

# **MODULE 4**

## **ADVANCED FEATURES OF JAVA**

### **CHAPTER 1**

#### **Java Library & Collections framework**

# STRING

In Java, string is basically **an object** that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[ ] ch={'h','a','i','j','a','v','a'};
```

```
String s=new String(ch);
```

**same as:**

```
String s = "haijava";
```

Java **String** class provides a lot of methods to perform operation on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

# Create a string object

There are two ways to create String object:

By string literal

By new keyword

## String Literal

A String literal is created by using double quotes. For Example:

```
String s = "welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first.

If the string already exists in the pool, a reference to the pooled instance is returned.

If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
```

```
String s2="Welcome"; //It doesn't create a new instance
```

In the above example, only one object will be created.

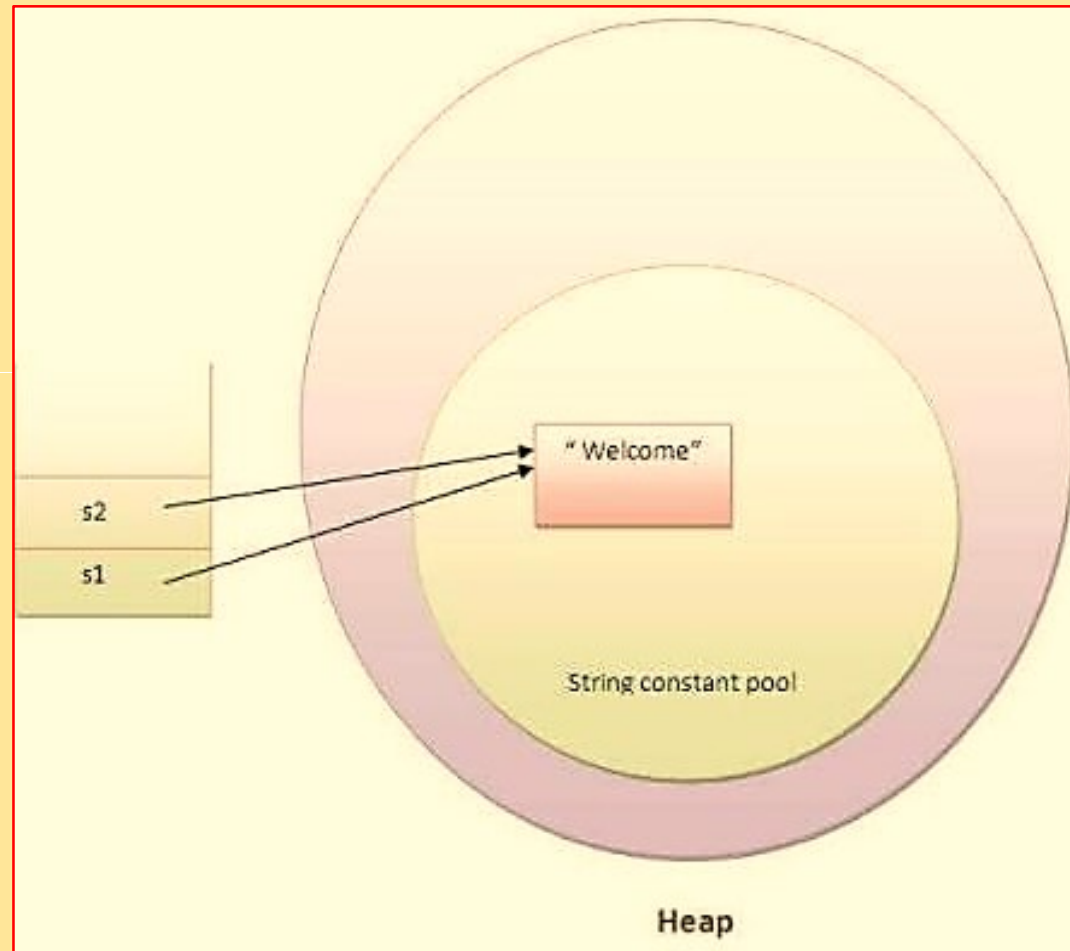
Firstly, JVM will not find any **string object with the value "Welcome"** in string constant pool, that is why it will create a new object.

After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

**Note:** String objects are stored in a special memory area known as the **"string constant pool"**.

## Why Java uses the concept of String literal

make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).



## By new keyword

`String s=new String("Welcome");` //creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.

The variable s will refer to the object in a heap (non-pool).

## String Example

```
public class StringExample{  
    public static void main(String args[]){  
        String s1="java";//creating string by java string literal  
        char ch[]={'s','t','r','i','n','g','s'};  
        String s2=new String(ch);//converting char array to string  
        String s3=new String("example");//creating java string by new keyword  
        System.out.println(s1);  
        System.out.println(s2);  
        System.out.println(s3);  
    }  
}
```

### OUTPUT

```
java  
strings  
example
```

# STRING CONSTRUCTORS

The string class supports several types of constructors in Java API. The most commonly used constructors of String class are as follows:

**String()** : To create an empty String, we will call a default constructor. For example:

```
String s = new String();
```

It will create a string object in the heap area with no value.



**String(String str)** : It will create a string object in the heap area and stores the given value in it. For example:

```
String s2 = new String("Hello Java");
```

Now, the object contains Hello Java.

**String(char chars[ ])** : It will create a string object and stores the array of characters in it. For example:

```
char chars[ ] = { 'a', 'b', 'c', 'd' };
```

```
String s3 = new String(chars);
```

The object reference variable s3 contains the address of the value stored in the heap area.

Let's take an example program where we will create a string object and store an array of characters in it

```
package stringPrograms;  
  
public class Science  
{  
    public static void main(String[] args)  
    {  
        char chars[] = { 's', 'c', 'i', 'e', 'n', 'c', 'e' };  
        String s = new String(chars);  
        System.out.println(s);  
    }  
}
```

Output:

science

## **String(char chars[ ], int startIndex, int count)**

It will create and initialize a string object with a subrange of character array.

The argument **startIndex** specifies the index at which the subrange begins and **count** specifies the number of characters to be copied.

For example:

```
char chars[ ] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' };
```

```
String str = new String(chars, 2, 3);
```

The object **str** contains the address of the value "ndo" stored in the heap area because the starting index is 2 and the total number of characters to be copied is 3.

## KAMPLE

```
package stringPrograms;
public class Windows
{
    public static void main(String[] args)
    {
        char chars[] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' };
        String s = new String(chars, 0,4);
        System.out.println(s);
    }
}
```

Output:

wind

In this example program, we will construct a String object that contains the same characters sequence as another string object.

```
package stringPrograms;
public class MakeString
{
    public static void main(String[] args)
    {
        char chars[] = { 'F', 'A', 'N' };
        String s1 = new String(chars);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output:

FAN

FAN

As you can see the output, s1 and s2 contain the same string. Thus, we can create one string from another string.

**String(byte byteArray[ ])** : It constructs a new string object by decoding the given array of bytes (i.e, by decoding ASCII values into characters) according to the system's default character set.

```
package stringPrograms;
public class ByteArray
{
    public static void main(String[] args)
    {
        byte b[] = { 97, 98, 99, 100 }; // Range of bytes: -128 to 127. These byte
        values will be converted into corresponding characters.
        String s = new String(b);
        System.out.println(s);
    }
}
```

Output:

abcd

## **String(byte byteArr[ ], int startIndex, int count)**

This constructor also creates a new string object by decoding the ASCII values using the system's default character set.

```
package stringPrograms;  
public class ByteArray  
{  
    public static void main(String[] args)  
    {  
        byte b[] = { 65, 66, 67, 68, 69, 70 }; // Range of bytes: -128 to 127.  
        String s = new String(b, 2, 4); // CDEF  
        System.out.println(s);  
    }  
}
```

Output:

CDEF

# STRING LENGTH

The **java string length()** method gives length of the string. It returns count of total number of characters.

## Internal implementation

```
public int length() {  
    return value.length;  
}
```

**Signature** - The signature of the string length() method is given below:

```
public int length()
```



## String length() method example - 1

```
public class LengthExample{  
    public static void main(String args[]){  
        String s1="javatpoint";  
        String s2="python";  
        System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string  
        System.out.println("string length is: "+s2.length());//6 is the length of python string  
    }  
}
```

### Output

```
string length is: 10  
string length is: 6
```

## String length() method example - 2

```
public class LengthExample2 {  
    public static void main(String[] args) {  
        String str = "Javatpoint";  
        if(str.length() > 0) {  
            System.out.println("String is not empty and length is: "+str.length());  
        }  
        str = "";  
        if(str.length() == 0) {  
            System.out.println("String is empty now: "+str.length());  
        }  
    }  
}
```

### Output

```
String is not empty and length is: 10  
String is empty now: 0
```

# STRING COMPARISON

We can compare string in java on the basis of content and reference

There are three ways to compare string in java:

- by **equals()** method

- by **==** operator

- by **compareTo()** method

## String compare by equals() method

The String equals() method compares the original content of the string.

It compares values of string for equality. String class provides two methods

**public boolean equals(Object another)** compares this string to the specified object.

**public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

```
class Teststringcomparison1{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        String s4="Saurav";  
        System.out.println(s1.equals(s2));//true  
        System.out.println(s1.equals(s3));//true  
        System.out.println(s1.equals(s4));//false  
    }  
}
```

```
Output:true  
true  
false
```

## Sample 2

```
class Teststringcomparison2{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="SACHIN";  
  
        System.out.println(s1.equals(s2));//false  
        System.out.println(s1.equalsIgnoreCase(s2));//true  
    }  
}
```

Output

```
false  
true
```

# String compare by == operator

The == operator compares references not values.

```
class Teststringcomparison3{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        System.out.println(s1==s2);//true (because both refer to same instance)  
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)  
    }  
}
```

```
Output:true  
        false
```

## String compare by compareTo() method

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

$s1 == s2$  : 0

$s1 > s2$  : positive value

$s1 < s2$  : negative value

```
class Teststringcomparison4{  
  public static void main(String args[]){  
    String s1="Sachin";  
    String s2="Sachin";  
    String s3="Ratan";  
    System.out.println(s1.compareTo(s2));//0  
    System.out.println(s1.compareTo(s3));//1(because s1>s3)  
    System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )  
  }  
}
```

Output:0

1

-1



g:

```
public class CompareToExample{  
    public static void main(String args[]){  
        String s1="hello";  
        String s2="hello";  
        String s3="meklo";  
        String s4="hemlo";  
        String s5="flag";  
  
        System.out.println(s1.compareTo(s2));//0 because both are equal  
        System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"  
        System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"  
        System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"  
    }  
}
```

## Output

```
0  
-5  
-1  
2
```

# SEARCHING STRINGS

## string contains()

The java string `contains()` method searches the sequence of characters in this string.

It returns true if sequence of char values are found in this string, otherwise returns false.

## Internal implementation

```
public boolean contains(CharSequence s) {  
    return indexOf(s.toString()) > -1;  
}
```

# Signature

The signature of string contains() method is given below:

`public boolean contains(CharSequence sequence)`

```
class ContainsExample{  
    public static void main(String args[]){  
        String name="what do you know about me";  
        System.out.println(name.contains("do you know"));  
        System.out.println(name.contains("about"));  
        System.out.println(name.contains("hello"));  
    }  
}
```

## Output

```
true  
true  
false
```

**g 2** - The contains() method searches case sensitive char sequence. If the argument is not case sensitive, it returns false. Let's see an example below.

```
public class ContainsExample2 {  
    public static void main(String[] args) {  
        String str = "Hello Javatpoint readers";  
        boolean isContains = str.contains("Javatpoint");  
        System.out.println(isContains);  
        // Case Sensitive  
        System.out.println(str.contains("javatpoint")); // false  
    }  
}
```

Output

```
true  
false
```

**g 3** -The contains() method is helpful to find a char-sequence in the string. We can use it in control structure to produce search based result. Let us see an example below.

```
public class ContainsExample3 {  
    public static void main(String[] args) {  
        String str = "To learn Java visit Javatpoint.com";  
        if(str.contains("Javatpoint.com")) {  
            System.out.println("This string contains javatpoint.com");  
        }else  
            System.out.println("Result not found");  
    }  
}
```

Output:

This string contains javatpoint.com

# CHARACTER EXTRACTION

## ♦String charAt()

The **java string charAt()** method returns a char value at the given index number.

The index number starts from 0 and goes to n-1, where n is length of the string.

It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

**Signature** - The signature of string charAt() method is given below

```
public char charAt(int index)
```

Example:

```
public class CharAtExample{  
    public static void main(String args[]){  
        String name="javatpoint";  
        char ch=name.charAt(4);//returns the char value at the 4th index  
        System.out.println(ch);  
    }  
}
```

Output

t

## ◆ StringIndexOutOfBoundsException with charAt()

Let's see the example of charAt() method where we are passing greater index value.

In such case, it throws StringIndexOutOfBoundsException at runtime.

```
public class CharAtExample{  
    public static void main(String args[]){  
        String name="javatpoint";  
        char ch=name.charAt(10);//returns the char value at the 10th index  
        System.out.println(ch);  
    }  
}
```



Output:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:  
String index out of range: 10  
at java.lang.String.charAt(String.java:658)  
at CharAtExample.main(CharAtExample.java:4)
```

## ♦ Java String charAt() Example 3

Let's see a simple example where we are accessing first and last character from the provided string.

```
public class CharAtExample3 {  
    public static void main(String[] args) {  
        String str = "Welcome to Javatpoint portal";  
        int strLength = str.length();  
        // Fetching first character  
        System.out.println("Character at 0 index is: "+ str.charAt(0));  
        // The last Character is present at the string length-1 index  
        System.out.println("Character at last index is: "+ str.charAt(strLength-1));  
    }  
}
```

Output:

```
Character at 0 index is: W  
Character at last index is: l
```

## ♦ Java String charAt() Example 4

Let's see an example where we are accessing all the elements present at odd index.

```
public class CharAtExample4 {  
    public static void main(String[] args) {  
        String str = "Welcome to Javatpoint portal";  
        for (int i=0; i<=str.length()-1; i++) {  
            if(i%2!=0) {  
                System.out.println("Char at "+i+" place "+str.charAt(i));  
            }  
        }  
    }  
}
```

Output:

```
Char at 1 place e  
Char at 3 place c  
Char at 5 place m  
Char at 7 place  
Char at 9 place o  
Char at 11 place J  
Char at 13 place v  
Char at 15 place t  
Char at 17 place o  
Char at 19 place n  
Char at 21 place  
Char at 23 place o  
Char at 25 place t  
Char at 27 place l
```

## ♦ Java String charAt() Example 5

Let's see an example where we are counting frequency of character in the string.

```
public class CharAtExample5 {  
    public static void main(String[] args) {  
        String str = "Welcome to Javatpoint portal";  
        int count = 0;  
        for (int i=0; i<=str.length()-1; i++) {  
            if(str.charAt(i) == 't') {  
                count++;  
            }  
        }  
        System.out.println("Frequency of t is: "+count);  
    }  
}
```

Output:

Frequency of t is: 4

# MODIFY STRINGS

The **java string replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

## Signature

There are two type of replace methods in java string.

```
public String replace(char oldChar, char newChar)
```

and

```
public String replace(CharSequence target, CharSequence  
replacement)
```

The second replace method is added since JDK 1.5.

## String replace(char old, char new) method example

```
public class ReplaceExample1{  
    public static void main(String args[]){  
        String s1="java is a very good language";  
        // replaces all occurrences of 'a' to 'e'  
        String replaceString=s1.replace('a','e');  
        System.out.println(replaceString);  
    }  
}
```

### Output

jeve is e very good lengluege

## String replace(CharSequence target, CharSequence replacement) method example

```
public class ReplaceExample2{  
    public static void main(String args[]){  
        String s1="my name is khan my name is java";  
        String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"  
        System.out.println(replaceString);  
    }  
}
```

### Output

my name was khan my name was java

## String replace() Method Example 3

```
public class ReplaceExample3 {  
    public static void main(String[] args) {  
        String str = "oooooo-hhhh-oooooo";  
        String rs = str.replace("h","s"); // Replace 'h' with 's'  
        System.out.println(rs);  
        rs = rs.replace("s","h"); // Replace 's' with 'h'  
        System.out.println(rs);  
    }  
}
```

Output

```
ooooooo-ssss-ooooooo  
ooooooo-hhhh-ooooooo
```



The **java string replaceAll()** method returns a string replacing all sequence of characters matching regex and replacement string.

### Internal implementation

```
public String replaceAll(String regex, String replacement) {  
    return Pattern.compile(regex).matcher(this).replaceAll(replacement);  
}
```

### Signature

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

## String replaceAll() example: replace character

Let's see an example to replace all the occurrences of a single character.

```
public class ReplaceAllExample1{  
    public static void main(String args[]){  
        String s1="java is a very good language";  
        String replaceString=s1.replaceAll("a","e");//replaces all occurrences  
                                                    of "a" to "e"  
        System.out.println(replaceString);  
    }  
}
```

**Output**    jeve is e very good lenege

## String replaceAll() example: replace word

Let's see an example to replace all the occurrences of single word or set of words.

```
public class ReplaceAllExample2{  
    public static void main(String args[]){  
        String s1="My name is Khan. My name is Bob. My name is Sonoo.";  
        String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"  
        System.out.println(replaceString);  
    }  
}
```

## Output

My name was Khan. My name was Bob. My name was Sonoo.

## String replaceAll() example: remove white spaces

Let's see an example to remove all the occurrences of white space

```
public class ReplaceAllExample3{  
    public static void main(String args[]){  
        String s1="My name is Khan. My name is Bob. My name is Sonoo.";  
        String replaceString=s1.replaceAll("\\s","");  
        System.out.println(replaceString);  
    }  
}
```

Output

MynameisKhan.MynameisBob.MynameisSonoo.

# STRING VALUE OF ( )

The java string **valueOf()** method converts different types of value into string.

By the help of string valueOf() method, we can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

## internal implementation

```
public static String valueOf(Object obj) {  
    return (obj == null) ? "null" : obj.toString();  
}
```

## ► Signature

The signature or syntax of string valueOf() method is given below

```
public static String valueOf(boolean b)
```

```
public static String valueOf(char c)
```

```
public static String valueOf(char[] c)
```

```
public static String valueOf(int i)
```

```
public static String valueOf(long l)
```

```
public static String valueOf(float f)
```

```
public static String valueOf(double d)
```

```
public static String valueOf(Object o)
```

## ◆ valueOf() method example

```
public class StringValueOfExample{  
    public static void main(String args[]){  
        int value=30;  
        String s1=String.valueOf(value);  
        System.out.println(s1+10);//concatenating string with 10  
    }}
```

Output

3010

## ◆ **valueOf(boolean bol)** Method Example

This is a boolean version of overloaded valueOf() method. It takes a boolean value and returns a string. Let's see an example.

```
public class StringValueOfExample2 {  
    public static void main(String[] args) {  
        // Boolean to String  
        boolean bol = true;  
        boolean bol2 = false;  
        String s1 = String.valueOf(bol);  
        String s2 = String.valueOf(bol2);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

**Output**

true

false



## ◆ **valueOf(char ch)** Method Example

This is a char version of overloaded valueOf() method. It takes a char value and returns a string. Let's see an example.

```
public class StringValueOfExample3 {  
    public static void main(String[] args) {  
        // char to String  
        char ch1 = 'A';  
        char ch2 = 'B';  
        String s1 = String.valueOf(ch1);  
        String s2 = String.valueOf(ch2);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Output

A

B

## ◆ **valueOf(float f)** and **valueOf(double d)** Example

This is a float version of overloaded valueOf() method. It takes a float value and returns a string. Let's see an example.

```
public class StringValueOfExample4 {  
    public static void main(String[] args) {  
        // Float and Double to String  
        float f = 10.05f;  
        double d = 10.02;  
        String s1 = String.valueOf(f);  
        String s2 = String.valueOf(d);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

### Output

10.05

10.02

# String.valueOf() Complete Examples

```
class StringValueOfExample5 {  
    public static void main(String[] args) {  
        boolean b1=true;  
        byte b2=11;  
        short sh = 12;  
        int i = 13;  
        long l = 14L;  
        float f = 15.5f;  
        double d = 16.5d;  
        char chr[]={ 'j','a','v','a' };  
        StringValueOfExample5 obj=new StringValueOfExample5();  
        String s1 = String.valueOf(b1);  
        String s2 = String.valueOf(b2);  
        String s3 = String.valueOf(sh);  
        String s4 = String.valueOf(i);  
        String s5 = String.valueOf(l);  
        String s6 = String.valueOf(f);  
        String s7 = String.valueOf(d);  
        String s8 = String.valueOf(chr);  
        String s9 = String.valueOf(obj);  
    }  
}
```

```
System.out.println(s1);  
System.out.println(s2);  
System.out.println(s3);  
System.out.println(s4);  
System.out.println(s5);  
System.out.println(s6);  
System.out.println(s7);  
System.out.println(s8);  
System.out.println(s9);
```

```
}
```

```
}
```

## Output

```
true  
11  
12  
13  
14  
15.5  
16.5  
java  
StringValueOfExample5@2a139
```

# Immutable String in Java

In java, **string objects are immutable**. Immutable simply means **unmodifiable** or **unchangeable**. Once string object is created its data or state can't be changed but a new string object is created.

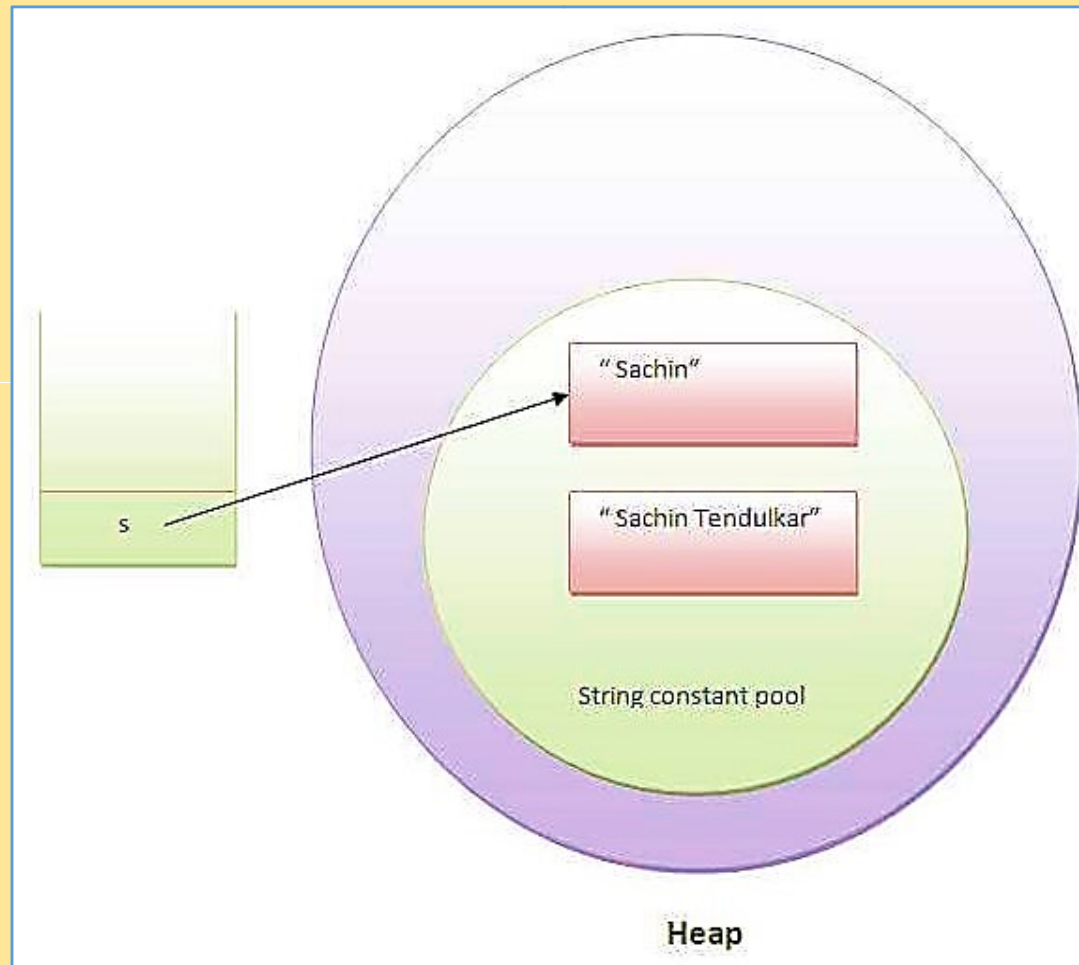
**Example**

```
class Testimmutablestring{  
    public static void main(String args[]){  
        String s="Sachin";  
        s.concat(" Tendulkar");//concat() method appends the string at the end  
        System.out.println(s);//will print Sachin because strings are immutable objects  
    }  
}
```

**Output**

Sachin

can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That any string is known as immutable.



As you can see in the figure that two objects are created but reference variable still refers to "Sachin" not to "Sachin Tendulkar". But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object. For example:

```
class TestImmutableString1{  
    public static void main(String args[]){  
        String s="Sachin";  
        s=s.concat(" Tendulkar");  
        System.out.println(s);  
    }  
}
```

**Output**

Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

## ❖ Why string objects are immutable in java

Because java uses the concept of string literal.

Suppose there are 5 reference variables, all refer to one object "sachin".

If one reference variable changes the value of the object, it will be affected to all the reference variables.

That is why string objects are immutable in java.

# String and StringBuffer

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Java **StringBuffer class** is used to create **mutable** (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.



## ◆ Important Constructors of StringBuffer class

Constructor	Description
<code>StringBuffer()</code>	creates an empty string buffer with the initial capacity of 16.
<code>StringBuffer(String str)</code>	creates a string buffer with the specified string.
<code>StringBuffer(int capacity)</code>	creates an empty string buffer with the specified capacity as length.

**Mutable string** - A string that can be modified or changed is known as mutable string. `StringBuffer` and `StringBuilder` classes are used for creating mutable string.

## ◆ StringBuffer append() method

```
class StringBufferExample{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.append("Java");//now original string is changed  
        System.out.println(sb);//prints Hello Java  
    }  
}
```

## ► StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.insert(1,"Java");//now original string is changed  
        System.out.println(sb);//prints HJavaello  
    }  
}
```

## ► StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.replace(1,3,"Java");  
        System.out.println(sb);//prints HJavallo  
    }  
}
```

## ► StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.delete(1,3);  
        System.out.println(sb);//prints Hlo  
    }  
}
```

## ► StringBuffer reverse() method

The reverse() method of StringBuffer class reverses the current string.

```
class StringBufferExample5{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.reverse();  
        System.out.println(sb);//prints olleH  
    }  
}
```

# COLLECTIONS IN JAVA

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many **interfaces** (Set, List, Queue, Deque) and **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

**Collection in Java** - Represents a single unit of objects, i.e., a group.

## **framework in Java**

It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

## **Collection framework**

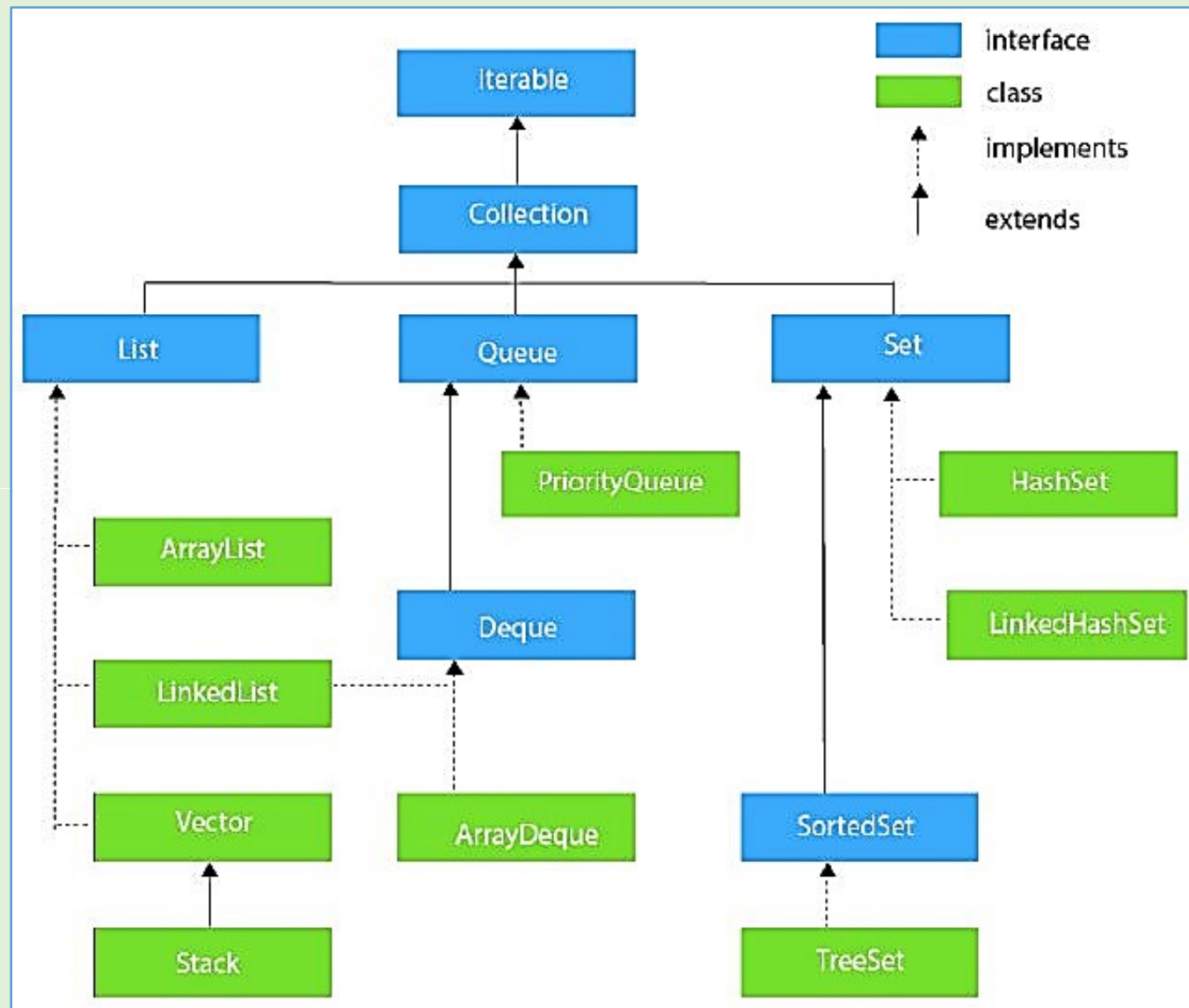
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

Interfaces and its implementations, i.e., classes

Algorithm



# Hierarchy of Collection Framework



The **java.util** package contains all the **classes** and **interfaces** for the Collection framework.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework.

It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add(Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

# LIST INTERFACE

List interface is the child interface of Collection interface.

It inhibits a list type data structure in which we can store the ordered collection of objects.

It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();  
List <data-type> list2 = new LinkedList();  
List <data-type> list3 = new Vector();  
List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## **ArrayList**

The ArrayList class implements the List interface. It uses a **dynamically array** to store the duplicate element of different data types.

The ArrayList class maintains the insertion order and is not synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;

class TestJavaCollection1{

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>(); //Creating arraylist

list.add("Ravi"); //Adding object in arraylist

list.add("Vijay");

list.add("Ravi");

list.add("Ajay");

//Traversing list through Iterator

Iterator itr=list.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Java ArrayList class uses a **dynamic array** for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime.

So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

The **ArrayList in Java can have the duplicate elements also.** It implements the List interface so we can use all the methods of List interface here.

The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about Java ArrayList class are:

Java ArrayList class can contain duplicate elements.

Java ArrayList class maintains insertion order.

Java ArrayList class is non synchronized.

Java ArrayList allows random access because array works at the index basis.

In ArrayList, manipulation is little bit slower than the LinkedList Java because a lot of shifting needs to occur if any element is removed from the array list.

# ArrayList Example

```
import java.util.*;  
  
public class ArrayListExample1{  
  
    public static void main(String args[]){  
  
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist  
        list.add("Mango");//Adding object in arraylist  
        list.add("Apple");  
        list.add("Banana");  
        list.add("Grapes");  
  
        //Printing the arraylist object  
        System.out.println(list);  
    }  
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```



# Iterating ArrayList using Iterator

```
import java.util.*;

public class ArrayListExample2{

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>();//Creating arraylist

list.add("Mango");//Adding object in arraylist

list.add("Apple");

list.add("Banana");

list.add("Grapes");

//Traversing list through Iterator

Iterator itr=list.iterator();//getting the Iterator

while(itr.hasNext()){//check if iterator has the elements

System.out.println(itr.next());//printing the element and move to next

}

}
```

## Output:

```
Mango
Apple
Banana
Grapes
```

# **MODULE 4**

## **CHAPTER 2**

# **MULTITHREADED PROGRAMMING**

# THREAD

JAVA is a multi-threaded programming language which means we can develop multi-threaded program using Java.

A multi-threaded program contains **two or more parts** that can run concurrently and each part can **handle a different task** at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

**Each part of such program is called a thread.** So, threads are **light weight processes** within a process.

Multiprocessing and multithreading, both are used to achieve **multitasking**. But we use multithreading than multiprocessing because **threads share a common memory area**.

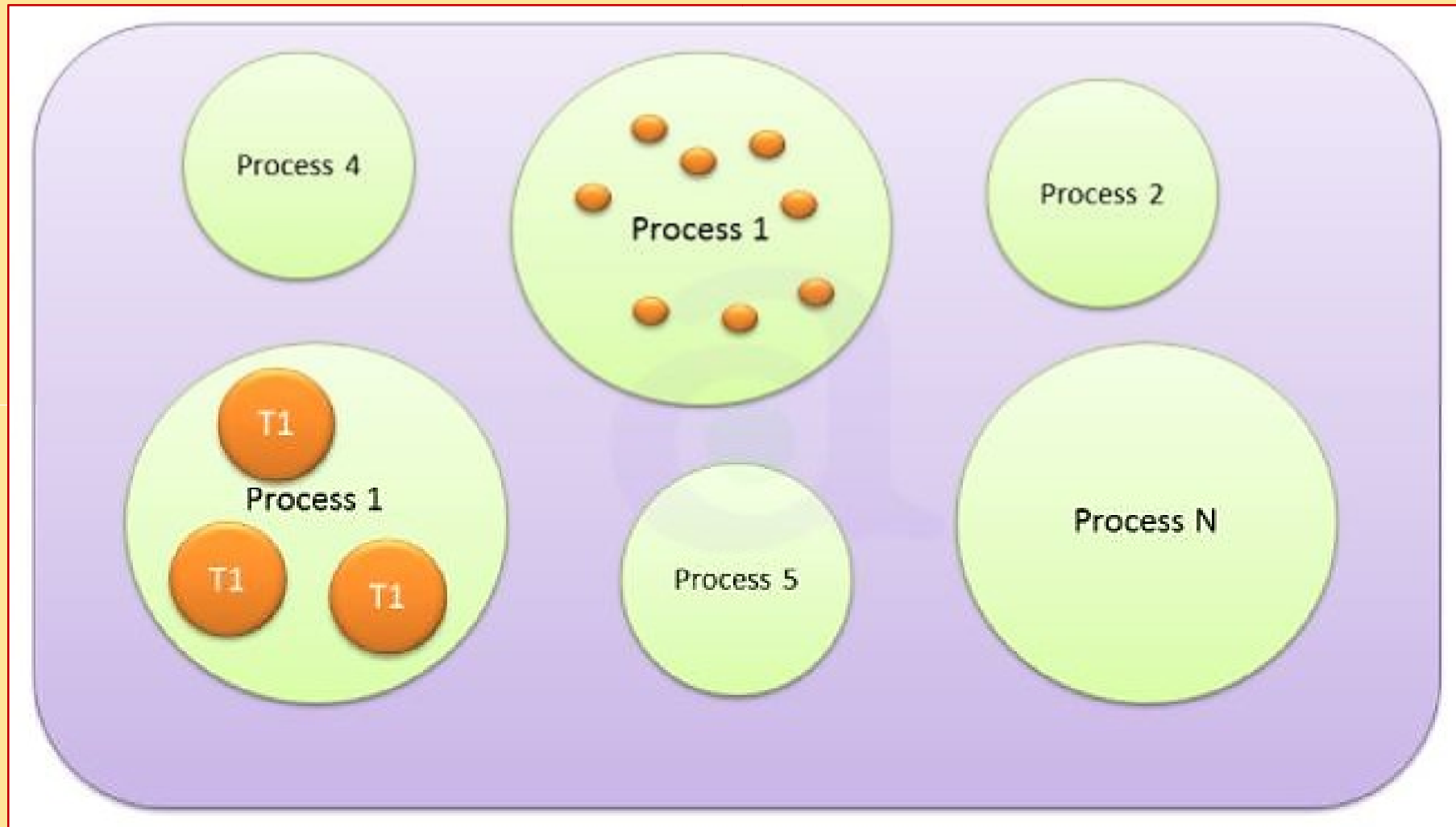
They don't allocate separate memory area so **saves memory**, and **context-switching** between the threads takes less time than process.

Java Multithreading is mostly used in **games, animation** etc..

A thread is a **lightweight sub process**, a smallest unit of processing. It is a **separate path of execution**.

They are **independent**, if there occurs exception in one thread, it doesn't affect other threads.

*At least one process is required for each thread.*



## Advantages of Java Multithreading

It doesn't block the user because threads are independent and you can perform multiple operations at same time.

You can perform many operations together so it saves time.

Threads are independent so it doesn't affect other threads and exception occur in a single thread.

Note: At a time one thread is executed only.

# LIFE CYCLE OF THREAD

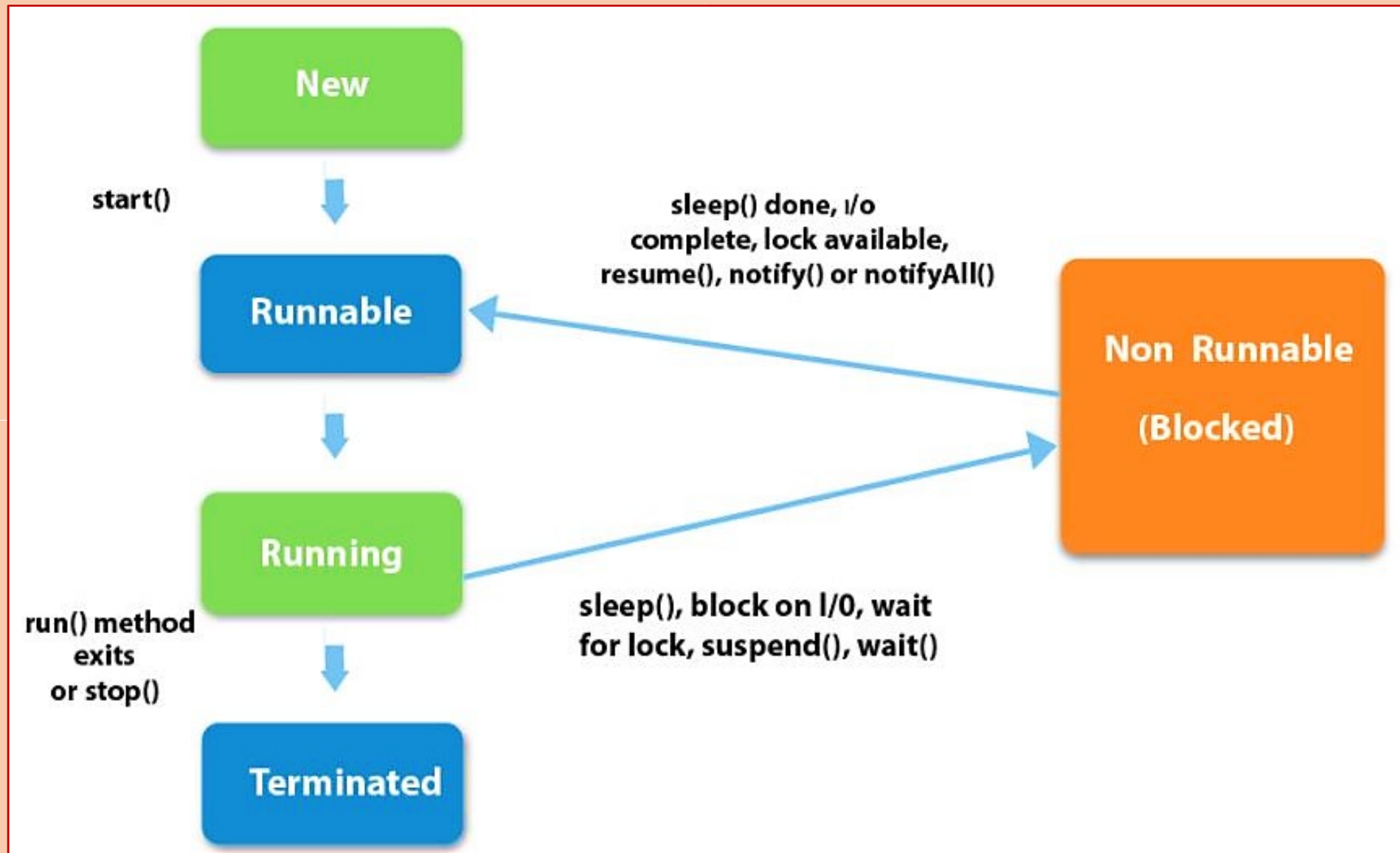
A thread can be in one of the **five states**.

According to sun, there is only 4 states in thread life cycle in java: new, runnable, non-runnable and terminated.

There is no running state. But for better understanding of threads, we can explain it in the 5 states.

- ❖ New
- ❖ Runnable
- ❖ Running
- ❖ Non-Runnable (Blocked)
- ❖ Terminated

# Life Cycle





- > **New** - The thread is in new state if you **create an instance of Thread class** but before the invocation of start() method.
- > **Runnable** - The thread is in runnable state after invocation of **start()** method, but the thread scheduler has not selected it to be the running thread.
- > **Running** - The thread is in running state if the thread scheduler has selected it.
- > **Non-Runnable (Blocked)** - This is the state when the thread is still **alive**, but is currently **not eligible to run**.
- > **Terminated** - A thread is in terminated or dead state when its **run()** method exits.

A Running Thread transit to one of the non-runnable states depending upon the circumstances.

**Sleeping:** The Thread sleeps for the specified amount of time.

**Blocked for I/O:** The Thread waits for a blocking operation to complete.

**Blocked for join completion:** The Thread waits for completion of another Thread.

**Waiting for notification:** The Thread waits for notification another Thread.

**Blocked for lock acquisition:** The Thread waits to acquire the lock of an object.

JVM executes the Thread, based on their priority and scheduling.

# CREATING THREAD

➤ There are two ways to create a thread:

By extending **Thread class**

By implementing **Runnable interface**.

❖ **Extending Thread class:**

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

## commonly used Constructors of Thread class:

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r, String name)

**Thread Methods** - Following is the list of important methods available in the Thread class.

**public void run()** : is used to perform action for a thread.

**public void start()** : starts the execution of the thread. JVM calls the run() method on the thread.

**public void sleep(long milliseconds)** : Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

**public void join()** : waits for a thread to die.

**public int getPriority()** : returns the priority of the thread.

**public int setPriority(int priority)** : changes the priority of the thread.

**public String getName():** returns the name of the thread.

**public Thread currentThread():** returns the reference of currently executing thread.

**public int getId():** returns the id of the thread.

**public Thread.State getState():** returns the state of the thread.

**public boolean isAlive():** tests if the thread is alive.

**public void suspend():** is used to suspend the thread(deprecated).

**public void resume():** is used to resume the suspended thread

**public void stop():** is used to stop the thread(deprecated).

**public boolean isDaemon():** tests if the thread is a daemon thread

## Thread.start() & Thread.run()

In Java's multi-threading concept, **start()** and **run()** are the two most important methods.

When a program calls the **start()** method, a new thread is created and then the **run()** method is executed.

But if we directly call the **run()** method then no new thread will be created and **run()** method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.

Let us understand it with an example:

```
class MyThread extends Thread {  
    public void run()  
    {  
        System.out.println("Current thread name: "  
            + Thread.currentThread().getName());  
        System.out.println("run() method called");  
    }  
}  
  
class Xyz {  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

## Output

Output:

```
Current thread name: Thread-0  
run() method called
```



## start ()

When we call the `start()` method of our thread class instance, a new thread is created with default name `Thread-0` and then `run()` method is called and everything inside it is executed on the newly created thread.

## run ()

When we called the `run()` method of our `MyThread` class, no new thread is created and the `run()` method is executed on the current thread i.e. *main* thread. Hence, no multi-threading took place. The `run()` method is called as a normal function call.

Let us try to call run() method directly instead of start() method:

```
class MyThread extends Thread {  
    public void run()  
    {  
        System.out.println("Current thread name: "  
            + Thread.currentThread().getName());  
        System.out.println("run() method called");  
    }  
  
class Xyz {  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.run();  
    }  
}
```

## Output

```
Current thread name: main  
run() method called
```

# fference

START()	RUN()
Creates a new thread and the run() method is executed on the newly created thread.	No new thread is created and the run() method is executed on the calling thread itself.
Can't be invoked more than one time otherwise throws <i>java.lang.IllegalStateException</i>	Multiple invocation is possible
Defined in <i>java.lang.Thread</i> class.	Defined in <i>java.lang.Runnable</i> interface and must be overridden in the implementing class.

## Implementing Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.

Runnable interface have only one method named **run()**.

**public void run():** is used to perform action for a thread.

## Steps to create a new Thread using Runnable :

Create a Runnable implementer and implement run() method.

Instantiate Thread class and pass the implementer to the Thread. Thread has a constructor which accepts Runnable instance.

Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start(), creates a new Thread which executes the code written in run().

# Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        Multi3 m1=new Multi3();  
        Thread t1 =new Thread(m1);  
        t1.start();  
    }  
}
```

Output:thread is running...

# MAIN THREAD

Every java program has a main method. The main method is the entry point to execute the program.

So, when the JVM starts the execution of a program, it creates a thread to run it and that thread is known as the **main thread**.

Each program must contain at least one thread whether we are creating any thread or not.

The **JVM provides** a default thread in each program.

**A program can't run without a thread**, so it requires at least one thread, and that thread is known as the main thread.

If you ever tried to run a Java program with compilation errors you would have seen the mentioning of main thread. Here is a simple Java program that tries to call the non-existent getValue() method

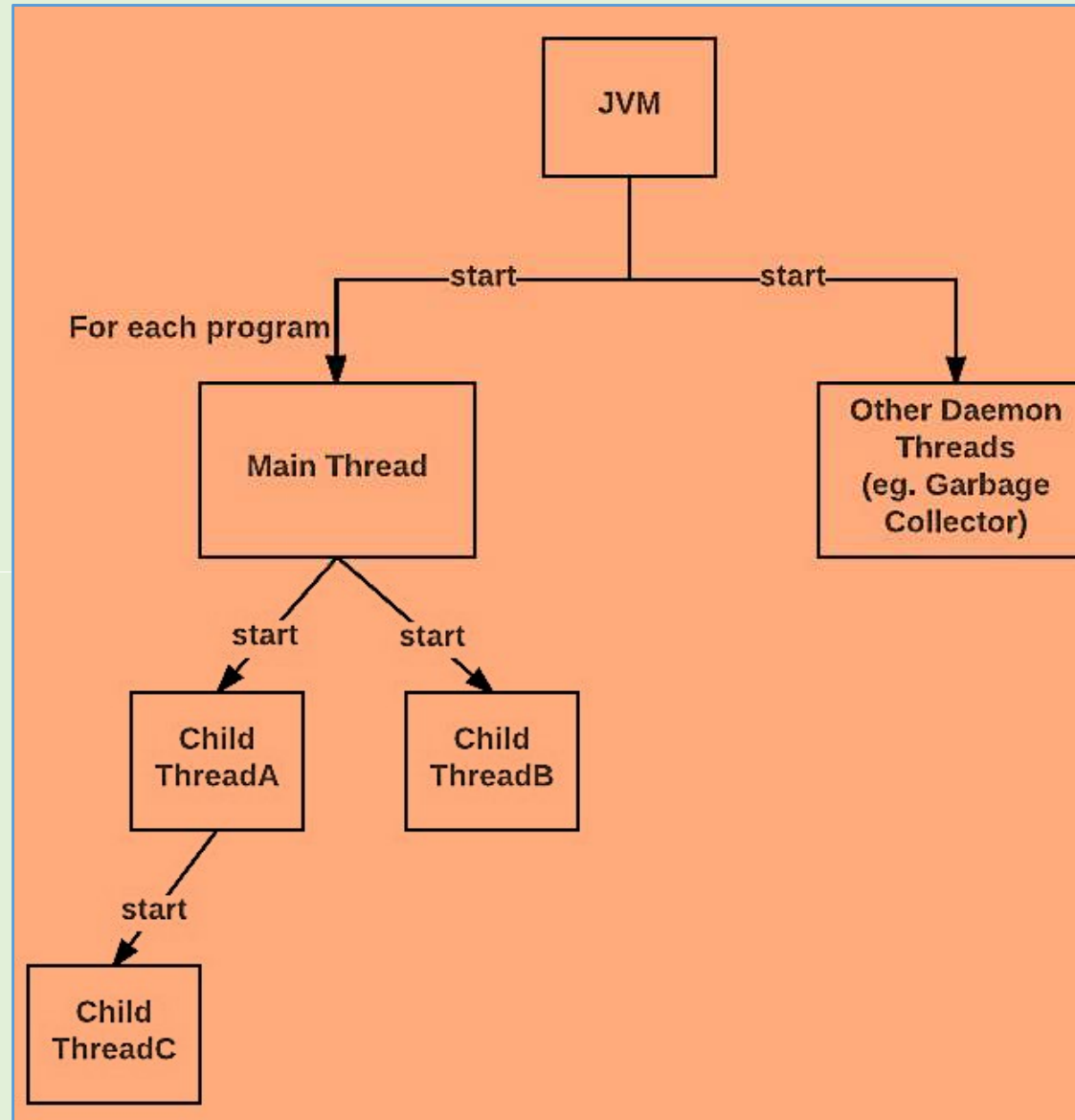
```
public class TestThread {  
    public static void main(String[] args) {  
        TestThread t = new TestThread();  
        t.getValue();  
    }  
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation  
The method getValue() is undefined for the type TestThread
```

As you can see in the error when the program is executed, main thread starts running and that has encountered a compilation problem.

## Properties

It is the thread from which other “child” threads will be spawned.  
Often, it must be the last thread to finish execution because it performs various shutdown actions





## How to control Main thread

The main thread is **created automatically** when our program started.

To control it we must obtain a **reference** to it.

This can be done by calling the method **currentThread( )** which is present in Thread class.

This method returns a reference to the thread on which it is called.

The **default priority of Main thread is 5** and for all remaining user threads priority will be inherited from parent to child.

inThread

```
public static void main(String args [ ] )
{
    Thread t = Thread.currentThread ( );
    System.out.println ("Current Thread : " + t);
    System.out.println ("Name : " + t.getName ( ) );
    System.out.println (" ");
    t.setName ("New Thread");
    System.out.println ("After changing name");
    System.out.println ("Current Thread : " + t);
    System.out.println ("Name : " + t.getName ( ) );
    System.out.println (" ");
    System.out.println ("This thread prints first 10 numbers");
    try
    {
        for (int i=1; i<=10;i++)
        {
            System.out.print(i);
            System.out.print(" ");
            Thread.sleep(1000);
        }
    }
    catch (InterruptedException e)
    {
        System.out.println(e);
    }
}
```

## Output

```
Current Thread : Thread[main,5,main]
Name : main
```

After changing name

```
Current Thread : Thread[New Thread,5,main]
Name : New Thread
```

```
This thread prints first 10 numbers
1 2 3 4 5 6 7 8 9 10
```

The program first creates a Thread object called 't' and assigns the reference of current thread (main thread) to it. So now main thread can be accessed via Thread object 't'.

This is done with the help of `currentThread()` method of Thread class which return a reference to the current running thread.

The Thread object 't' is then printed as a result of which you see the output Current Thread : Thread [main,5,main].

The first value in the square brackets of this output indicates the name of the thread, the name of the group to which the thread belongs.

The program then prints the name of the thread with the help of `getName()` method.

The name of the thread is changed with the help of `setName()` method.

The thread and thread name is then again printed.

Then the thread performs the operation of printing first 10 numbers.

When you run the program you will see that the system wait for sometime after printing each number.

This is caused by the statement `Thread.sleep (1000)`.

# CREATING MULTIPLE THREADS

```
class ThreadA extends Thread
{
    public void run()
    {
        for (int i=1;i<=5;i++)
        {
            System.out.println("ThreadA i="+(-1*i));
        }
        System.out.println("Exiting ThreadA");
    }
}

class ThreadB extends Thread
{
    public void run()
    {
        for (int j=1;j<=5;j++)
        {
            System.out.println("ThreadB j="+2*j);
        }
        System.out.println("Exiting ThreadB");
    }
}
```

```
class ThreadC extends Thread
{
    public void run()
    {
        for (int k=1;k<=5;k++)
        {
            System.out.println("ThreadC k="+2*(k-1));
        }
        System.out.println("Exiting ThreadC");
    }
}

class MultiThreadDemo
{
    public static void main (String args [])
    {
        ThreadA t1 = new ThreadA();
        ThreadB t2 = new ThreadB();
        ThreadC t3 = new ThreadC();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

## Output

ThreadA i = -1  
ThreadB j = 2  
ThreadC k = 1  
ThreadA i = -2  
ThreadB j = 4  
ThreadC k = 3  
ThreadA i = -3  
ThreadB j = 6  
ThreadC k = 5

ThreadA i = -4  
ThreadB j = 8  
ThreadC k = 7  
ThreadA i = -5  
ThreadB j = 10  
ThreadC k = 9  
Exiting ThreadA  
Exiting ThreadB  
Exiting ThreadC

# THREAD SYNCHRONIZATION

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.

For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

Following is the general form of the synchronized statement :

**Syntax**

```
synchronized(object identifier) {  
    // Access shared variables and other shared resources  
}
```

## Understanding the problem without Synchronization

In this example, we are not using synchronization and creating multiple threads that are accessing display method and producing the random output.



```

s First {
public void display(String msg)

System.out.print ("["+msg);
try {
    Thread.sleep(1000);
}
catch(InterruptedException e) {
    e.printStackTrace();
}
System.out.println ("]");

```

```

s Second extends Thread {
    String msg;
    First fobj;
    Second (First fp,String str) {
        fobj = fp;
        msg = str;
        start();
    }

```

```

public void run() {
    fobj.display(msg);
}

```

```

public class Syncro
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

In the above program, object fnew of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void display). Hence the result is nonsynchronized and such situation is called a race condition

#### OUTPUT:

```

[welcome [ new [ programmer]
]
]

```

## Synchronized Keyword

To synchronize above program, we must synchronize access to the shared display() method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display method.

With a synchronized method, the lock is obtained for the duration of the entire method.

So if you want to lock the whole object, use a synchronized method

**synchronized void display (String msg)**

Example : implementation of synchronized method

```

First

synchronized public void display(String msg) {
    System.out.print ("[" +msg);
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e) {
        printStackTrace();
    }
    System.out.println ("]");
}

```

```

Second extends Thread {
    String msg;
    First fobj;
    Second (First fp,String str) {
        fobj = fp;
        msg = str;
        start();
    }

    public void run() {
        fobj.display(msg);
    }
}

```

```

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

### OUTPUT:

```
[welcome]
```

```
[programmer]
```

```
[new]
```

## Using Synchronized block

If we want to synchronize access to an object of a class or only part of a method to be synchronized then we can use synchronized block for it.

It is capable to make any part of the object and method synchronized.

With synchronized blocks we can specify exactly when the lock is needed. If you want to keep other parts of the object accessible to other threads, use synchronized block.

### Example

In this example, we are using synchronized block that will make the display method available for single thread at a time.

```

First {
    public void display(String msg) {
        System.out.print ("["+msg);
    }
    Thread.sleep(1000);
    catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println ("]");
}

Second extends Thread {
    String msg;
    Object fobj;
    Second (First fp, String str) {
        fobj = fp;
        msg = str;
        start();
    }

    public void run() {
        synchronized(fobj)    //Synchronized block
        {
            fobj.display(msg);
        }
    }
}

```

```

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1 = new Second (fnew, "new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

### OUTPUT:

[welcome]

[new]

[programmer]

Which is more preferred - Synchronized method or Synchronized block?

In Java, synchronized keyword causes a performance cost.

A synchronized method in Java is very slow and can degrade performance.

So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

# Thread suspend() method

The **suspend()** method of thread class puts the thread from running to waiting state.

This method is used if you want to stop the thread execution and start it again when a certain event occurs.

This method allows a thread to temporarily cease execution.

The suspended thread can be resumed using the **resume** method.

**Syntax**

```
public final void suspend()
```



# ample

```
public class JavaSuspendExp extends Thread
```

```
{  
    public void run()  
    {  
        for(int i=1; i<5; i++)  
        {  
            try  
            {  
                // thread to sleep for 500 milliseconds  
                sleep(500);  
                System.out.println(Thread.currentThread().getName());  
            } catch (InterruptedException e) { System.out.println(e); }  
            System.out.println(i);  
        }  
    }  
}
```

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

```
{  
    // creating three threads  
    JavaSuspendExp t1=new JavaSuspendExp ();  
    JavaSuspendExp t2=new JavaSuspendExp ();  
    JavaSuspendExp t3=new JavaSuspendExp ();  
    // call run() method  
    t1.start();  
    t2.start();  
    // suspend t2 thread  
    t2.suspend();  
    // call run() method  
    t3.start();  
}
```



# Output

Thread-0

1

Thread-2

1

Thread-0

2

Thread-2

2

Thread-0

3

Thread-2

3

Thread-0

4

Thread-2

4

# Thread resume() method

The resume() method of thread class is only used with suspend() method.

This method is used to resume a thread which was suspended using suspend() method.

This method allows the suspended thread to start again.

**Syntax**

```
public final void resume()
```

# Example

```
public class JavaResumeExp extends Thread

public void run()

{
    for(int i=1; i<5; i++)
    {
        try
        {
            // thread to sleep for 500 milliseconds
            sleep(500);
            System.out.println(Thread.currentThread().getName());
        } catch (InterruptedException e) { System.out.println(e); }
        System.out.println(i);
    }
}
```

```
public static void main(String args[])
{
    // creating three threads
    JavaResumeExp t1=new JavaResumeExp
    JavaResumeExp t2=new JavaResumeExp
    JavaResumeExp t3=new JavaResumeExp
    // call run() method
    t1.start();
    t2.start();
    t2.suspend(); // suspend t2 thread
    // call run() method
    t3.start();
    t2.resume(); // resume t2 thread
}
}
```

tput

Thread-0

1

Thread-2

1

Thread-1

1

Thread-0

2

Thread-2

2

Thread-1

2

Thread-0

3

Thread-2

3

Thread-1

3

Thread-0

4

Thread-2

4

Thread-1

4

# Thread stop() method

The stop() method of thread class terminates the thread execution.

Once a thread is stopped, it cannot be restarted by start() method.

## Syntax

```
public final void stop()
```

```
public final void stop(Throwables obj)
```

# ample

```
public class JavaStopExp extends Thread

public void run()

for(int i=1; i<5; i++)
{
    try
    {
        // thread to sleep for 500 milliseconds
        sleep(500);
        System.out.println(Thread.currentThread().getName());
    } catch (InterruptedException e) { System.out.println(e); }
    System.out.println(i);
}
```

```
public static void main(String args[])
{
    // creating three threads
    JavaStopExp t1=new JavaStopExp ();
    JavaStopExp t2=new JavaStopExp ();
    JavaStopExp t3=new JavaStopExp ();
    // call run() method
    t1.start();
    t2.start();
    // stop t3 thread
    t3.stop();
    System.out.println("Thread t3 is stopped");
}
}
```

# **MODULE 4**

## **CHAPTER 3**

### **EVENT HANDLING**

# EVENT

Change in the state of an object is known as event i.e. event describes the **change in state** of source.

Events are generated as result of user interaction with the graphical user interface components.

For example, clicking on a button, moving the mouse, entering character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

## Types of Event

The events can be broadly classified into **two** categories:



## Foreground Events

Those events which require the **direct interaction** of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

## Background Events

Those events that require the **interaction of end user** are known as background events. Operating system interrupts, hardware failure, software failure, timer expires, an operation completion are the example of background events.

# EVENT HANDLING

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.

This mechanism has the code which is known as **event handler** that is executed when an event occurs.

Java Uses the **Delegation Event Model** to handle the events.

This model defines the standard mechanism to generate and handle the events.

Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

**Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides as with classes for source object.

**Listener** - It is also known as **event handler**. Listener is responsible for **generating response to an event**. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event.

The user interface element is able to delegate the processing of an event to the separate piece of code.

In this model, **Listener needs to be registered with the source object** so that the listener can receive the event notification.

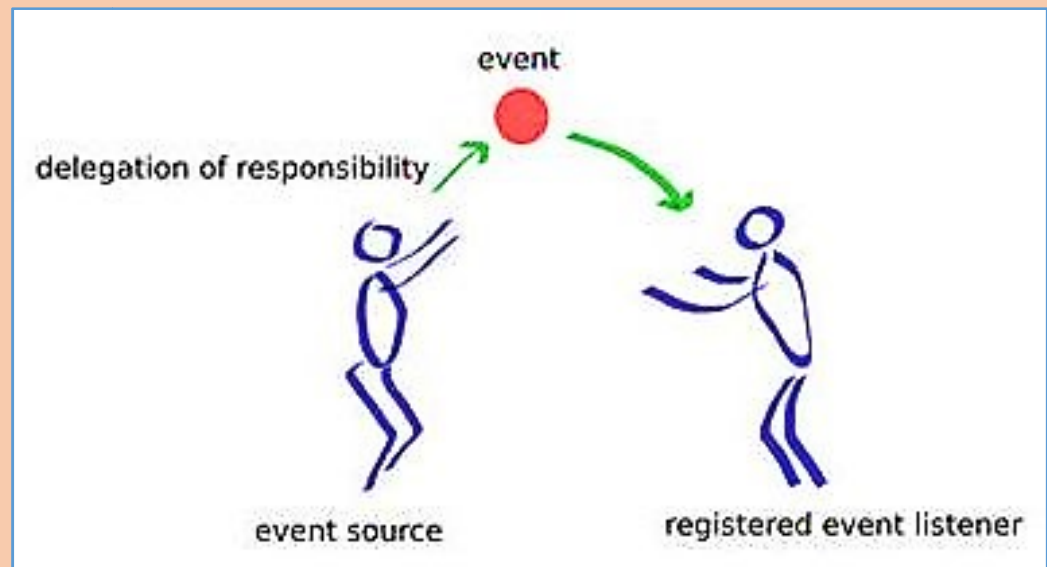
This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

## How Events are handled

A source generates an Event and send it to one or more listeners registered with the source.

Once event is received by the listener, they process the event and then return.

Events are supported by a number of Java packages, like `java.util`, `java.awt` and `java.awt.event`



## Event classes and interface

Event Classes	Description	Listener Interface
<b>ActionEvent</b>	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
<b>MouseEvent</b>	generated when mouse is dragged, moved, clicked, pressed or released and also when it enters or exit a component	MouseListener
<b>KeyEvent</b>	generated when input is received from keyboard	KeyListener
<b>ItemEvent</b>	generated when check-box or list item is clicked	ItemListener
<b>TextEvent</b>	generated when value of textarea or textfield is changed	TextListener
<b>MouseEvent</b>	generated when mouse wheel is moved	MouseWheelListener

<b>WindowEvent</b>	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
<b>ComponentEvent</b>	generated when component is hidden, moved, resized or set visible	ComponentEventListener
<b>ContainerEvent</b>	generated when component is added or removed from container	ContainerListener
<b>AdjustmentEvent</b>	generated when scroll bar is manipulated	AdjustmentListener
<b>FocusEvent</b>	generated when component gains or loses keyboard focus	FocusListener

## Steps involved in event handling

The User clicks the button and the event is generated.

Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

Event object is forwarded to the method of registered listener class.

The method is now get executed and returns.



## ► Points to remember about listener

In order to design a listener class we have to develop some listener interfaces.

These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.

If we do not implement the predefined interfaces then your class can not act as a listener class for a source object.

# SOURCES OF EVENT

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

# EVENT LISTENER INTERFACES

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyApplet extends JApplet implements KeyListener {

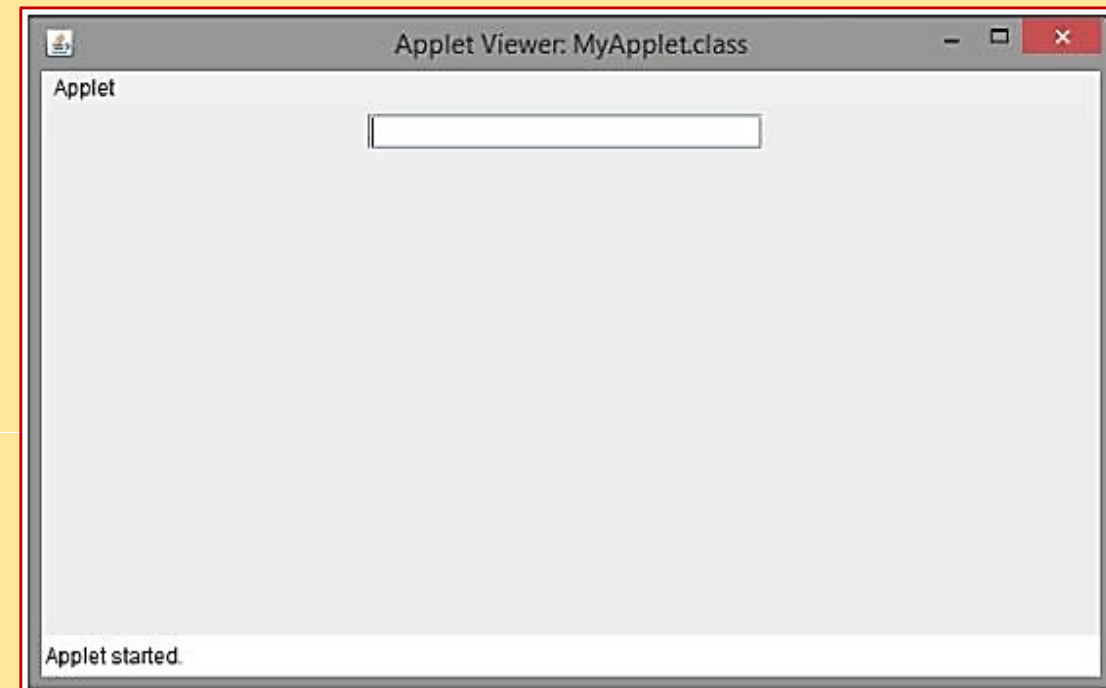
    JTextField jtf;
    JLabel label;

    public void init()
    {
        setSize(600,300);
        setLayout(new FlowLayout());
        jtf = new JTextField(20);
        add(jtf);
        jtf.addKeyListener(this);
        label = new JLabel();
        add(label);
    }

    public void keyPressed(KeyEvent ke){}
    public void keyReleased(KeyEvent ke){}
    public void keyTyped(KeyEvent ke)
    {
        label.setText(String.valueOf(ke.getKeyChar()));
    }
}
```

# Key Event Handling

## Initial output of the program



After the user enters a character into the text field, the same character is displayed in the label beside the text field as shown in the below image:

