at index = " + str2. lastIndexOf("of"));
            System.out.println("The occurrence of \"e\" after index 8
in str1 = " + str1.indexOf('e', 8));
            System.out.println("The index of last occurrence of \"n\"
= " + str1. lastIndexOf('n', 15));
      }
}
```

Output:

```
E:\ >javac SearchString.java

C:\>java SearchString
The index of "e" in the String str1 is at index = 1
The last index of "e" in str1 is at index = 18
The last occurrence  "of" in String str2 is at index = 21
The occurrence of "e" after index 8 in str1 = 10
The index of last occurrence of "n" = 9
```

## 8. Data Conversion and Miscellaneous Methods

**Table 12.5** Some important remaining methods of class `String`

| *Method* | *Description* |
|---|---|
| `int codePointAt(int index)` | Returns code point (character) at specified index |
| `int codePointBefore(int index)` | Returns code point of character before the specified index value |
| `int codePointCount(int indexStart, int indexEnd)` | Returns the code points in the range specified by indexStart and indexEnd |
| `boolean contains(CharSequence cseq)` | Returns true if the string contains the specified CharSequence |
| `boolean contentEquals(CharSequence cseq)` | Compares the invoking string with specified CharSequence and returns true if equal |
| `boolean contentEquals(BufferString bufstr)` | Compares the invoking string with specified `StringBuffer` string and returns true if equal |
| `static String copyValueOf(char [] chary)` | Returns a string that represents the character sequence in the char array |
| `static String copyValueOf(char [] chary, int offset, int count)` | Returns a string that represents the character sequence in the specified part of the array |
| `static String format(Locale loc, String format, Object …arguments)` | Returns the formatted string according to the specified format and other objects |
| `byte [] getbytes()` | Converts the invoking string into an array of bytes using the default Charset of the computer and stores the result into a new byte array |
| `byte [] getbytes(Charset cset)` | Encodes the string into a sequence of bytes by using the given Charset and returns byte array and stores the result into a new byte array |

**Example: StringValue.java**

```
public class StringValue
{
      public static void main(String[] args)
      {
      int n = 70;
      long l= 25674;
      float fit = 45.76f;
      String s1 = new String("Railways");
      String s2 = new String();
      String s3 = s2.valueOf (fit);
      char [] array = {'D', 'e', '1', 'h', 'i'};

      System.out.println("s2.valueOf(n) = " + s2.valueOf(n));
      System.out.println("s2.valueOf(l) = " + s2.valueOf(l));
      System.out.println("s3 = " + s3);
      System.out.println("s2.valueOf(array) = " + s2.valueOf(array));
      System.out.println("s1.toString() = " +s1.toString());
      }
}

C:\>javac StringValue.java

C:\>java StringValue
s2.valueOf(n) = 70
s2.valueOf(l) = 25674
s3 = 45.76
s2.valueOf(array) = De1hi
s1.toString() = Railways
```

## 9. Class String Buffer

It defines the strings that can be modified as well as the number of characters that may be changed, replaced by another, a string that may be appended, etc.

- The strings are also thread safe. For the strings created by class StringBuffer, the compiler allocates extra capacity for 16 more characters so that small modifications do not involve relocation of the string.

The class is declared as follows:

public final class StringBuffer
extends Object
implements Serializable, CharSequence

**Table 12.6** Constructors of `StringBuffer` class

| *Constructor* | *Description* |
|---|---|
| `StringBuffer()` | Creates an empty buffer string but keeps capacity for 16 characters |
| `StringBuffer(int capacity)` `throws NegativeArraySizeException` | Sets the size of the buffer string capacity equal to the sum of the argument and 16 characters, so that the string may be expanded without relocation |
| `StringBuffer (String str)` `throws NullPointException` | Creates and initializes buffer string with the string `str` but it reserves the space for 16 more characters so that it may be modified without relocation |
| `StringBuffer(CharSequence charSeq)` `throws NullPointException` | Creates `StringBuffer` object that is made out of character sequence |

**Table 12.7** Frequently used methods of `StringBuffer` class

| *Method* | *Description* |
|---|---|
| `StringBuffer append(boolean b)` | Appends string representation of argument |
| `StringBuffer append(char ch)` | Appends string representation of argument |
| `StringBuffer append(char[] c)` | Appends string representation of argument |

| *Method* | *Description* |
|---|---|
| `StringBuffer append(CharSeq s)` | Appends string representation of argument |
| `StringBuffer append(double d)` | Appends string representation of argument |
| `StringBuffer append(float flt)` | Appends string representation of argument |
| `StringBuffer append(int n)` | Appends string representation of argument |
| `StringBuffer append(long lng)` | Appends string representation of argument |
| `StringBuffer append(CharSeq s)` | Appends string representation of argument |
| `StringBuffer append(String str)` | Appends the string *str* at the end of invoking `StringBuffer` object |
| `StringBuffer append (CharSequence,int start,int end)` `throws IndexOutOfBoundsException` | Appends a specified subsequence to the invoking sequence |
| `append (char[] str,int offset,int length)throws IndexOutOfBoundsException` | Appends the string representation of char array to this sequence |
| `int capacity()` | Returns allocated capacity of invoking string |
| `char charAt(int index)throws IndexOutOfBoundsException` | Returns value of character at the specified *index* value |
| `void setCharAt(int index,char c)throws IndexOutOfBoundsException` | Sets a specified character c at the specified index value *index* in invoking string |

**Example1: StringBufferDemo.java**

```
class StringBufferDemo
{
      public static void main (String args[])
      {
      StringBuffer bufStr = new StringBuffer ("Hello World
Example" );
      System.out.println("bufStr = " + bufStr);
      System.out.println("Length of bufStr =" +
bufStr.length());

      bufStr.setLength (11);
      System.out.println("New Length of bufStr = " +
bufStr.length());

      System.out.println("Capacity of bufStr =" +
bufStr.capacity());
      System.out.println("New bufStr =" + bufStr );

      char ch=bufStr.charAt(4);
      System.out.println("Character at 4th position = " + ch);

      bufStr.setCharAt(7, 'e');
      System.out.println(" Now New bufStr =" + bufStr);
      }
}
```

Output:

```
C:\>javac StringBufferDemo.java

C:\>java StringBufferDemo
bufStr = Hello World Example
Length of bufStr =19
New Length of bufStr = 11
Capacity of bufStr =35
New bufStr =Hello World
Character at 4th position = o
 Now New bufStr =Hello Werld
```

**Example2: StringBufferDemo2.java**

```
class StringBufferDemo2
{
      public static void main (String args[])
      {
      StringBuffer bufStr = new StringBuffer ("Hello World");
      System.out.println("bufStr =" + bufStr);
      bufStr.reverse();

      System.out.println("After reversing bufStr =" + bufStr);
      StringBuffer str = new StringBuffer("Delhi is a city.");

      System.out.println("Before insert, str = " + str);
      str.insert(11, "very big ");
      System.out.println("After insert, str = " + str);

      str.delete (11,16);
      System.out.println("After deletion str = " + str);

      str.replace (15, 21, "market.");
      System.out.println("After replace str = " + str);

      str.deleteCharAt(21);
      System.out.println("After delete dot, str = " + str);

      str.append(" of").append(" electronic goods.");
      System.out.println("After append str = " + str);
      }
}
```

**Output:**

```
C:\>javac StringBufferDemo2.java

C:\>java StringBufferDemo2
bufStr =Hello World
After reversing bufStr =dlroW olleH
Before insert, str = Delhi is a city.
After insert, str = Delhi is a very big city.
After deletion str = Delhi is a big city.
After replace str = Delhi is a big market.
After delete dot, str = Delhi is a big market
After append str = Delhi is a big market of electronic goods.
```

## 10.Class String Builder.

The StringBuilder class is the subclass of Object in java.lang package.

This class is used for creating and modifying strings. Its declaration is as follows:

```
public final class StringBuilder extends Object
 implements Serializable, CharSequence
```

The four constructors of the class are described as follows:

1. StringBuilder()—Creates a StringBuilder object with no characters but with initial capacity of 16 characters.

2. StringBuilder(CharSequence *chSeq)*—Creates a StringBuilder object with characters as specified in CharSequence *chSeq*.

3. StringBuilder(int *capacity*)—Creates a StringBuilder object with specified capacity. It throws NegativeArraySizeException.

4. StringBuilder(String str)—Creates a StringBuilder object initialized with contents of a specified string. It throws NullPointException if str is null.

**Table 12.8** Methods of `StringBuilder` class

| *Method* | *Description* |
|---|---|
| `StringBuilder append(boolean b)` | Appends string version of `boolean` argument `b` |
| `StringBuilder append(char ch)` | Appends string version of `char` argument |
| `StringBuilder append(char[] str)` | Appends string version of `char` array argument |
| `StringBuilder append(char [] str, int offset, int length) throws IndexOutOfBoundsException` | Appends string version of `char` subarray to the string |
| `StringBuilder append(CharSequence ch) throws IndexOutOfBoundsException` | Appends specified `char` sequence to the string |
| `StringBuilder append(char Sequence s, int start, int end) throwsIndexOutOfBoundsException` | Appends specified part of sequence to string |
| `StringBuilder append(double dbl)` | Appends string version of `double` argument to string |
| `StringBuilder append(float flt)` | Appends string version of `float` argument to string |
| `StringBuilder append(int k)` | Appends string version of `int` argument `k` |
| `StringBuilder append(long lg)` | Appends string version of `long` argument |
| `StringBuilder append(String str)` | Appends `String str` to current string |
| `StringBuilder append(Object obj)` | Appends string version of `Object obj` to string |
| `StringBuilder append(StringBuffer sbuff)` | Appends specified string to invoking string |
| `StringBuilder appendCodePoint(int CodePoint)` | Appends string version of `CodePoint` argument to invoking string |
| `int capacity()` | Returns current capacity of invoking string |
| `char charAt(int index)throws IndexOutOfBoundsException` | Returns character at specified index value |

**Example: StringBuildDemo.java**

```
public class StringBuildDemo
{
      public static void main(String[] args)
      {
      StringBuilder builder1= new StringBuilder("Delhi");
      System.out.println ("Before append, builder1 = " + builder1);
      builder1.append (-110024);
      System.out.println ("After append, builder1 = " + builder1);

      StringBuilder builder2 = new StringBuilder();
      System.out.println("The length of builder2 = "+
      builder2.length());

      System.out.println("The capacity of builder2 = "+
      builder2.capacity());

      System.out.println("Before append, builder2 = " + builder2);
      builder2.append("New Delhi");
      System.out.println("After append, builder2 = " + builder2);
      }0
}
```

Output:

```
C:\>javac StringBuildDemo.java

C:\>java StringBuildDemo
Before append, builder1 = Delhi
After append, builder1 = Delhi-110024
The length of builder2 = 0
The capacity of builder2 = 16
Before append, builder2 =
After append, builder2 = New Delhi
```

**Example: StringBuildDemo2.java**

```
import java.lang.StringBuilder;

public class StringBuildDemo2
{
      public static void main(String[] args)
      {
      StringBuilder builder1= new StringBuilder();

      builder1.insert(0,"Java is a programming language");
      System.out.println ("The string builder =" + builder1);

      builder1.insert (10, "object oriented ");
      System.out.println("The new string is =" + builder1);

      builder1.deleteCharAt(8).delete(7,8);
      System.out.println("The new string after delete = " + builder1);

      StringBuilder builder2 = new StringBuilder();
      builder2.insert(0, "Delhi is a big city");
      System.out.println("The builder2 = " + builder2);

      builder2.insert (0,"New ").insert (18, " business").replace (27, 35,
" center.");
      System.out.println("After modification builder2 = " + builder2);
      }
}
```

**Output:**

```
C:\>javac StringBuildDemo2.java


C:\>java StringBuildDemo2

The string builder =Java is a programming language

The new string is =Java is a object oriented programming language

The new string after delete = Java is object oriented programming language

The builder2 = Delhi is a big city

After modification builder2 = New Delhi is a big business center.
```

# II. Multithreaded Programming

## 1. Introduction

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

- Some computer **programs may be divided into segments** that can run independent of each other or with minimal interaction between them.
- **Each segment may be considered as an independent path of execution called a thread**.
- If the computer has multiple processors, the threads may run concurrently on different processor thus saving computing time.
- Threads are useful even if the computer has a single-core processor.
- The different processes in a computer take different times.

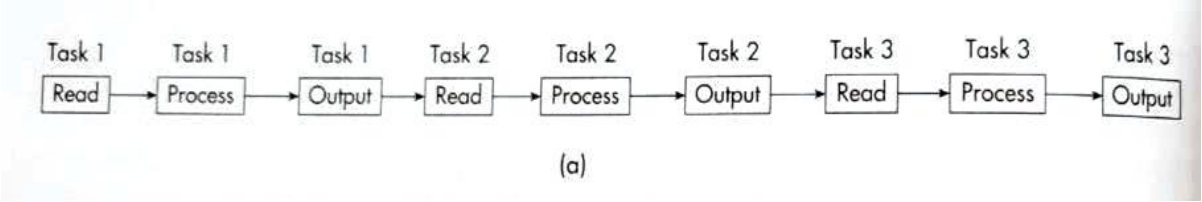
## 2. Need for Multiple Threads

- The present speed of about 15 GHz is about the upper possible limit because beyond this, the cooling problems are tremendous.
- Further, increase in throughput of computer is possible only by dividing the program into segments that are data dependent and can be processed simultaneously by more than one processor;
- thus, it decreases the total time of computation.
- This is the basis on which supercomputers are built.
- In a supercomputer, thousands of processors are employed to concurrently process the data.
- Hardware developers have gone a step further by placing more than one core processor in the same CPU chip.
- Thus, now, **we have multi-core CPUs.**

**Multithreaded Programming for Multi-core Processor**

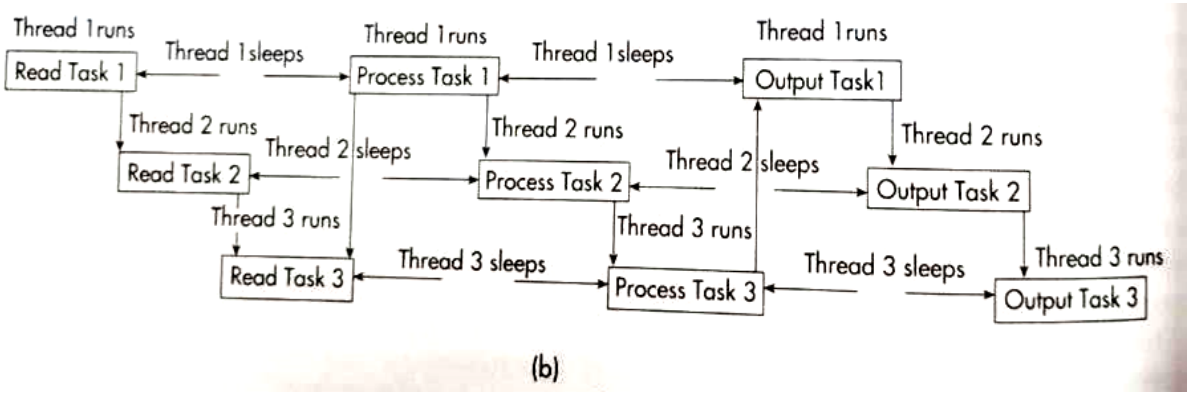
- A CPU may have two cores - dual core or four cores - quad, six cores, or more.
- CPUs having as many as 50 cores have also been developed.
- Moreover, computers with multi-core CPU are affordable and have become part of common man's desktop computer.
- Advancements in hardware are forcing the development of suitable software for optimal utilization of the processor capacity. **Multithread processing is the solution**.
- Multithread programming is inbuilt in Java and CPU capacity utilization may be improved by having multiple threads that concurrently execute different parts of a program.

A. How Single Processor – tasks carried out in Single Sequence is illustrated in the following diagram.



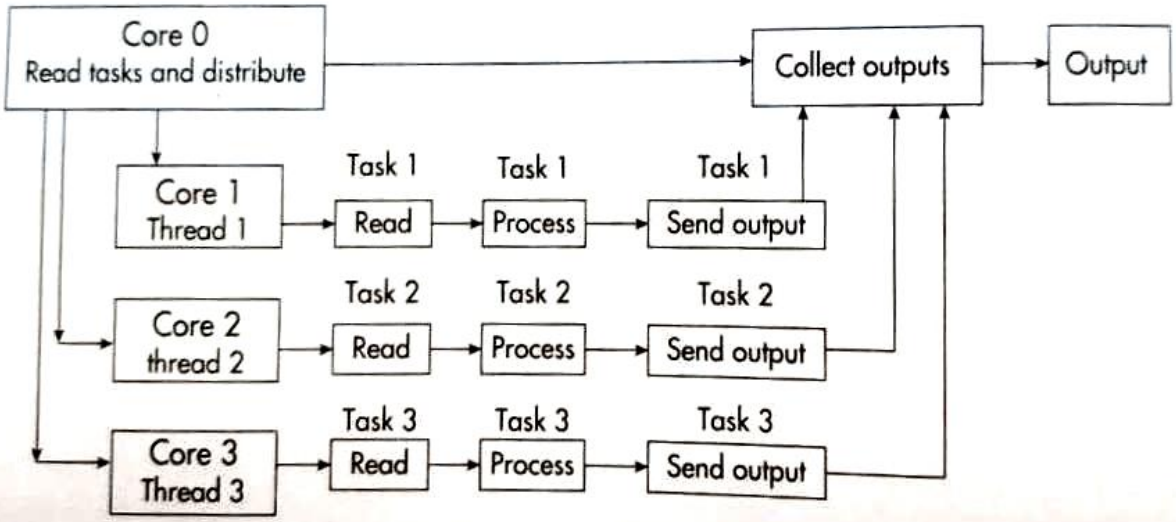

(a)

B. The following diagram shows the Single processor – threads share the time of processor




(b)

C. The following diagram shows the Multi-core processor – threads execute concurrently




(c)

## 3. Thread Class

- In Java, threads are based on the class Thread belonging to java.lang package, that is, java.lang.Thread.
- A thread is an object of class Thread and can invoke the instance methods defined in the class.
- Even if a thread is not explicitly defined, one thread is automatically created by the system for the execution of the main method. **It is generally called main Thread**.
- A thread does not start working on birth. It has to invoke the start() method that gives it the life, otherwise it is a lifeless thread object.
- After getting life, a thread executes a code of instructions (target) as specified by the user in the overloaded method run().

**The Thread class has defined several constructors.**

i. **Thread():** It allocates a new thread object as thread(null, null, generatedname). Every thread must have a name.

ii. **Thread(String threadname)**: It allocates a thread with name specified by the user. It is of the form thread(nall null, name). A thread may be created as

```
Thread t2 new Thread("MyThread");
```

iii. **Thread(Runnable object)** : The constructor allocates a thread with a specified target. The name by the compiler as `Thread-0`, `Thread-1`, and so on.

iv. **Thread (Runnable object, String threadName)**: Thread is allocated with a specified target and user specified name threadnume.

v. **Thread (ThreadGroupgroup, Runnable object)**: It allocates thread with specified group and target.

vi. **Thread (ThreadGroupgroup, Runnable object, String Threadname)**: The constructor allocates thread with specified thread group, target, and thread name.

**Example: ThreadX.java**

```
public class ThreadX extends Thread
{
      public void run()
      {
      System.out.println("It is Threadx class");
      }

      public static void main(String args[])
      {
      Thread a= new Thread (new ThreadX(), "FirstThread");
      Thread b= new Thread (new ThreadX());
      System.out.println("Name of a = " + a.getName());
      System.out.println("Name of b = "+ b.getName());
      th.start();
      t1.start();
      }
}
```

**Output:**

```
C:\ >javac ThreadX.java

C:\ >java ThreadX
Name of a = FirstThread
Name of b = Thread-2
It is Threadx class
It is Threadx class
```

## Thread Group

- Every thread is in fact a member of a thread group. The thread groups are useful for invoking methods that apply to all the threads.
- The thread is made a member of a group at the time of its creation with constructor.
- The thread remains the member of the designated group throughout its life.
- It cannot become the member of another group.
- If a thread's group is not specified in its constructor, as is the usual case, the thread is entered into the same group as the thread that created it.
- The default thread group for a newly executing Java application is the main group.
- When a thread dies, its thread group becomes null

**Methods of Thread Class**

- All threads are objects of class Thread.
- The methods of Thread class for manipulating threads, changing their properties, and understanding their behaviour.
- The class Thread contains several methods that control the execution as well as for setting and getting attributes of threads.
- Methods of Thread class are as follows.

Thread S public void **start()**
public void **run()**
public final boolean **isAlive()**
public final String **getName()**
public static Thread **currentThread()**
public final void **setName(**String name**)**
public static void **yield()**
public static void **sleep** (long milliseconds)
public static void **sleep** (long millisecs, int nanosecs)
public final vold **join()**
public final void **join**(long milliseconds)
public final void **join**(long milliseconds, int nanoseconds)

public final int **getPriority**()
public static int **activeCount**()
public final void **setPriority**(int newpriority)
public long **getID**()
public **Thread.State getState**()
public void **interrupt**()
public static boolean **interrupted**()
public boolean **isInterrupted**()
public final void **checkAccess**()
public static int **enumerate**(Thread [] array)
public String **toString**()
public final boolean **isDaemon()**
public final void **setDaemon**(boolean b)
public static boolean **holdstock**(Object obj)
public final ThreadGroup **getThreadGroup**()

**Deprecated Methods of Class Thread**

The following methods of class Thread have been deprecated because these are either unsafe or are deadlock pruned.

**stop()** : The invoking thread stops executing with clean-up, and thus, exposes sensitive resources to other threads.
**destroy()**: It terminates the thread without clean-up. Deadlock pruned method.
**suspend()**: It temporarily suspends the execution of thread without clean-up. Deadlock pruned method
**resume()**: It brings back the suspended thread to runnable state. Used only after suspend().
**countStackFrames()**: It is not well-defined. Returns number of stack frames in this thread.

**Example: ThreadNew3.java**

```
public class ThreadNew3 extends Thread
{
  public void run()
  {
      System.out.println("In run() Method ");
      System.out.println("The current Thread = " + this.
currentThread());
      System.out.println("Is present thread daemon Thread:"+
this.isDaemon());

      int n = 10;
      System.out.println("Square root of " + n + " = " +
Math.sqrt(n));
      System.out.println("The active count = "+ this.activeCount());
```

```
  }

   public static void main(String args[])
  {
      Thread t1 = new Thread (new ThreadNew3());
      System.out.println("Is Thread ti alive before Start():  " +
t1.isAlive());
      t1.start();
      System.out.println("Is Thread is alive after start(): " +
t1.isAlive());
      System.out.println("ThreadGroup of t1 = " +
t1.getThreadGroup());
      System.out.println("Name of t1 = " + t1.getName());
      t1.setName("SecondThread");
      System.out.println("New name of t1 = " + t1.getName());
  }
}

C:\ >javac ThreadNew3.java

C:\ >java ThreadNew3
Is Thread ti alive before Start():  false
Is Thread is alive after start(): true
In run() Method
ThreadGroup of t1 = java.lang.ThreadGroup[name=main,maxpri=10]
The current Thread = Thread[Thread-1,5,main]
Name of t1 = Thread-1
Is present thread daemon Thread:false
New name of t1 = SecondThread
Square root of 10 = 3.1622776601683795
The active count = 2
```

## 4. Main Thread

- When we run a program in Java, one thread is automatically created and it executes the main method. The thread is called main thread. This is the thread from which other threads are created.
- The threads created from the main thread are called child threads.
- A program keeps running as long as the user threads are running.
- The main thread also has the status of a user thread.
- The child threads spawned from the main thread also have the status of user thread.
- Therefore, main thread should be the last thread to finish so that the program finishes when the main method finishes.
- A thread is controlled through its reference like any other object in Java.
- The main thread can also be controlled by methods of Thread class through its reference that can be obtained by using the static method current Thread(). Its signature is

```
Thread thread = Thread.currentThread();
```

- **sleep()** method suspend the execution for the given specified time interval.
  Ex:

```
thread.sleep(500);
```

the avbove statement will suspend the execution of main() method for 500 ms, which is the argument of the method.

- setName() method add the new name to the existing thread
  Ex:

```
thread.setName("My Thread")
```

Illustration of main thread and methods setName() and getName()

**Example: MyThread.java**

```
class MyThread
{
  public static void main (String args[])
  {
      Thread thread = Thread.currentThread(); //Thread reference
      System.out.println("CurrentThread :" + thread);

      System.out.println("Before modification ,Thread Name =" +
thread.getName());
      System.out.println("Change the name to MyThread.");
      thread.setName("MyThread"); //new name for main thread
      System.out.println("After modification ,Thread Name =" +
thread.getName());
      //try block contains code to be executed by main thread
      try
      {
        for (int i=0; i<4; i++)
         {
              boolean B = thread.isAlive();
```

```
            System.out.println("Is the thread alive? " + B);
            System.out.println(Thread.currentThread()+ " i = "+ i);
            Thread.sleep(1000);
          }
           }catch(InterruptedException e)
        {
         System.out.println("Main thread interrupted");
        }
    }
}

Output:
C:\>javac MyThread.java

C:\>java MyThread
CurrentThread :Thread[main,5,main]
Before modification ,Thread Name =main
Change the name to MyThread.
After modification ,Thread Name =MyThread
Is the thread alive? true
Thread[MyThread,5,main] i = 0
Is the thread alive? true
Thread[MyThread,5,main] i = 1
Is the thread alive? true
Thread[MyThread,5,main] i = 2
Is the thread alive? true
Thread[MyThread,5,main] i = 3
```

## 5. Creation of New Threads

The new thread is created by creating a new instance of Thread class. The enhancements in Java 8 enables us to create and execute new threads in the following ways

i. By extending Thread class
ii. By implementing Runnable interface: This may be carried out in the following four styles:
    a. Conventional code
    b. Lambda expression
    c. Method reference
    d. Anonymous inner class

### i. Creation of New Thread by Extending Thread Class

A thread is an object of class Thread that contains methods to control the behaviour of threads. **A class** that **extends** the class **Thread** inherits all methods and constructors

The procedure followed in this case is given as follows:

1) A class is declared that extends Thread class. Since this is a subclass, it inherits the methods of Thread class.
2) This class calls a constructor of Thread class in its constructor.
3) In this class, the method run() is defined to override the run() method of Thread class. The method run() contains the code that the thread is expected to execute.

4) The object of class the method start() inherited from Thread class for the execution of run().

**Example: MyClass.java**

```
public class MyClass extends Thread
{
      MyClass()
      {
         super("MyThread"); // constructor of Myclass
         start();
      }

      //The run() method is defined below
      public void run()
      {
        System.out.println("It is MyThread.");
      }

      public static void main(String args[])
      {
          new MyClass(); //creates a new instance of Myclass
      }
}
```

**Output:**

```
C:\>javac MyClass.java

C:\>java MyClass
It is MyThread.
```

### ii. Creation of New Threads by Implementing Runnable interface

The runnable is an interface that has only one method in it, that is

```
public interface Runable( public void run());
```

The full definition of method run() is included in the class. The thread begins with execution of run() and ends with end of run() method. The step wise procedure is given here.

1. Declare a class that implements Runnable interface.
2. Define the run() method in the class. This is the code that the thread will execute.
3. Declare an object of Thread class in main() method.
4. Thread class constructor defines the thread declared with operator new and the Runnable object is passed on to the new thread constructor.
5. Method start() is invoked by the thread object created.

The following Program illustrates a simple example of implementing Runnable.

**Example: MyThreadClass.java**

```
public class MyThreadClass implements Runnable
{
   public void run()
   {
           System.out.println("This is Runnable implementation.");
   }
   public static void main(String args[])
   {
      Thread Th = new Thread(new MyThreadClass());
         Th.start();
   }
}
```

```
Output

This Is Runnable implementation.
```

### iii. Creation of Threads by Lambda Expression, Method Reference, and Anonymous Class

The Lambda expression and method references are simplify the code for creating the thread, as illustrated in the following program.

```
public class ThreadMethods
{
      public static void main(String[] args)
      {
      //Method reference

      new Thread (ThreadMethods::Method1).start();

      //The following line is Lambda expression
      new Thread(() -> Method2()).start();

      //The anonymous inner class or conventional method
```

```
      new Thread(new Runnable()
      {
        public void run()
        { Method3();}
      }).start();

      }

      static void Method1()
      {
        System.out.println("It method reference thread.");
      }
      static void Method2()
      {
        System.out.println("It is Lambda expression method
thread.");
      }
      static void Method3()
      {
        System.out.println("It is conventional method thread.");
      }
}
```

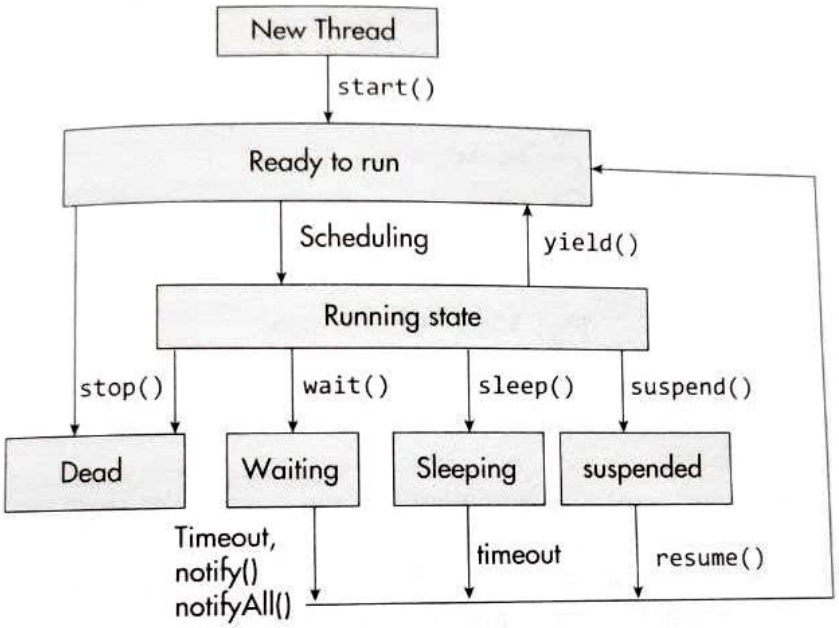
```
Output:
C:\>javac ThreadMethods.java

C:\>java ThreadMethods
It method reference thread.
It is Lambda expression method thread.
It is conventional method thread.
```

## 6. Thread States

Thread states are as follows.

i. New thread state
ii. Ready-to-run state
iii. Running state
iv. Non-runnable state-waiting, sleeping, or blocked state
v. Dead state

The following Figure illustrates the different states of a thread.



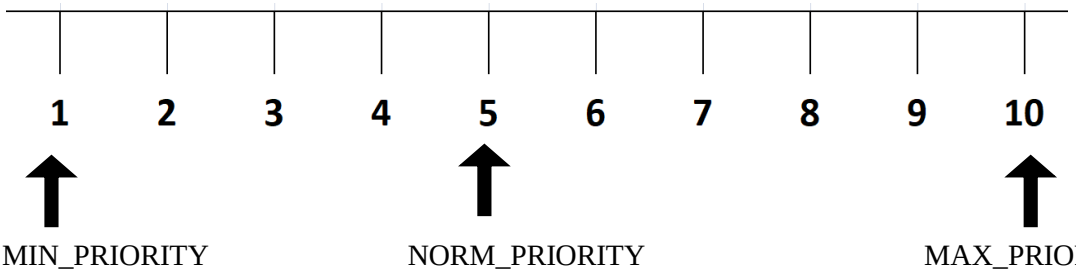

The transition from one state to another is shown in the figure. The different states are as follows

**1. New thread state**. The thread is just created. It is simply a lifeless object of class Thread.
**2. Runnable state:** When the thread invokes the method start(), it becomes alive. This state is called runnable state. It is not a running state. It is simply a ready-to-run. The thread has to wait till it is scheduled, that is, selected from the group of threads waiting in running state for dispatch to the CPU by the scheduler
**3 Running state.** If the scheduler chooses the thread from the waiting list of threads and dispatches it CPU, the thread transits to runnable state, that is, it starts executing the method run() meant for it. This is turning state. If it completes its run() method successfully, its life span is over. The thread is automatically terminated and it goes to **dead state** from which it cannot be revived.
**4. Sleep state:** The code that a thread is executing may require it to relinquish the CPU for sometime so the other threads can possess CPU. The sleep method may be invoked by the thread. The time period of them is the argument of sleep() method. It is either `long milliseconds`
or `int nanoseconds`. After the sleep the thread returns normally to runnable state.
**5. Blocked state.** A running thread gets into blocked state if it attempts to execute a task.
**6. State wait:** A thread gets into wait state on invocation of any of the three methods wait() or wait (long millisees) or wait (long millisecs, int nanosecs).
**7. yield.** The use of code Thread yield(), is another method besides the sleep for the thread to cease the use of CPU. The thread transits to Runnable state.

**8. Suspended.** The term belongs to legacy code and is defined in Java 1.1. The suspended thread can be brought back to normal Runnable state only by method resume(). Both the methods suspend() and
resume() are now deprecated, instead one should use wait()and notify().

## 7. Thread Priority

Thread priority is an important factor among others that helps the scheduler to decide which thread to dispatch the CPU from the group of waiting threads in their runnable state.




All the threads have a priority rating of be When several threads are present, the priority value determines which thread has the chance to possess the Ofest.

The actual allocation is done by the scheduler. Thus, a thread with higher priority has higher chance getting the CPU and also a higher share of CPU time. Keeping equal priority for all threads ensures that each has equal chance to share CPU time, and thus, no thread starves, because when a thread with higher priority enters the Runnable state, the operating system may pre-empt the scheduling by allocating CPU to the thread with higher priority.

When this thread returns from sleep or wait state, the same story may be repeated. When several threads of different priorities are present, it is quite likely that a thread with the lowest priority may not get a chance to possess CPU. This is called starvation.

Thread priority can be changed by method **setPriority (n).**
The priority may be obtained by calling **getPriority()** method.

**Example: ThreadPriority2.java**

```
class PThread extends Thread
{
      String ThreadName;
      Thread Th;
      int P;
      PThread (String Str, int n)
      {
      ThreadName = Str;
      P=n;
      Th = new Thread(this, ThreadName);
      Th.setPriority(P);
      System.out.println("Particulars of new thread:" + Th);
      }
      public void threadStart()
      {
         Th.start();
      }

      public void run()
      {
```

```
      System.out.println("Priority of new thread: " +
Th.getPriority());
          }
}
public class ThreadPriority2
{
  public static void main (String args[])
  {
      PThread PT1 =  new PThread("First", Thread.MAX_PRIORITY);
      PThread     PT2     =           new     PThread("Second",
Thread.NORM_PRIORITY);
      PThread PT3 =  new PThread("Third", 1);

      PT1.threadStart();
      PT2.threadStart();
      PT3.threadStart();
      try
      {
        Thread.sleep(50);
      } catch(InterruptedException e)
      {
      System.out.println("Main thread sleep interrupted");
      }
  }
}
```

**Output:**

```
C:\>javac ThreadPriority2.java

C:\>java ThreadPriority2
Particulars of new thread:Thread[First,10,main]
Particulars of new thread:Thread[Second,5,main]
Particulars of new thread:Thread[Third,1,main]
Priority of new thread: 10
Priority of new thread: 5
Priority of new thread: 1
```

## 8. Synchronization

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

In several multithreaded programs, the different threads are required to access the same resource, for example, a memory object during the execution. One thread may attempt to update it, while the other wants to read it and a third may try to alter the content of the memory.

Such a situation may give rise to **race condition** and the result may be quite different if the sequence of operations actually followed is different from the desired one.

Proper execution requires that at any point of time, only one thread be allowed to use the critical resource till it finishes its task.

We are all aware of the computerized banking system. In one branch, money may be deposited in your account, while you are withdrawing money at another branch. There can be a problem if one thread has partly finished its task, while the other gets in. It will corrupt the data.

Synchronization solves this problem by allowing only one thread can access the resource. The second thread should be allowed only when the first has finished its task.

**Example: SyncDemo.java**

```
class Counter
{
      int count=0;
      public  void increment()
      {
        count++;
      }
}

public class SyncDemo
{
  public static void main(String[] args) throws Exception
  {
      Counter c = new Counter();

      Thread t1 = new Thread(new Runnable()
      {
        public void run()
        {
                  for(int i=1;i<=5000; i++)
            {
               c.increment();
```

```
            }
        }
      });

      Thread t2 = new Thread(new Runnable()
      {
        public void run()
        {
                  for(int i=1;i<=5000; i++)
            {
               c.increment();
            }
        }
      });


            t1.start();
            t2.start();
      t1.join();
      t2.join();

            System.out.println("Count = "+ c.count);
 }
```

**Output:**

```
C:\>javac SyncDemo.java

C:\>java SyncDemo
Count = 8343

C:\>java SyncDemo
Count = 9998

C:\>java SyncDemo
Count = 9865

C:\>java SyncDemo
Count = 9989

C:\>java SyncDemo
Count = 9790

C:\>java SyncDemo
Count = 9954

C:\>java SyncDemo
Count = 9799
```

Here the count should be 10000, but it is printing less than 10000. To solve this problem, add synchronized keyword to the increment() method as shown in the

following code.

```
class Counter
{
      int count=0;
      public synchronized void increment()
      {
        count++;
      }
}
```

**Output: After adding synchronized keyword to increment() method**

```
C:\>javac SyncDemo.java

C:\>java SyncDemo
Count = 10000

C:\>java SyncDemo
Count = 10000

C:\>java SyncDemo
Count = 10000
```

## 9. Deadlock and Race Situations

In a multithreaded program, the race and deadlock conditions are common when improperly synchronized threads have a shared data source as their target.

**A race condition may occur when more than one thread tries to execute the code concurrently.** A thread partly does the job when the second thread enters.

If the process is **atomic**, that is, it cannot be subdivided the effect of race condition will be limited.
However, when the process is **not atomic** and can be subdivided into two or more sub-processes, it may occur that a thread has done subprocess, and the second thread enters and starts executing. In such cases, the results can be far from the desired ones.

Therefore, a race condition is a situation in which two or more threads try to execute code and their actions get interleaved. The solution for such condition is the synchronization, and therefore, only one thread should enter code at a time and othersshould wait until the first has done its job.

In multithreaded programs, the deadlock conditions are also common

**A deadlock condition** may lead to program crash. Such a situation occurs when two threads are attempting to carry out synchronized hods that are inter-dependent, that is, the completion of Method1 needs Method2 and completion of Method2 needs Method1.

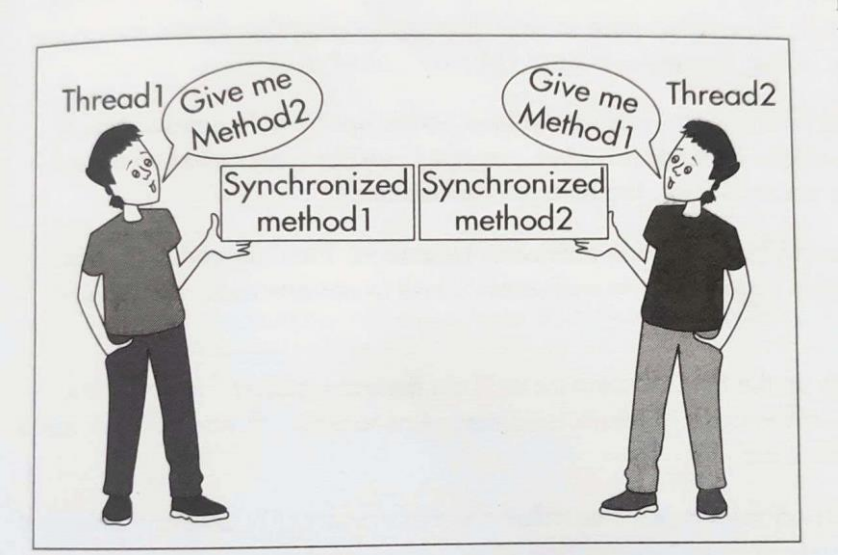



**Fig. 13.8** Example of deadlock situation

## 10.Inter-thread Communication

- Iter-thread communication involves synchronized threads to communicate with each other to pass information. In this mechanism, one of the threads is paused in its critical section to run and another thread is allowed to enter in the same critical section to be executed
- The inter-communication between threads is carried out by three methods, wait(), notify(), and notifyAll(), which can be called within the synchronized context.

**Methods for Inter-Thread communication are as follows**

i. final void wait()
ii. final void wait(long time milliseconds)
iii. final void wait (long time milliseconds, int nanoseconds)
iv. final void notify()
v. final void notifyAll()

**Example: ThreadDemo2.java**

```
class BankAccount
{
      int accountNumber;
      static double balance;

      BankAccount(int n, double y)
      {
         accountNumber = n;
         balance = y;
      }

      synchronized void withDraw(int wd)
      {
            if(balance < wd)
                System.out.println("Less balance " + + balance + " is
available; waiting to deposit more " );
```

```
            if(balance>= wd)
            {
               System.out.println("balance is available : "+ balance);
               balance = balance -wd;
               System.out.println("balance after withdrawal : " +
balance);
            }
            try{
                 wait();
            }catch(Exception e)
            {
            }
            if(balance> wd)
            {
               System.out.println("balance is available : "+ balance);
               balance = balance -wd;
               System.out.println("balance after withdrawal : " +
balance);
            }

      }

      synchronized double deposit(int dp)
      {
            System.out.println("Going to deposit:  " + dp );
            balance = balance + dp;
            System.out.println("Balance after deposit = "+ balance);
            notify();
            return(balance);
      }
}

public class ThreadDemo2
{
      public static void main(String[] args)
      {
      final BankAccount ba = new BankAccount(2345, 1000.0);
      new Thread()
      {
            public void run()
            {
              ba.withDraw(5000);
            }
      }.start();


      new Thread()
      {
            public void run()
            {
            ba.deposit(15000);
            }
      }.start();

      }
}
```

```
C:\ >javac ThreadDemo2.java

C:\ >java ThreadDemo2
Less balance 1000.0 is available; waiting to deposit more
Going to deposit:  15000
Balance after deposit = 16000.0
balance is available : 16000.0
balance after withdrawal : 11000.0
```

## 11.Suspending Resuming and Stopping of Threads.

- In Java 1.1, the following three methods are defined
    - suspend() - pause the operation of thread
    - resume() - resume the operation of thread
    - stop() - to terminate the thread

- These direct methods are very convenient to control the operation of threads in a multithread environment.
- However, in some situations, they may crash the program or cause serious damage to critical data.
- For example, if a thread has got the lock to a critical synchronized section of code and gets suspended, it will not release the lock for which other threads may be waiting.
- Instead of methods suspend() and resume(), the methods wait() and notify() are used.

**Example: ThreadDemo2.java**

# III. Java Database Connectivity

## 1. Introduction

JDBC stands for Java Database Connectivity and has been developed by Sun Microsystems.

It is a standard Java API that defines how the front-end application (that may be web application or simple Java application) may access the database.

It provides a standard library for accessing a wide variety of database systems. Previously, Open Database Connectivity (ODBC) API used to be the database API for connecting and executing query with the database.

JDBC applications are platform independent, and thus, they can be used for connecting with Internet applications. JDBC applications are simpler and easier to develop.

The following Figure illustrates the connectivity model of JDBC.

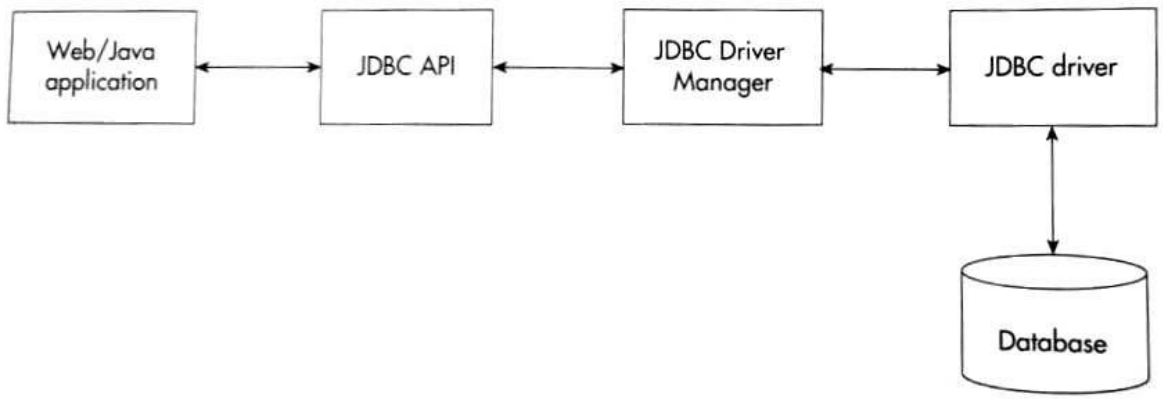



**Fig. 27.1** JDBC connectivity model

JDBC API and JDBC driver form the important components in order to fetch/store the information to the database.

JDBC API is installed at the client side. Therefore, when the user wants to fetch some data from the database, the user sets up the connection to JDBC manager using JDBC API through the Java application.

JDBC manager needs a medium in order to communicate with the database. JDBC driver provides this medium and the required information to JDBC manager, JDBC library includes APIs that define interfaces and classes for writing database applications in Java.

Through the JDBC API, we can access a wide variety of database systems including relational as well as non-relational database system. This chapter focuses on using JDBC to access data in Relational Database

**Relational Database Management System (RDBMS).** It is a database system that is based on relational model. Data in RDBMS is stored in the form of database objects called tables.

Table is a collection of related data entries and consists of columns and rows.

Some of the most widely used relational database systems include Microsoft MySQL server, Oracle, and IBM's DB2.

Any Java-based application can access a relational database. For this, any RDBM system needs to be installed that provides driver conforming to Java Database Connectivity.

JDBC API enables the programmers to change the underlying DBMS (for example, from MySQL to Oracle), without changing the Java code that accesses the database.

## 2. JDBC Architecture

### i. Two-tier Architecture for Data Access

JDBC API supports both two-tier and three-tier processing models for database access. This implies that a Java application can communicate directly with the database or through a middle-tier element.

In this model, Java application communicates directly with the database. For this, JDBC driver is required and it can establish direct communication with the database.

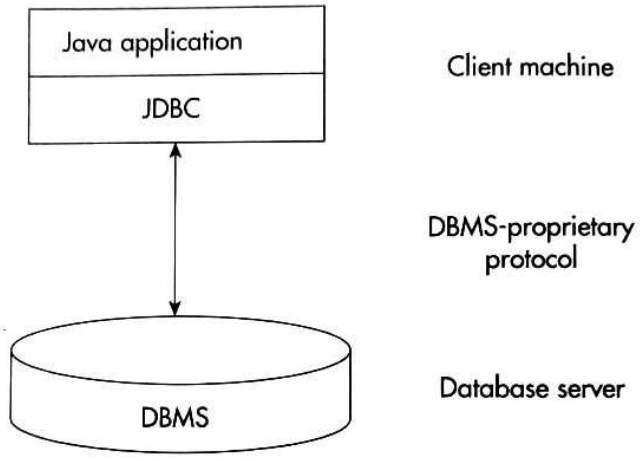



**Fig. 27.2** Two-tier architecture

As can be seen from the figure, both Java application and JDBC API are located at the client machine and the DBMS and database are located at the database server.

User sends commands to the database. The commands processed and the results of these statements are sent to the user.

Java application and the database may reside on the same machine. Alternatively, database may be on the server machine, while the Java application may be on the client machine, which may be connected via the network.

### ii. Three-tier Architecture for Data Access

In this model, user's commands are first sent to the application server forming the middle tier. Application server containing the JDBC API sends the SQL statements to the database located on the database server. The commands are processed and the result is sent to the middle tier, which then sends it to the user.

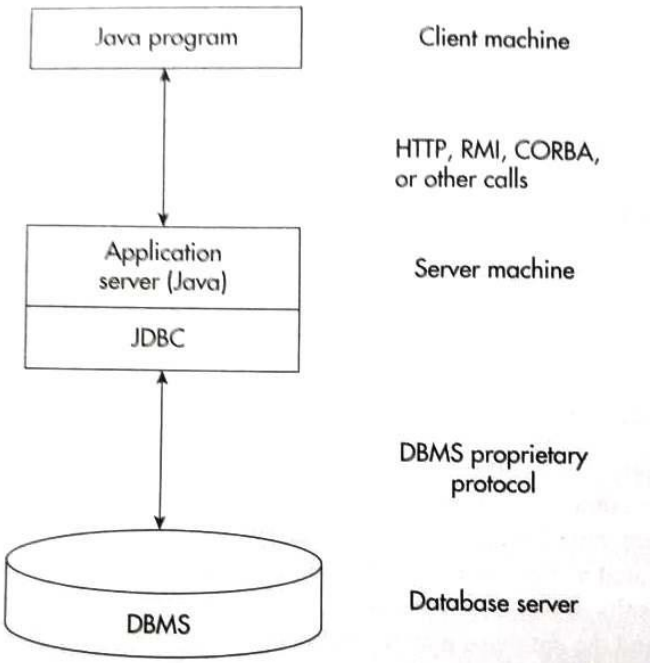



**Fig. 27.3** Three-tier architecture

The above Figure depicts the basic three-tier model.

Middle tier has often been written in languages such as C or C++ that provide the fast performance.

However, Java platform has become the standard platform for middle-tier development with the advancements made in the optimizing compilers. These compilers translate Java byte code into an efficient machine specific code.

This model is usually common in web applications in which the client tier is implemented in the web browser. Web server forms the middle tier and the database management system runs on database server. This model provides better performance and simplifies the deployment of applications.

## 3. Installing MySQL and MySQL Connector

- MySQL is the most widely used open-source relational database management system.
- It is considered to be one of the best RDBMS systems for developing web-based software applications as it provides speed, flexi bility, and reliability.

- Before we can use MySQL with JDBC, we first need to install MySQL.
- **"MySQL community edition"** freely available and can be downloaded from the **MySQL website**

**http://www.mysql.com .**

Steps to install MySQL database system on Windows platform are as follows:

i. On successful download, click the mySQL icon. MySQL Server 5.6 setup. Click on the Next button.
ii. License agreement page would appear next. Read the terms and conditions and click on I accept the wizard window would terms in the license Agreement checkbox.
iii. Next, it will ask for the set up type that suits your needs. Click on Typical button in Choose set up Type screen.
iv. Then click on install button for starting the installation process wizard.
v. Completed MySQL Server 5.6 Setup Wizard would appear. Click on finish to exit the
vi. MySQL Server Instance Configuration screen would appear, and select the Detailed Configuration.
vii. Following this, MySQL Server Instance Configuration Wizard would open up and choose the server as Developer Machine.
viii. Now click on Next button, and select the Dedicated MySQL Server Machine or you may choose Developer Machine if other applications and tools are being run on your system.
ix. Thereafter, select multifunctional database for general purpose database, when MySQL Server Instance Configuration Wizard screen appears. Then, select the drive where the database files would be stored.
x. Keep the default port as 3306 and specify the root password.

After the successful installation of MySQL, we need to install **MySQL Connector/J** (where J stands for Java). **This is the JDBC driver that allows Java applications to interact with MySQL.**

MySQL connector/J can be downloaded from

**dev.mysql.com/downloads/connector/j/3,1.html.**

## SQL Statements

SQL statements are used for performing various actions on the database.

i. **SQL select statements :**
The select statements are used to select data from the database.

**Syntax :**

```
select column_name1,  column_name2  from  tablename;
```

**Example-1:**

```
select id, stdName from Student;
```

Here :

id and stdName are the column names

Student is the table name;

**Example-2:**

```
Select * from Student;
```

The above statement selects all the columns from Student table.

### ii. SQL insert into Statement:

SQL insert into statement It is used to insert new records in a table.

```
insert into tablename values (valuel, value2, value3_);
```

Alternatively, we can also write,

```
insert into tablename (calumn1, column2...)
       Values (valuel, value2; value3.);
```

For instance,

```
insert into Student values (Riya, Sharma, 60);
```

### iii. SQL where clause

SQL where clause It is used to extract only those records that satisfy a particular criterion.
The syntax for this statement is:

```
select column_namel, column_name2 from table_name
where column_name operator value;

select from * Student where id=347;
```

### iv. SQL delete statement

SQL delete statement It is used to delete the records in the table.
The syntax for this statement

```
delete from table_name where column_name = value;
```

example:

```
delete from Student where firstname="Riya";
```

## 4. JDBC Environment Setup

This section focuses on how to set up the connection to MySQL database from NetBeans IDE. NetBeans IDE supports MySQL RDBMS. In order to access MySQL database server in NetBeans IDE, we have to first configure the MySQL Server properties. This involves the following steps:

1. Open the NetBeans IDE, right click the database node in the services window. Select Register MySQL Server MySQL server properties dialog box would open. Confirm that the server host name and port are correct. The default server host name is localhost and 3306 is the default server port name.

2. Enter the administrator password. You can give any password and default is set to blank password.

3. At the top of dialog box, click on the Admin properties tab. Here, you can enter the information for controlling the MySQL server.

4. In the admin properties dialog box, enter the path/URL to admin tool. This is the location of MySQL Administration.

5. Next, you have to enter the path to start the command. For this, look for mysqld in the bin folder MySQL installation directory of

6. In the path to stop the command field, type or browse to the location of MySQL stop command. This is the path where mysqladmin the bin folder of MySQL installation directory is located.

Additional steps that must be checked before starting with the involvement of Java-based database connectivity:

1. Before starting, make sure that MySQL Database server is running. If database server is not connected, it will show 'disconnected'. For connecting it, right click the Database, and choose 'connect''. This may also prompt to give password to connect to the database server

2. When a new project is created in NetBeans, copy the mysql-connector/java JAR file into the library folder.

**JDBC Connectivity Model and API**

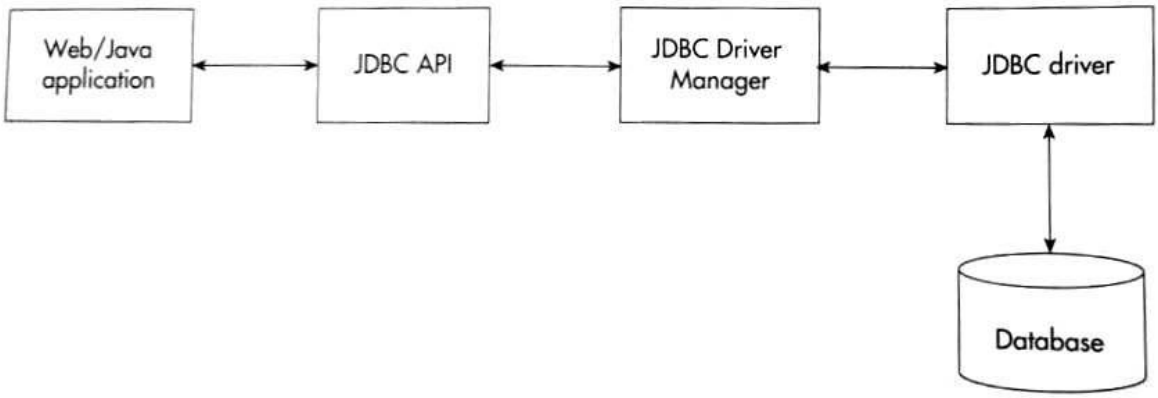



**Fig. 27.1** JDBC connectivity model

Fig. 27.1 depicts the JDBC connectivity model. JDBC API enables Java applications to be connected to relational databases through standard API. This makes possible for the user to establish a connection to a database, create SQL or MySQL statements, execute queries in the database, and so on. JDBC API comprises the following interfaces and classes:

**Driver manager**

This class manages a list of database drivers. It matches the connection request from the Java application with the database driver using communication sub-protocol. **It acts as an interface between the user and the driver and is used to get a Connection object.** Commonly used methods of this class are as follows

Method

**`Connection getConnection(String url)`**:
It is used to establish the connection with the specified URL

**`Connection getConnection(String url, String username, String password)`**
It is used to establish the connection with the specified URL, username, and password

# Driver

It handles communication with the database server. JDBC drivers are written in Java language in order to connect with the database. JDBC driver is a software component comprising a set of Java classes that provides linkage between Java program running on Java platform and RDBM system that is residing on the operating system.

**There are basically four different types of JDBC drivers and these implementations** vary because of the wide variety of operating systems and hardware platforms available in which Java operates

## Type 1 JDBC-ODBC bridge driver

Type 1 JDBC driver provides a standard API for accessing SQL on Windows platform. In this type of the driver, JDBC bridge is used to access ODBC drivers installed on the client machine. For using ODBC, Data Source Name (DSN) on the client machine is required to be configured.
The driver converts JDBC interface calls into ODBC calls. It is, therefore, the least efficient driver of the four types. These drivers were mostly used in the beginning and now it is usually used for experimental purposes when no other alternative is available

## Type 2 driver (also known as Native API driver)

In Type 2 driver, Java interface for vendor-specific API is provided and it is implemented in native code. It includes a set of Java classes that make use of **Java Native Interface (JNI)** and acts as bridge between Java and the native code.

JNI is a standard programming interface that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (these include the programs that are specific to a particular hardware and operating system like C/C++).

Thus, the driver converts JDBC method calls into native calls of the database API. For using this driver, it is required that RDBMS system must reside in the same host as the client program.

The Type 2 driver provides more functionality and performance than Type 1 driver. For using this driver in a distributed environment, it is required that all the classes that operate on the database should reside on the database host system.

## Type 3 driver (also known as Network-Protocol driver)

It is similar to Type 2 driver but in this case, the user accesses the database through TCP/IP connection. The driver sends JDBC interface calls to an inter mediate server, which then connects to the database on behalf of the JDBC driver.

Type 3 and 4 drivers are preferably used if the program application does not exist on the same host as the database. It requires data base-specific coding to be done in the middle tier.

## Type 4 driver (also known as Native-Protocol driver)

Type 4 JDBC driver is completely written in Java, and thus, it is platform independent. The driver converts JDBC calls directly into vendor-specific database protocol.

It is installed inside the Java Virtual Machine of the client and most of the JDBC drivers are of Type 4.

It provides better performance when compared to Type 1 and 2 drivers as it does not have the over head of conversion of calls into ODBC or database API calls, However, at the client side, a separate driver is required for each database.

## Packages:

There are two packages that make up the JDBC API.
They are

i. **java.sql**
ii. **javax.sql**

iii. java. sql package provides API for accessing and processing data that is stored in a data source. This API comprises framework whereby different drivers can be installed dynamically in order to access different data sources. For instance, it comprises API for establishing a connection with a database via the Driver manager facility, sending SQL statements a database, retrieving and updating the results of the query, and so on.

iv. javax.sql package provides the required API for server-side data source access and processing. This package supplements the java.sql package. It is included in the Java Platform Standard Edition (Java SETM).

## Connection:

This interface comprises methods for making a connection with the database. All types of communication with the database is carried out through the connection object only.

## Statement

Object created from this interface is used to submit the SQL statements to the database.

## ResultSet

It acts as an iterator that enables to move through the data. Object created from interface is used to data received from the database after executing SQL query.

### SQL Exception

This class handles errors that may occur in a database application.