What is a Thread in Java?

A thread in Java is the direction or path that is taken while a program is being executed. Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution. At this point, when the main thread is provided, the main() method is invoked by the main thread.

A thread is critical in the program because it enables multiple operations to take place within a single method. Each thread in the program often has its own program counter, stack, and local variable.

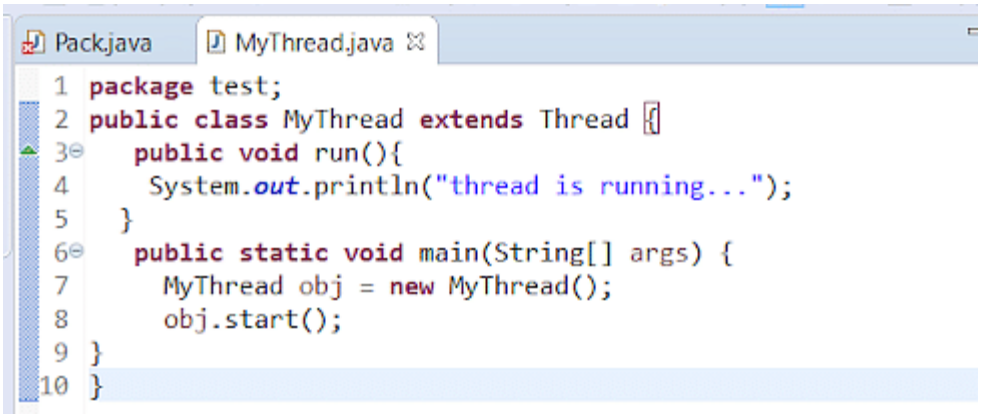
Creating a Thread in Java

A thread in Java can be created in the following two ways:

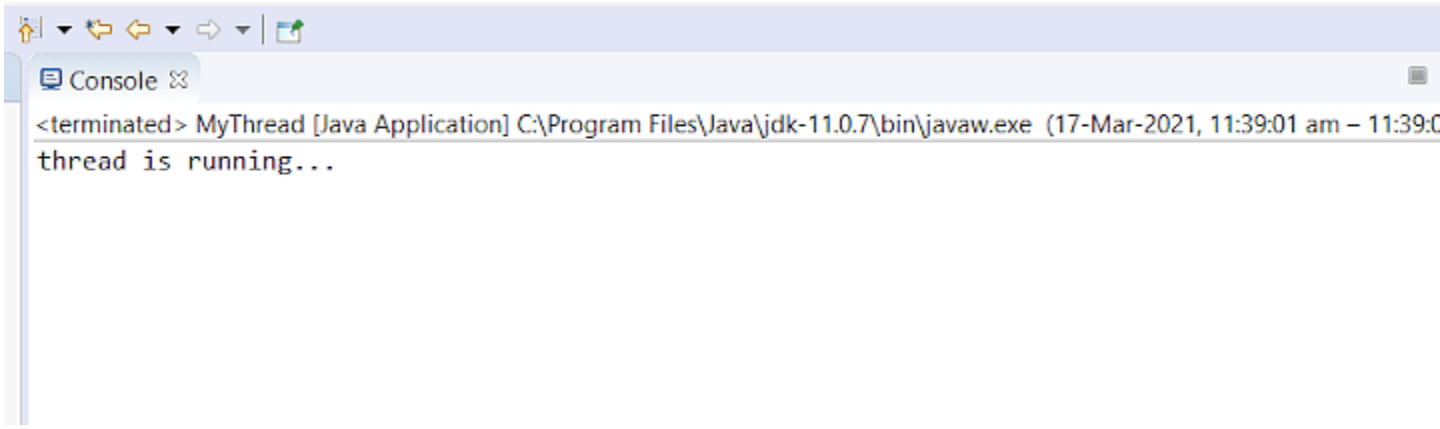
- Extending java.lang.Thread class

In this case, a thread is created by a new class that extends the Thread class, creating an instance of that class. The run() method includes the functionality that is supposed to be implemented by the Thread.

Below is an example to create a thread by extending java.lang.Thread class.



Output



Here, start() is used to create a new thread and to make it runnable. The new thread begins inside the void run() method.

- Implementing Runnable interface

This is the easy method to create a thread among the two. In this case, a class is created to implement the runnable interface and then the run() method.

The code for executing the Thread should always be written inside the run() method.

Here's a code to make you understand it.

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

Output

```
<terminated> MyThread [Java Application] C:\Program Files\Java\jdk-11.0.7\bin\javaw.exe  (17-Mar-2021, 11:49:41 am – 11:49:44
thread is running..
```

The start() method is used to call the void run() method. When start() is called, a new stack is given to the thread, and run() is invoked to introduce a new thread in the program.

Multithreading in Java

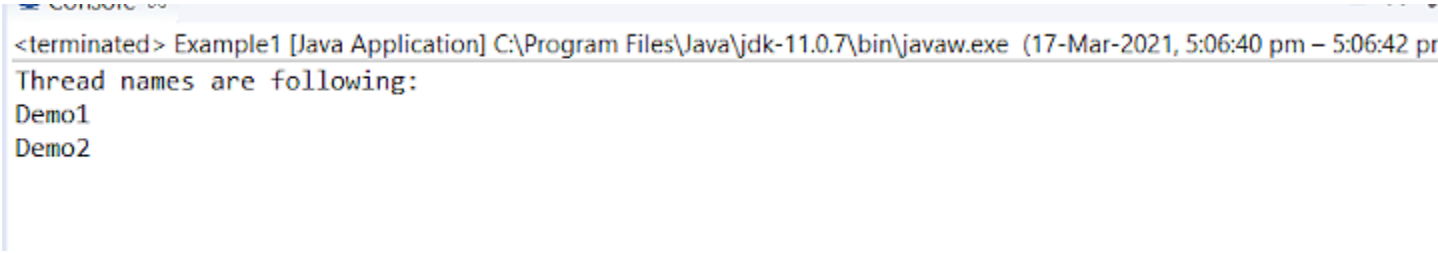
In Java, multithreading is the method of running two or more threads at the same time to maximize CPU utilization. As a result, it is often referred to as Concurrency in Java. Each thread runs in parallel with the others. Since several threads do not assign different memory areas, they conserve memory. Furthermore, switching between threads takes less time.

In Java, multithreading enhances program structure by making it simpler and easier to navigate. These generalized threads can be used in high-server media applications to easily change or enhance the configuration of these complex structures.

Here is an example of Multithreading in Java.

```
1  package test;
2  public class Example1 implements Runnable{
3
4      public static void main(String[] args) {
5          Thread Example1 = new Thread("Demo1");
6          Thread Example2 = new Thread("Demo2");
7          Example1.start();
8          Example2.start();
9          System.out.println("Thread names are following:");
10         System.out.println(Example1.getName());
11         System.out.println(Example2.getName());
12     }
13     @Override
14     public void run() {
15     }
16
17 }
18
19
```

Output



Java Thread isAlive() method

The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead. This method returns true if the thread is still running and not finished.

Syntax

```
1. public final boolean isAlive()
```

Return

This method will return true if the thread is alive otherwise returns false.

Example

```
1. public class JavaIsAliveExp extends Thread
2. {
3.    public void run()
4.    {
5.       try
6.       {
7.          Thread.sleep(300);
8.          System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
9.       }
10.      catch (InterruptedException ie) {
11.      }
```

```
12.    }
13.    public static void main(String[] args)
14.    {
15.        JavaIsAliveExp t1 = new JavaIsAliveExp();
16.        System.out.println("before starting thread isAlive: "+t1.isAlive());
17.        t1.start();
18.        System.out.println("after starting thread isAlive: "+t1.isAlive());
19.    }
20. }
```

21. **Output:**

```
22.  before starting thread isAlive: false
23.  after starting thread isAlive: true
24.  is run() method isAlive true
```

Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose *th* be the object the class Thread whose thread is doing its execution currently, then the *th.join();* statement ensures that *th* is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

Description of The Overloaded join() Method

join(): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

1. **public final void** join() **throws** InterruptedException

join(long mls): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.
public final synchronized void join(**long** mls) **throws** InterruptedException, where mls is in mill iseconds

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

1. **public final synchronized void** join(**long** mls, **int** nanos) **throws** InterruptedException, where mls is in milliseconds.

Example of join() Method in Java

The following program shows the usage of the join() method.

FileName: ThreadJoinExample.java

```
1. // A Java program for understanding
2. // the joining of threads
3.
4. // import statement
5. import java.io.*;
6.
7. // The ThreadJoin class is the child class of the class Thread
8. class ThreadJoin extends Thread
9. {
10. // overriding the run method
11. public void run()
12. {
```

```
13. for (int j = 0; j < 2; j++)
14. {
15. try
16. {
17. // sleeping the thread for 300 milli seconds
18. Thread.sleep(300);
19. System.out.println("The current thread name is: " + Thread.currentThread().getName());
20. }
21. // catch block for catching the raised exception
22. catch(Exception e)
23. {
24. System.out.println("The exception has been caught: " + e);
25. }
26. System.out.println( j );
27. }
28. }
29. }
30.
31. public class ThreadJoinExample
32. {
33. // main method
34. public static void main (String argvs[])
35. {
36.
37. // creating 3 threads
38. ThreadJoin th1 = new ThreadJoin();
39. ThreadJoin th2 = new ThreadJoin();
40. ThreadJoin th3 = new ThreadJoin();
41.
42. // thread th1 starts
43. th1.start();
44.
45. // starting the second thread after when
46. // the first thread th1 has ended or died.
47. try
```

```
48. {
49. System.out.println("The current thread name is: "+ Thread.currentThread().getName());
50.
51. // invoking the join() method
52. th1.join();
53. }
54.
55. // catch block for catching the raised exception
56. catch(Exception e)
57. {
58. System.out.println("The exception has been caught " + e);
59. }
60.
61. // thread th2 starts
62. th2.start();
63.
64. // starting the th3 thread after when the thread th2 has ended or died.
65. try
66. {
67. System.out.println("The current thread name is: " + Thread.currentThread().getName());
68. th2.join();
69. }
70.
71. // catch block for catching the raised exception
72. catch(Exception e)
73. {
74. System.out.println("The exception has been caught " + e);
75. }
76.
77. // thread th3 starts
78. th3.start();
79. }
80. }
```

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
```

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

1. **public static void** sleep(**long** mls) **throws** InterruptedException
2. **public static void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread

The following example shows how one can use the sleep() method on the custom thread.

FileName: TestSleepMethod1.java

```
1. class TestSleepMethod1 extends Thread{
2.  public void run(){
3.   for(int i=1;i<5;i++){
4.   // the thread will sleep for the 500 milli seconds
5.     try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
6.     System.out.println(i);
7.   }
8.  }
9.  public static void main(String args[]){
```

```
10.  TestSleepMethod1 t1=new TestSleepMethod1();
11.  TestSleepMethod1 t2=new TestSleepMethod1();
12.
13.  t1.start();
14.  t2.start();
15.  }
16. }
```

Output:

```
1
1
2
2
3
3
4
4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

FileName: TestSleepMethod2.java

```
1.  // important import statements
2.  import java.lang.Thread;
3.  import java.io.*;
4.
5.
6.  public class TestSleepMethod2
7.  {
8.     // main method
9.  public static void main(String argvs[])
10. {
11.
12. try {
```

```
13. for (int j = 0; j < 5; j++)
14. {
15.
16. // The main thread sleeps for the 1000 milliseconds, which is 1 sec
17. // whenever the loop runs
18. Thread.sleep(1000);
19.
20. // displaying the value of the variable
21. System.out.println(j);
22. }
23. }
24. catch (Exception expn)
25. {
26. // catching the exception
27. System.out.println(expn);
28. }
29. }
30. }
```

Output:

```
0
1
2
3
4
```

Synchronization in Java

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.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
1. class Table{
2. void printTable(int n){//method not synchronized
3.   for(int i=1;i<=5;i++){
4.     System.out.println(n*i);
5.     try{
6.      Thread.sleep(400);
7.     }catch(Exception e){System.out.println(e);}
8.   }
9.
10.  }
11. }
12.
13. class MyThread1 extends Thread{
14. Table t;
15. MyThread1(Table t){
16. this.t=t;
17. }
18. public void run(){
19. t.printTable(5);
20. }
21.
22. }
```

```
23. class MyThread2 extends Thread{
24. Table t;
25. MyThread2(Table t){
26. this.t=t;
27. }
28. public void run(){
29. t.printTable(100);
30. }
31. }
32.
33. class TestSynchronization1{
34. public static void main(String args[]){
35. Table obj = new Table();//only one object
36. MyThread1 t1=new MyThread1(obj);
37. MyThread2 t2=new MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

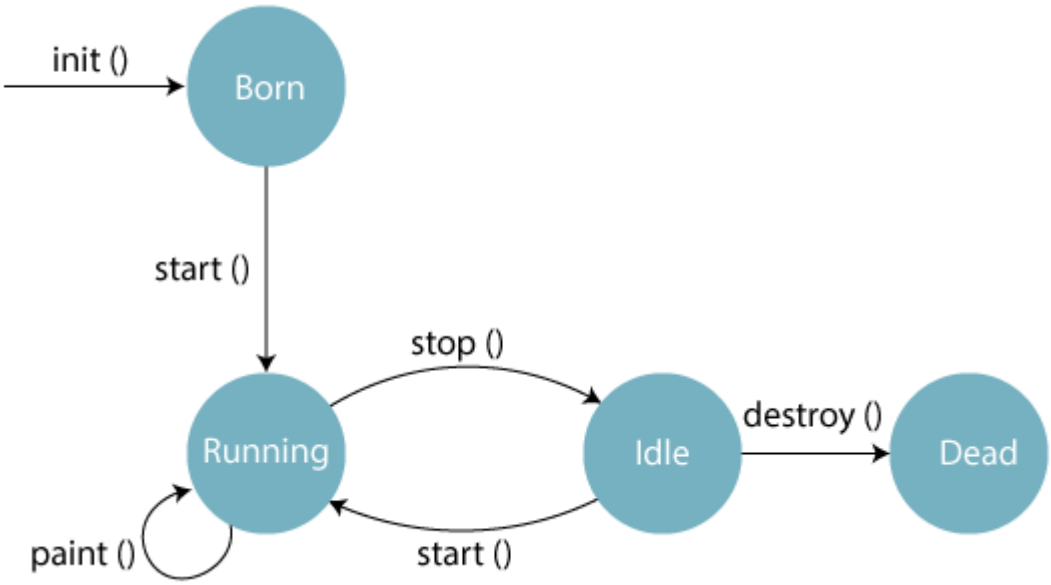
Applet Life Cycle in Java

In Java, an applet is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint() and destroy().These methods are invoked by the browser to execute.

Along with the browser, the applet also works on the client side, thus having less processing time.

Methods of Applet Life Cycle



There are five methods of an applet life cycle, and they are:

- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.
- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is

loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.

- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.
- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

Sequence of method execution when an applet is executed:

1. init()
2. start()
3. paint()

Sequence of method execution when an applet is executed:

1. stop()
2. destroy()

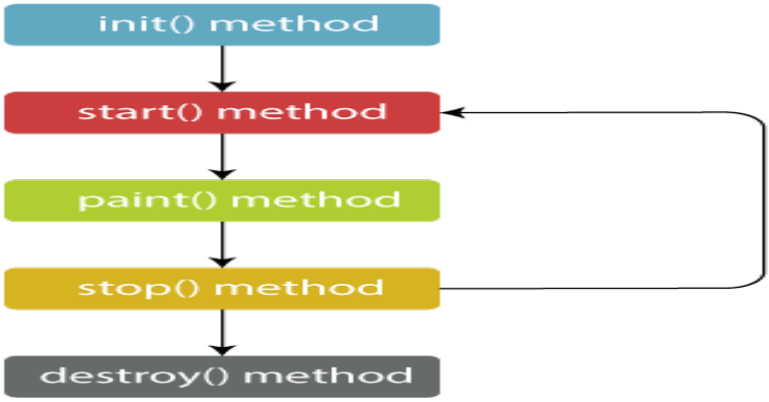
Applet Life Cycle Working

- The Java plug-in software is responsible for managing the life cycle of an applet.
- An applet is a Java application executed in any web browser and works on the client-side. It doesn't have the main() method because it runs in the browser. It is thus created to be placed on an HTML page.
- The init(), start(), stop() and destroy() methods belongs to the **applet.Applet** class.
- The paint() method belongs to the **awt.Component** class.
- In Java, if we want to make a class an Applet class, we need to extend the **Applet**

- Whenever we create an applet, we are creating the instance of the existing Applet class. And thus, we can use all the methods of that class.

Flow of Applet Life Cycle:

These methods are invoked by the browser automatically. There is no need to call them explicitly.



Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener

AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}

- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class AEvent extends Frame implements ActionListener{
4. TextField tf;
5. AEvent(){
6. 
7. //create components
8. tf=new TextField();
9. tf.setBounds(60,50,170,20);
10. Button b=new Button("click me");
11. b.setBounds(100,120,80,30);
12. 
13. //register listener
14. b.addActionListener(this);//passing current instance
15. 
16. //add components and set size, layout and visibility
17. add(b);add(tf);
18. setSize(300,300);
19. setLayout(null);
```

```
20. setVisible(true);
21. }
22. public void actionPerformed(ActionEvent e){
23. tf.setText("Welcome");
24. }
25. public static void main(String args[]){
26. new AEvent();
27. }
28. }
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.