Unit - 3

Multithreading: -

1. **Differences between multi-threading and multitasking**
2. **Thread life cycle**
3. **Creating threads**
4. **Thread priorities**
5. **Synchronizing threads**
6. **Inter thread communication**

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

Process:-

- A Program does nothing unless its instructions are executed by a CPU. A program in execution is called a process.

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

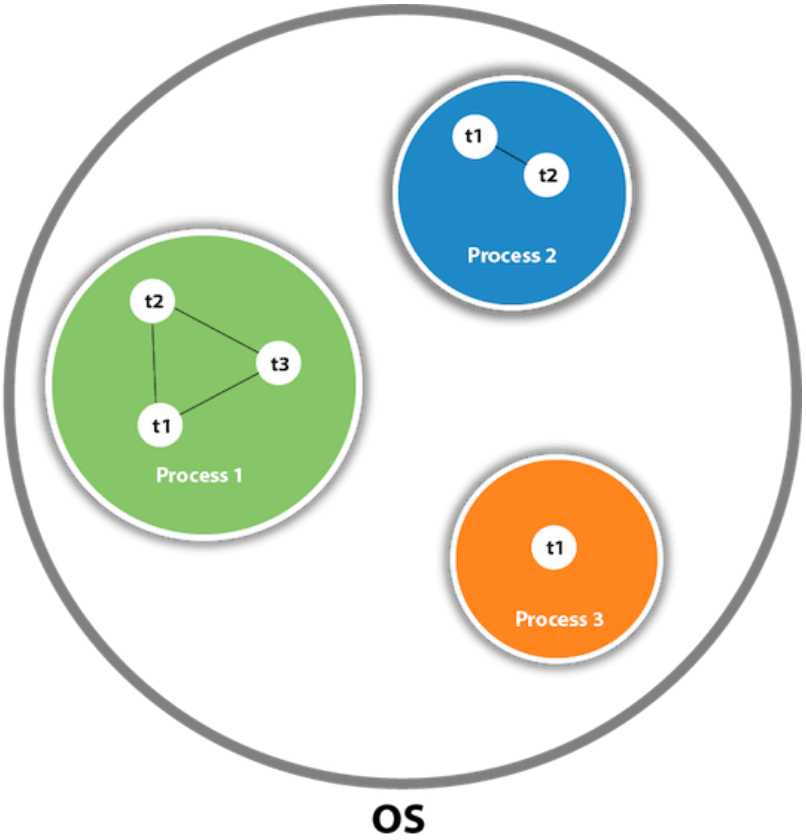
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

Thread:-

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

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared 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.

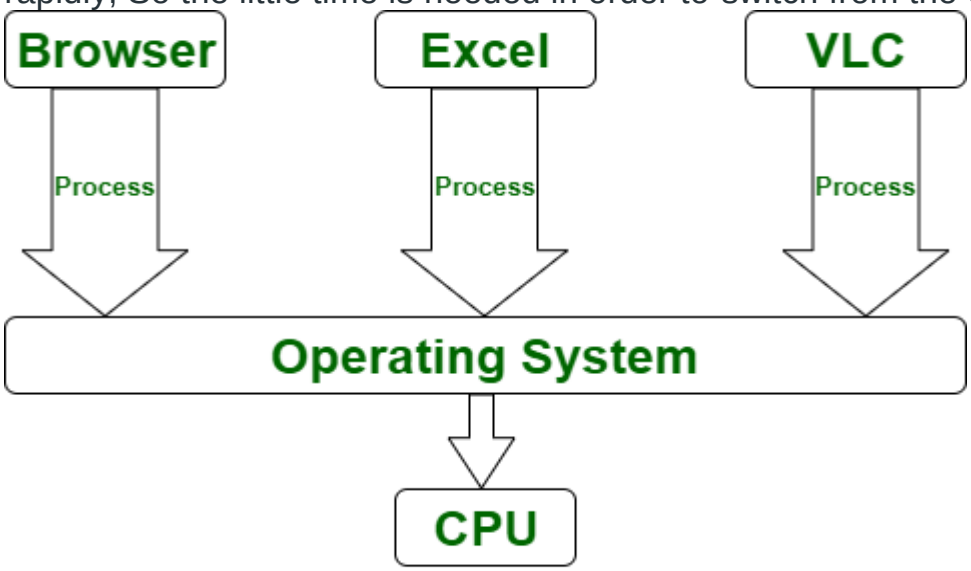
Advantages of Java Multithreading

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

2) You **can perform many operations together, so it saves time**.

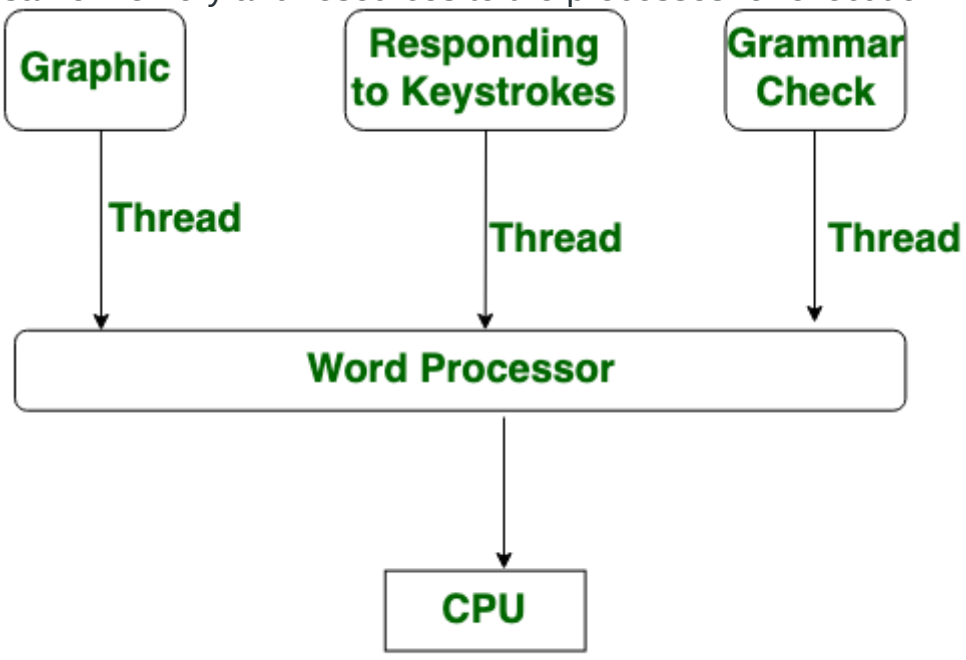
3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking: Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, In multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.



Multitasking

Multithreading: Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.



Multithreading

Example Program:-

- Write a program to find the thread used by JVM to execute the statements.

```
class ThreadClass2
{
    public static void main(String args[])
    {
        System.out.println(Thread.currentThread());
        System.out.println(Thread.currentThread().getName());
    }
}
```

O/P:-

```
Thread[main,5,main]
Main
```

1.difference between multitasking and multithreading:

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.(or) Multitasking is a process of executing multiple tasks simultaneously.	In multithreading, multiple threads are created from a process.(or) Multithreading is a process of executing multiple threads simultaneously.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
5.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don’t share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
11.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
12.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
13.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
15.	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

2.Thread life cycle

Starting from the birth of a thread, till its death, a thread exists in different states which are collectively called 'Thread Life Cycle'.

Thread States in Java

A thread is a path of execution in a program that enters any one of the following five states during its life cycle. The five states are as follows:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead

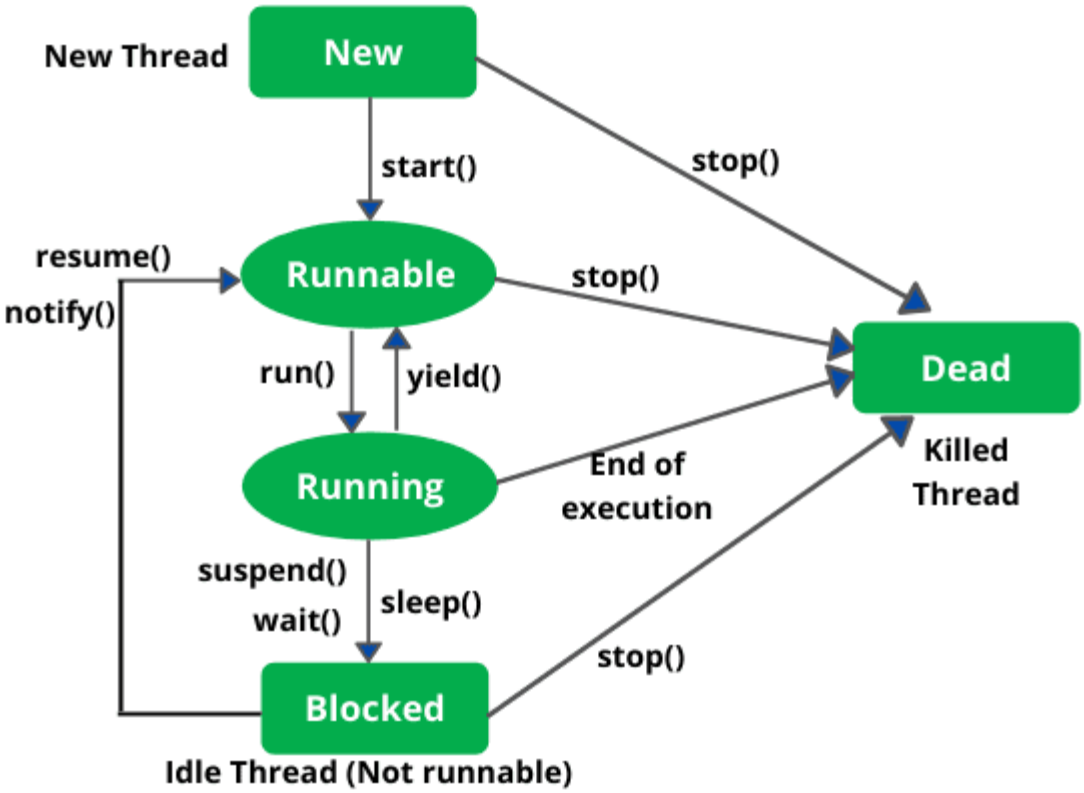


Fig. Thread Life Cycle

1. **New (Newborn State):** When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state but the start() method has not been called yet on the instance.
   In other words, Thread object exists but it cannot execute any statement because it is not an execution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

2. **Runnable state:** Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.
    - In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution.
    - If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.
3. **Running state:** Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.
    - In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.
    - A running thread may give up its control in any one of the following situations and can enter into the blocked state.

      a) When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

      b) When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

      c) When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.
4. **Blocked state:** A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.
5. **Dead state:** A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

Thread Class

Thread class contains several constructors for creating threads for tasks and methods for controlling threads. It is a predefined class declared in java.lang default package.

Each thread in Java is created and controlled by a unique object of the Thread class. An object of thread controls a thread running under JVM.
Thread class contains various methods that can be used to start, control, interrupt the execution of a thread, and for many other thread related activities in a program.

Thread class extends Object class and it implements Runnable interface. The declaration of thread class is as follows:

```
public class Thread

  extends Object

    implements Runnable
```

Look at the below figure that shows class diagram of Java Thread class.

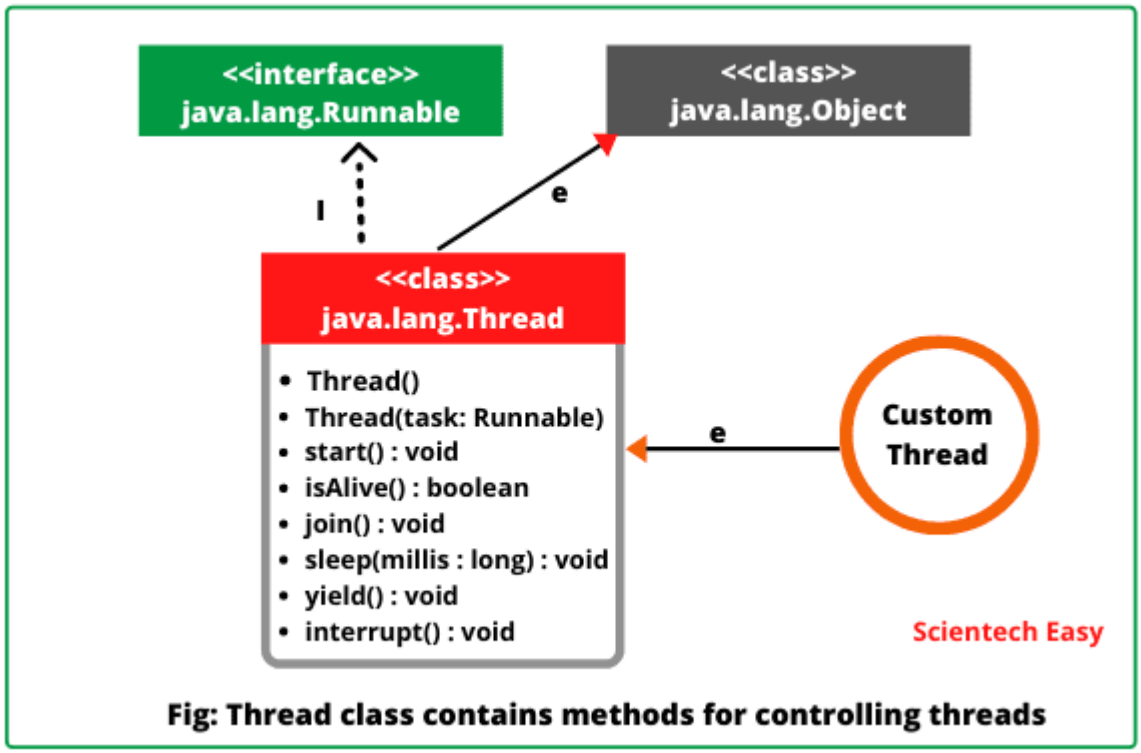


Fig: Thread class contains methods for controlling threads

Thread class Constructor: -

The various constructors of thread class are defined in java.lang package that can be used to create an object of thread are as follows:

1. **Thread():** This is a basic and default constructor without parameters. It simply creates an object of Thread class.
2. **Thread(String name):** It creates a new thread object with specified name to a thread.
3. **Thread(Runnable r):** It creates a thread object by passing a parameter r as a reference to an object of the class that implements Runnable interface.
4. **Thread(Runnable r, String name):** This constructor creates a thread object by passing two arguments r and name. Here, variable r is a reference of an object of class that implements Runnable interface.

Methods of Thread class:-

Thread class provides various static methods that are as follows:

1. **currentThread():** The currentThread() returns the reference of currently executing thread. Since this is a static method, so we can call it directly using the class name. The general syntax for currentThread() is as follows:

Syntax:

```
public static Thread currentThread()
```

2. **sleep():** The sleep() method puts currently executing thread to sleep for specified number of milliseconds. This method is used to pause the current thread for specified amount of time in milliseconds.
Since this method is static, so we can access it through Thread class name. The general syntax of this method is as follows:

Syntax:

```
public static void sleep(long milliseconds) throws InterruptedException
```

The general syntax for overloaded version of sleep method is as follows:

Syntax:

```
public static void sleep(long milliseconds, int nanoseconds ) throw InterruptedException
```

The overloaded version of sleep() method is used to pause specified period of time in milliseconds and nanoseconds. Both methods throw InterruptedException and must be used within Java try-catch block.

3. **yield():** The yield() method pauses the execution of current thread and allows another thread of equal or higher priority that are waiting to execute. Currently executing thread give up the control of the CPU. The general form of yield() method is as follows:

Syntax:

```
public static void yield()
```

4. **activeCount():** This method returns the number of active threads.

Syntax:

```
public static int activeCount()
```

Since this method is static, so it can be accessed through Thread class name. It does not accept anything.

The instance methods of Thread class are as follows:

1. **start():** The start() method is used to start the execution of a thread by calling its run() method. JVM calls the run() method on the thread. The general syntax for start() method is as follows:

Syntax:

```
public void start()
```

2. **run():** The run() method moves the thread into running state. It is executed only after the start() method has been executed. The general syntax is as follows:

Syntax:

```
public void run()
```

3. **getName():** This method returns the name of the thread. The return type of this method is String. The general form of this method is:

Syntax:

```
public final String getName()
```

4. **setName():** This method is used to set the name of a thread. It takes an argument of type String. It returns nothing.

Syntax:

```
public final void setName(String name)
```

5. **getPriority():** This method returns the priority of a thread. It returns priority in the form of an integer value ranging from 1 to 10. The maximum priority is 10, the minimum priority is 1, and normal priority is 5.

Syntax:

```
public final int getPriority() // Return type is an integer.
```

6. **setPriority():** This method is used to set the priority of a thread. It accepts an integer value as an argument. The general syntax is given below:

Syntax:

```
public final void setPriority(int newPriority)
```

7. **isAlive():** This method is used to check the thread is alive or not. It returns a boolean value (true or false) that indicates thread is running or not. The isAlive() method is final and native. The general syntax for isAlive() method is as follows:

Syntax:

```
public final native boolean isAlive()
```

8. **join():** The join() method is used to make a thread wait for another thread to terminate its process. The general syntax is

Syntax:

```
public final void join() throw InterruptedException
```

This method throws InterruptedException and must be used within a try-catch block.

9. **stop():** This method is used to stop the thread. The general form for this method is as follows:

Syntax:

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

This method neither accepts anything nor returns anything.

10. **suspend():** The suspend() method is used to suspend or pause a thread.

Syntax:

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

11. **resume():** This method is used to resume the suspended thread. It neither accepts anything nor returns anything.

Syntax:

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

12. **isDaemon():** This method is used to check the thread is daemon thread or not.

Syntax:

```
public final boolean isDaemon()
```

Runnable Interface Method:-

public void run():-

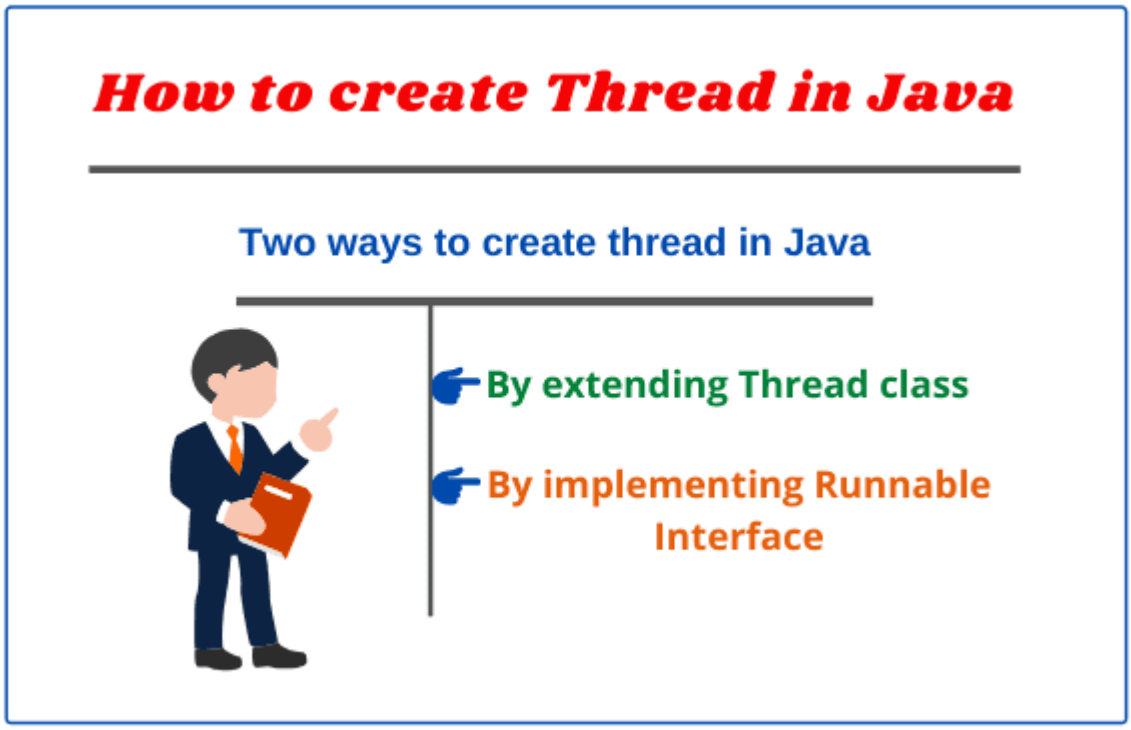
This method takes in no arguments. When the object of a class implementing Runnable class is used to create a thread, then the run method is invoked in the thread which executes separately.

3.Creating threads

Creating threads in Java | We know that every Java program has at least one thread called main thread. When a program starts, main thread starts running immediately. Apart from this main thread, we can also create our own threads in a program that is called child thread. Every child threads create from its main thread known as parent thread.

There are two ways to create a new thread in Java. They are as follows:

1. One is by extending java.lang.Thread class
2. Another is by implementing java.lang.Runnable interface



1.By Extending Thread class

Extending Thread class is the easiest way to create a new thread in Java. The following steps can be followed to create your own thread in Java.

1. Create a class that extends the Thread class. In order to extend a thread, we will use a keyword extends. The Thread class is found in java.lang package.

The syntax for creating a new class that extends Thread class is as follows:

```
Class Myclass extends Thread
```

2. Now in this newly created class, define a method run(). Here, run() method acts as entry point of the new thread. The run() method contains the actual task that thread will perform. Thus, we override run() method of Thread class.

```
public void run()

{

 // statements to be executed.

}
```

3. Create an object of newly created class so that run() method is available for execution. The syntax to create an object of Myclass is as follows:

```
Myclass obj = new Myclass();
```

4. Now create an object of Thread class and pass the object reference variable created to the constructor of Thread class.

```
Thread t = new Thread(obj);

or,

 Thread t = new Thread(obj, "threadname");
```

5. Run the thread. For this, we need to call to start() method of Thread class because we cannot call run() method directly. The syntax to call start() method is as follows:

```
t.start();
```

Now, the thread will start execution on the object of Myclass, and statements inside run() method will be executed while calling it. By following all the above steps, you can create a Thread in Java.

Example Program 1:

```
class MyThread extends Thread{
    @Override
    public void run()
    {
        for(int i=1;i<=2;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
public class TreadClass {
    public static void main(String args[])
    {
    MyThread t=new MyThread();
    t.start();
    for(int i=1;i<=2;i++)
        {
```

```
            System.out.println("Main Thread");
        }
    }
}
```

Out Put:-

Main Thread

Main Thread

Child Thread

Child Thread

Example Program 2:-

```
class MyThread extends Thread{
    @Override
    public void run()
    {
        for(int i=1;i<=2;i++)
        {
            System.out.println(Thread.currentThread());
        }
    }
}
public class TreadClass {
    public static void main(String args[])
    {
    MyThread t=new MyThread();
    MyThread t1=new MyThread();
     t.start();
     t1.start();
    for(int i=1;i<=2;i++)
        {
            System.out.println(Thread.currentThread());
        }
```

```
    }
}
```

Out Put:-

```
Thread[Thread-0,5,main]
Thread[Thread-1,5,main]
Thread[main,5,main]
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
Thread[main,5,main]
```

2. By Implementing Runnable Interface

1. Threads can also be created by implementing Runnable interface of java.land package. Creating a thread by implementing Runnable interface is very similar to creating a thread by extending Thread class.
2. All steps are similar for creating a thread by implementing Runnable interface except the first step. The first step is as follows:
3. 1. To create a new thread using this method, create a class that implements Runnable interface of java.lang package.
4. The syntax for creating a new class that implements Runnable interface is as follows:
5. class Myclass implements Runnable

Example Program:-

```
class MyThread implements Runnable
{
  public void run()
  {
   for(int i=1;i<=5;i++)
   {
      System.out.println("My Child Thread");
   }

  }
```

```
}
public class RunnableInterface {
    public static void main(String args[])
    {
        // Create an object of MyThread class.
        MyThread r=new MyThread();
        // Create an object of Thread class and pass reference variable th to
Thread class constructor.
        Thread t=new Thread(r);
        t.start();// This thread will execute statements inside run() method
of MyThread object.
        for(int i=1;i<=5;i++)
     {
         System.out.println("Parent Thread");
     }

    }
}
```

Out Put:-

```
Parent Thread
My Child Thread
My Child Thread
My Child Thread
My Child Thread
My Child Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
```

Thread Schedular:-

Thread scheduler in Java is the component of JVM that determines the execution order of multiple threads on a single processor (CPU).
It decides the order in which threads should run. This process is called **thread scheduling in Java**.

When a system with a single processor executes a program having multiple threads, CPU executes only a single thread at a particular time.

Other threads in the program wait in Runnable state for getting the chance of execution on CPU because at a time only one thread can get the chance to access the CPU.

The thread scheduler selects a thread for execution from runnable state. But there is no guarantee that which thread from runnable pool will be selected next to run by the thread scheduler.

Java runtime system mainly uses one of the following two strategies:

1. **Preemptive scheduling**
2. **Time-sliced scheduling**

Preemptive Scheduling

This scheduling is based on priority. Therefore, this scheduling is known as priority-based scheduling. In the priority-based scheduling algorithm, Thread scheduler uses the priority to decide which thread should be run.

If a thread with a higher priority exists in Runnable state (ready state), it will be scheduled to run immediately.

In case more than two threads have the same priority then CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

Time-Sliced Scheduling

The process of allocating time to threads is known as **time slicing in Java**. Time-slicing is based on non-priority scheduling. Under this scheduling, every running thread is executed for a fixed time period.
A currently running thread goes to the Runnable state when its time slice is elapsed and another thread gets time slots by CPU for execution.

With time-slicing, threads having lower priorities or higher priorities gets the same time slots for execution.

4. Thread priorities

Thread priority in Java is a number assigned to a thread that is used by *Thread scheduler* to decide which thread should be allowed to execute.
In Java, each thread is assigned a different priority that will decide the order (preference) in which it is scheduled for running.

Thread priorities are represented by a number from 1 to 10 that specifies the relative priority of one thread to another. The thread with the highest priority is selected by the scheduler to be executed first.

The default priority of a thread is 5. *Thread class in Java* also provides several priority constants to define the priority of a thread. These are:

3 constants defined in Thread class:

1. MIN_PRIORITY = 1
2. NORM_PRIORITY = 5
3. MAX_PRIORTY = 10

These constants are public, final, and static members(or) variables of the Thread class.

Thread scheduler selects the thread for execution on the first-come, first-serve basis. That is, the threads having equal priorities share the processor time on the first-come, first-serve basis.

When multiple threads are ready for execution, the highest priority thread is selected and executed by JVM. In case when a high priority thread stops, yields, or enters into the blocked state, a low priority thread starts executing.

If any high priority thread enters into the runnable state, it will preempt the currently running thread forcing it to move to the runnable state. Note that the highest priority thread always preempts any lower priority thread.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

Syntax:

```
ThreadName.setPriority(n);

 where, n is an integer value which ranges from 1 to 10.
```

Example program:-

```
class A extends Thread
{
   @Override
   public void run()
   {
      System.out.println(Thread.currentThread().getPriority());
      System.out.println(Thread.currentThread().getName());
   }
}
public class ThreadPriority {
   public static void main(String args[])
   {
      A a=new A();
      Thread t=new Thread(a,"c1");
      A a1=new A();
      Thread t1=new Thread(a1,"c2");
      t.setPriority(4);//If you assign priority value 17 its generate IllegalArgumentException
      t1.setPriority(3);
      System.out.println(t1.MIN_PRIORITY);
      Thread.currentThread().setPriority(7);//main thread priority assigning
      System.out.println(Thread.currentThread().getPriority());
      t.start();
      t1.start();
   }

}
```

O/P:-

```
7
4
3
c1
1
c2
```

5.Synchronizing threads:-

Thread synchronization in java is a way of programming several threads to carry out independent tasks easily. It is capable of controlling access to multiple threads to a particular shared resource. The main reason for using thread synchronization are as follows:

- To prevent interference between threads.
- To prevent the problem of consistency.

Types of Thread Synchronization

In Java, there are two types of thread synchronization:

1. Process synchronization
2. Thread synchronization

Process synchronization- The process means a program in execution and runs independently isolated from other processes. CPU time, memory, etc resources are allocated by the operating system.

Thread synchronization- It refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization. It is used because it avoids interference of thread and the problem of inconsistency.

In java, thread synchronization is further divided into two types:

- Mutual exclusive- it will keep the threads from interfering with each other while sharing any resources.
- Inter-thread communication- It is a mechanism in java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

Mutual Exclusive

It is used for preventing threads from interfering with each other and to keep distance between the threads. Mutual exclusive is achieved using the following:

- Synchronized Method
- Synchronized Block
- Static Synchronization

Lock Concept in Java

- Synchronization Mechanism developed by using the synchronized keyword in java language. It is built on top of the locking mechanism, this locking mechanism is taken care of by Java Virtual Machine (JVM). The synchronized keyword is only applicable for methods and blocks, it can't apply to classes and variables. Synchronized keyword in java creates a block of code is known as a critical section. To enter into the critical section thread needs to obtain the corresponding object's lock.

The problem without Synchronization:

Example Program:-

```
class E implements Runnable

{

   int available=5000,amount;
```

```
    E(int amount)
    {
        this.amount=amount;
    }
    @Override
    public void run()
    {
        String name=Thread.currentThread().getName();
        if(available>=amount)
        {
            System.out.println(name+" Withdraw Amount");
            available=available-amount;
        }
        else
        {
            System.out.println("Sorry amount not available");
        }
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}
public class ThreadSynchronization {
   public static void main(String args[])
   {
```

```
    E e=new E(5000);
    Thread t=new Thread(e);
    Thread t1=new Thread(e);
    Thread t2=new Thread(e);
      t.setName("Rahul");
      t1.setName("Sujan");
      t2.setName("Vishnu");
    t.start();
    t1.start();
    t2.start();
  }
}
```

OutPut:-

Vishnu withdraw Amount

Sujan withdraw Amount

Rahul withdraw Amount

Here we didn't use the synchronized keyword so multiple threads are executing at a time so in the output, vishnu is interfering with sujan and Rahul, and hence, we are getting inconsistent results

Java Synchronized Method

If we use the Synchronized keywords in any method then that method is Synchronized Method.

- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won't be released until the thread completes its function.

Syntax:

```
Acess_modifiers synchronized return_type method_name (Method_Parameters) {
// Code of the Method.
}
```

Example program:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }
   @Override
   public synchronized void run()
   {
      String name=Thread.currentThread().getName();
      if(available>=amount)
      {
         System.out.println(name+" Withdraw Amount");
         available=available-amount;
      }
      else
      {
         System.out.println("Sorry amount not available");
      }
      try{
         Thread.sleep(1000);
      }
      catch(InterruptedException e)
      {
      }
   }
}
```

```
public class ThreadSynchronization {

  public static void main(String args[])

  {

     E e=new E(5000);

     Thread t=new Thread(e);

     Thread t1=new Thread(e);

     Thread t2=new Thread(e);

      t.setName("Rahul");

      t1.setName("Sujan");

      t2.setName("Vishnu");

     t.start();

     t1.start();

     t2.start();

  }

}
```

OutPut:-

```
Rahul withdraw Amount

Sorry amount not available

Sorry amount not available
```

Here we used synchronized keywords. It helps to execute a single thread at a time. It is not allowing another thread to execute until the first one is completed, after completion of the first thread it allowed the second thread. Now we can see the output correctly only Rahul withdraw amount next other persons get no amount.

Synchronized Block

- Suppose you don't want to synchronize the entire method, you want to synchronize few lines of code in the method, then a synchronized block helps to synchronize those few lines of code. It will take the object as a parameter. It will work the same as Synchronized Method. In the case of synchronized method lock accessed is on the method but in the case of synchronized block lock accessed is on the object.

Syntax:

```
synchronized (object) {

//code of the block.

}
```

Program to understand the Synchronized Block:

Example:-

```
class E implements Runnable
{
    int available=5000,amount;
    E(int amount)
    {
        this.amount=amount;
    }

    @Override
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
        String name=Thread.currentThread().getName();
        synchronized(this){
        if(available>=amount)
        {
            System.out.println(name+" withdraw Amount");
            available=available-amount;
        }
        else
        {
            System.out.println("Sorry amount not available");
        }
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
```

```
        {
        }
        }
    }
}
public class ThreadSynchronization {
  public static void main(String args[])
  {
     E e=new E(1);
     Thread t=new Thread(e);
     Thread t1=new Thread(e);
     Thread t2=new Thread(e);
       t.setName("Rahul");
       t1.setName("Sujan");
       t2.setName("Vishnu");
     t.start();
     t1.start();
     t2.start();
  }
}
```

OutPut:-
Rahul
Vishnu
Sujan
Rahul withdraw Amount
Sorry amount not available
Sorry amount not available

In this example, we didn't synchronize the entire method but we synchronized few lines of code in the method. We got the results exactly as the synchronized method.

Static Synchronization

- In java, every object has a single lock (monitor) associated with it. The thread which is entering into synchronized method or synchronized block will get that lock, all other threads which are remaining to use the shared resources have to wait for the completion of the first thread and release of the lock.

- Suppose in the case of where we have more than one object, in this case, two separate threads will acquire the locks and enter into a synchronized block or synchronized method with a separate lock for each object at the same time. To avoid this, we will use static synchronization.
- In this, we will place synchronized keywords before the static method. In static synchronization, lock access is on the class not on object and Method.

Syntax:

```
synchronized static return_type method_name (Parameters) {

//code

}
```

Program without Static Synchronization:
Example:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }

   @Override
   public synchronized void run()
   {
     // System.out.println(Thread.currentThread().getName());
      String name=Thread.currentThread().getName();
      //synchronized(this){
      if(available>=amount)
      {
         System.out.println(name+" withdraw Amount");
         available=available-amount;
      }
      else
      {
         System.out.println("Sorry amount not available");
      }
      try{
         Thread.sleep(1000);
      }
      catch(InterruptedException e2)
      {

      }
      //}
   }
}
public class ThreadSynchronization {
  public static void main(String args[])
  {
    E e=new E(1);
```

```
      Thread t1=new Thread(e);
      Thread t2=new Thread(e);
      E e1=new E(1);
      Thread t3=new Thread(e1);
      Thread t4=new Thread(e1);
        t1.setName("Rahul");
        t2.setName("Sujan");
        t3.setName("Vishnu");
        t4.setName("Rushi");
      t1.start();
      t2.start();
      t3.start();
      t4.start();
  }
}
```

OutPut:-

```
Rahul withdraw Amount
Vishnu withdraw Amount
Sorry amount not available
Sorry amount not available
```

Program With static synchronized

Example:-

```
class E extends Thread
{
   static int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }

   public static synchronized void withdraw()
   {
      //System.out.println(Thread.currentThread().getName());
      String name=Thread.currentThread().getName();
      //synchronized(this){
      if(available>=amount)
      {
         System.out.println(name+" Withdraw Amount");
         available=available-amount;
      }
      else
      {
         System.out.println("Sorry amount not available");
      }
      //}
   }
    @Override
    public void run()
    {
        withdraw();
    }
```

```
}
public class ThreadSynchronization {
  public static void main(String args[])
  {
     E e=new E(5000);
     Thread t1=new Thread(e);
     Thread t2=new Thread(e);
     E e1=new E(5000);
     Thread t3=new Thread(e1);
     Thread t4=new Thread(e1);
      t1.setName("Rahul");
      t2.setName("Sujan");
      t3.setName("Vishnu");
      t4.setName("Rushi");
     t1.start();
     t2.start();
     t3.start();
     t4.start();
  }
}
```

OutPut:-

```
Rahul Withdraw Amount
Sorry amount not available
Sorry amount not available
Sorry amount not available
```

6.Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description

public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of t

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

1. **public final void** notify()

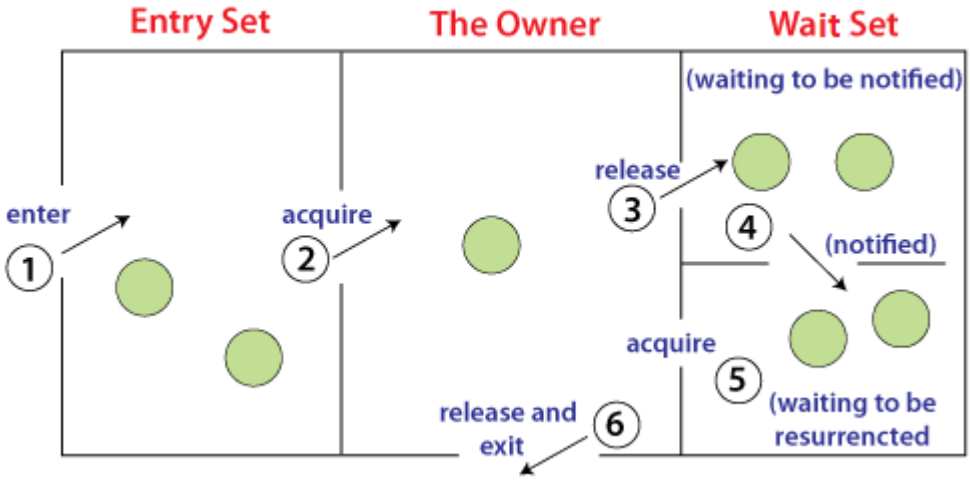
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

1. **public final void** notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is comple

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
1. class Customer{
2. int amount=10000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(this.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10. }
11. this.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
14.
15. synchronized void deposit(int amount){
16. System.out.println("going to deposit...");
17. this.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
```

```
20. }
21. }
22.
23. class Test{
24. public static void main(String args[]){
25. final Customer c=new Customer();
26. new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29. new Thread(){
30. public void run(){c.deposit(10000);}
31. }.start();
32.
33. }}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

Unit 4

Event Handling:-

1. **Events**
2. **Event sources**
3. **Event classes**
4. **Event Listeners**
5. **Delegation event model**
6. **Handling mouse events**
7. **Handling keyboard events**
8. **Adapter classes**

Event Handling:-

1. Events

What is an Event?

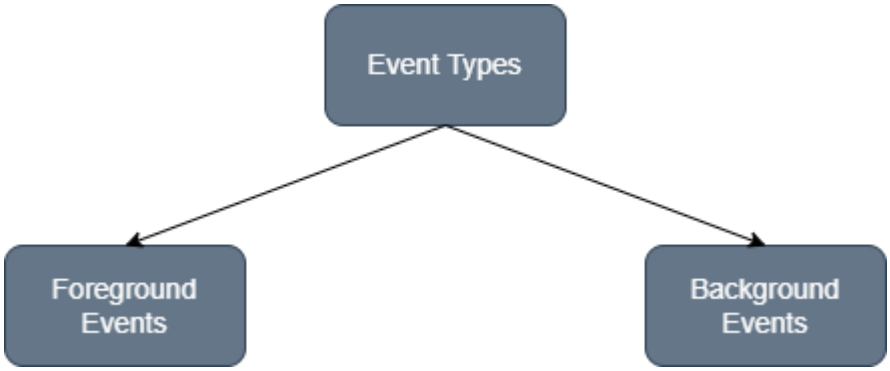
Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components.

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

Java provides a package **java.awt.event** that contains several event classes.

We can classify the events in the following two categories:

1. Foreground Events
2. Background Events



Foreground Events

Foreground events are those events that require user interaction to generate. In order to generate these foreground events, the user interacts with components in GUI. When a user clicks on a button, moves the cursor, and scrolls the scrollbar, an event will be fired.

Background Events

Background events don't require any user interaction. These events automatically generate in the background. OS failure, OS interrupts, operation completion, etc., are examples of background events.

2.Event sources

A source is an object that generates an event. Generally sources are components. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

Syntax:-

public void addTypeListener (TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener.

For example, the method that registers a keyboard event listener is called **addKeyListener( ).**

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Sources of Events:

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;
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.

Registration Methods:

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

 Button

o public void addActionListener(ActionListener a){}

 MenuItem

o public void addActionListener(ActionListener a){}

 TextField

o public void addActionListener(ActionListener a){}

o public void addTextListener(TextListener a){}

 TextArea

o public void addTextListener(TextListener a){}

 Checkbox

o public void addItemListener(ItemListener a){}

 Choice

o public void addItemListener(ItemListener a){}

 List

o public void addActionListener(ActionListener a){}

o public void addItemListener(ItemListener a){}

 Mouse

o public void addMouseListener(MouseListener a){}

4.Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements.

1.It must have been registered with one or more sources to receive notifications about specific types of events.

2. It must implement methods to receive and process these notifications.The methods that receive and process events are defined in a set of interfaces found in java.awt.event package.

An event listener registers with an event source to receive notifications about the events of a particular type. Various event listener interfaces defined in the java.awt.event

package are given below:

Interface	Description
ActionListener	Defines the actionPerformed() method to receive and process action events. *void actionPerformed(ActionEvent ae)*
MouseListener	Defines five methods to receive mouse events, such as when a mouse is clicked, pressed, released, enters, or exits a component *void mouseClicked(MouseEvent me)* *void mouseEntered(MouseEvent me)* *void mouseExited(MouseEvent me)* *void mousePressed(MouseEvent me)* *void mouseReleased(MouseEvent me)*
MouseMotionListener	Defines two methods to receive events, such as when a mouse is dragged or moved. *void mouseDragged(MouseEvent me)* *void mouseMoved(MouseEvent me)*
AdjustmentListner	Defines the adjustmentValueChanged() method to receive and process the adjustment events. *void adjustmentValueChanged(AdjustmentEvent ae)*
TextListener	Defines the textValueChanged() method to receive and process an event when the text value changes. *void textValueChanged(TextEvent te)*
WindowListener	Defines seven window methods to receive events. *void windowActivated(WindowEvent we)* *void windowClosed(WindowEvent we)* *void windowClosing(WindowEvent we)* *void windowDeactivated(WindowEvent we)* *void windowDeiconified(WindowEvent we)* *void windowIconified(WindowEvent we)* *void windowOpened(WindowEvent we)*
ItemListener	Defines the itemStateChanged() method when an item has been *void itemStateChanged(ItemEvent ie)*
WindowFocusListener	This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here: *void windowGainedFocus(WindowEvent we)* *void windowLostFocus(WindowEvent we)*
ComponentListener	This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here: *void componentResized(ComponentEvent ce)* *void componentMoved(ComponentEvent ce)* *void componentShown(ComponentEvent ce)* *void componentHidden(ComponentEvent ce)*

ContainerListener	This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here: *void componentAdded(ContainerEvent ce)* *void componentRemoved(ContainerEvent ce)*
FocusListener	This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here: *void focusGained(FocusEvent fe)* *void focusLost(FocusEvent fe)*
KeyListener	This interface defines three methods. *void keyPressed(KeyEvent ke)* *void keyReleased(KeyEvent ke)* *void keyTyped(KeyEvent ke)*

3.Event classes

Event Classes and Listener Interfaces:

The java.awt.event package provides many event classes and Listener interfaces for event handling. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the super class for all events. It's one constructor is shown here:

EventObject(Object *src*) - Here, *src* is the object that generates this event.

EventObject contains two methods:

Object getSource( ) - Object on which event initially occurred.
String toString( ) - toString( ) returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

The package **java.awt.event** defines many types of events that are generated by various user interface elements

Event Class	Description	Listener Interface
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.	ActionListener
AdjustmentEvent	Generated when a scroll bar is manipulated.	AdjustmentListener
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.	ComponentListener
ContainerEvent	Generated when a component is added to or removed from a container.	ContainerListener
FocusEvent	Generated when a component gains or losses keyboard focus.	FocusListener
InputEvent	Abstract super class for all component input event classes.	
ItemEvent	Generated when a check box or list item is clicked	ItemListener
KeyEvent	Generated when input is received from the keyboard.	KeyListener
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.	MouseListener and MouseMotionListener
TextEvent	Generated when the value of a text area or text field is changed.	TextListener
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.	WindowListener

Useful Methods of Component class:

Method	Description
public void add(Component c)	inserts a component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

The ActionEvent Class:

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK (Ex. Escape), and SHIFT_MASK.

ActionEvent has these three constructors:

- ActionEvent(Object src, int type, String cmd)
- ActionEvent(Object src, int type, String cmd, int modifiers)
- ActionEvent(Object src, int type, String cmd, long when, int modifiers)

You can obtain the command name for the invoking ActionEvent object by using the getActionCommand( ) method, shown here:

String getActionCommand( ) -Returns the command string associated with this action

The AdjustmentEvent Class:

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The ComponentEvent Class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The **getComponent( )** method returns the component that generated the event. It is shown here:

Component getComponent( )

The ContainerEvent Class:

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**.

The FocusEvent Class:

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

The InputEvent Class:

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the super class for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the InputEvent class defined the following eight values to represent the modifiers:

ALT_MASK	ALT_GRAPH_MASK	BUTTON2_MASK	BUTTON3_MASK
BUTTON1_MASK	CTRL_MASK	META_MASK	SHIFT_MASK

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	ALT_GRAPH_DOWN_MASK	BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK	BUTTON3_DOWN_MASK	CTRL_DOWN_MASK
META_DOWN_MASK	SHIFT_DOWN_MASK	

The KeyEvent Class

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED, KEY_RELEASED,** and **KEY_TYPED.**

The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing shift does not generate a character.

There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.

The MouseEvent Class:

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse
MOUSE_DRAGGED	The user dragged the mouse
MOUSE_ENTERED	The mouse entered a component
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

int getX( )
int getY( )

The TextEvent Class:

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant **TEXT_VALUE_CHANGED.**

The WindowEvent Class:

The **WindowEvent** class defines integer constants that can be used to identify different types of events:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window was iconified.
WINDOW_ICONIFIED	The window gained input focus.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.

What is Event Handling?

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

Steps to perform Event Handling Following steps are required to perform event handling:

1. Register the component with the Listener
2. Implement the concerned interface

5.Delegation event model

The Delegation Event model is defined to handle events in GUI programming languages. The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

Event Processing in Java

Java support event processing since Java 1.0. It provides support for AWT (Abstract Window Toolkit), which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing.

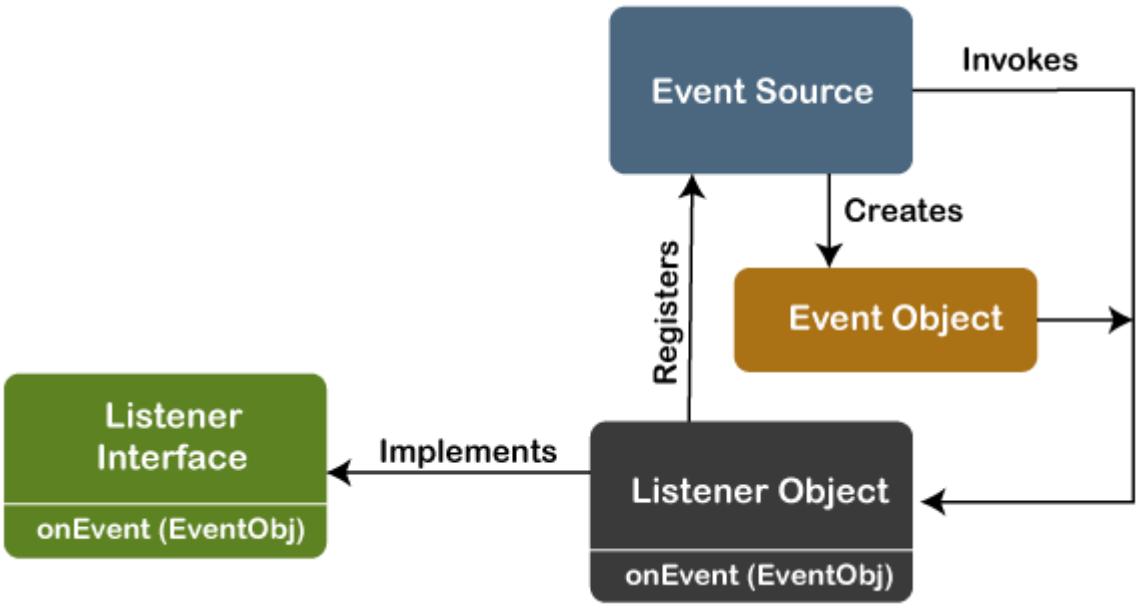


Fig.Delegation Event Model

But, the modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events.

In this model, a source generates an event and forwards it to one or more listeners. The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.

In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

Basically, an Event Model is based on the following three components:

- Events
- Events Sources
- Events Listeners

Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

1. **public void** addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as **addKeyListener()**. For the mouse event listener, the method will be called as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example:

1. **public void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersException

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

```
1. public void removeTypeListener(TypeListener e2?)
```

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

6.Handling mouse events

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods to handle mouse events.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

```
1. public abstract void mouseClicked(MouseEvent e);
2. public abstract void mouseEntered(MouseEvent e);
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. public abstract void mouseReleased(MouseEvent e);
```

Java MouseListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseListenerExample extends Frame implements MouseListener{
4.     Label l;
```

```
5.    MouseListenerExample(){
6.        addMouseListener(this);
7.
8.        l=new Label();
9.        l.setBounds(20,50,100,20);
10.       add(l);
11.       setSize(300,300);
12.       setLayout(null);
13.       setVisible(true);
14.   }
15.   public void mouseClicked(MouseEvent e) {
16.       l.setText("Mouse Clicked");
17.   }
18.   public void mouseEntered(MouseEvent e) {
19.       l.setText("Mouse Entered");
20.   }
21.   public void mouseExited(MouseEvent e) {
22.       l.setText("Mouse Exited");
23.   }
24.   public void mousePressed(MouseEvent e) {
25.       l.setText("Mouse Pressed");
26.   }
27.   public void mouseReleased(MouseEvent e) {
28.       l.setText("Mouse Released");
29.   }
30.public static void main(String[] args) {
31.   new MouseListenerExample();
32.}
33.}
```

Output:

7.Handling keyboard events

KeyboardEvent objects describe a user interaction with the keyboard; each event describes a single interaction between the user and a key (or combination of a key with modifier keys) on the keyboard. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods to handle mouse events.

Methods of KeyListener interface

The signature of 3 methods found in keyListener interface are given below:

void keyPressed(KeyEvent ke)

void keyReleased(KeyEvent ke)

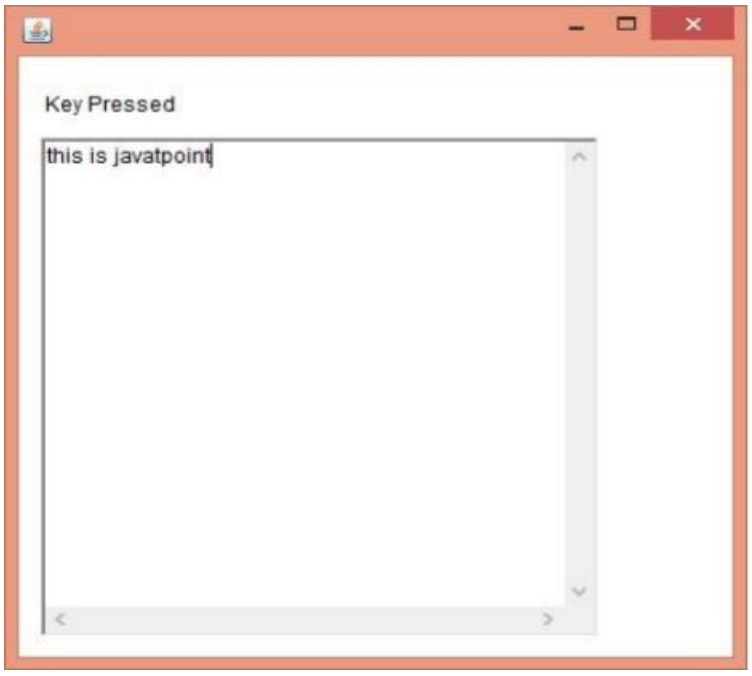
void keyTyped(KeyEvent ke)

Example program:-

```
import java.awt.*;
    import java.awt.event.*;
    public class KeyListenerExample extends Frame implements KeyListener{
        Label l;
        TextArea area;
        KeyListenerExample(){
            l=new Label();
            l.setBounds(20,50,100,20);
            area=new TextArea();
            area.setBounds(20,80,300, 300);
            area.addKeyListener(this);
            add(l);
           add(area);
            setSize(400,400);
            setLayout(null);
            setVisible(true);
        }
        public void keyPressed(KeyEvent e) {
            l.setText("Key Pressed");
        }
```

```
    public void keyReleased(KeyEvent e) {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

    public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```

OUTPUT:



8.Adapter classes

Java provides a special feature, called an *adapter class,* that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

For example,

MouseListener	MouseAdapter
void mouseClicked(MouseEvent me)	*void mouseClicked(MouseEvent me){ }*
void mouseEntered(MouseEvent me)	*void mouseEntered(MouseEvent me) { }*
void mouseExited(MouseEvent me)	*void mouseExited(MouseEvent me) { }*
void mousePressed(MouseEvent me)	*void mousePressed(MouseEvent me) { }*
void mouseReleased(MouseEvent me)	*void mouseReleased(MouseEvent me) { }*

Table: Commonly used Listener Interfaces implemented by Adapter Classes

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

The adapter classes are found in **java.awt.event package.**

Example Program:-

WindowAdapter Example

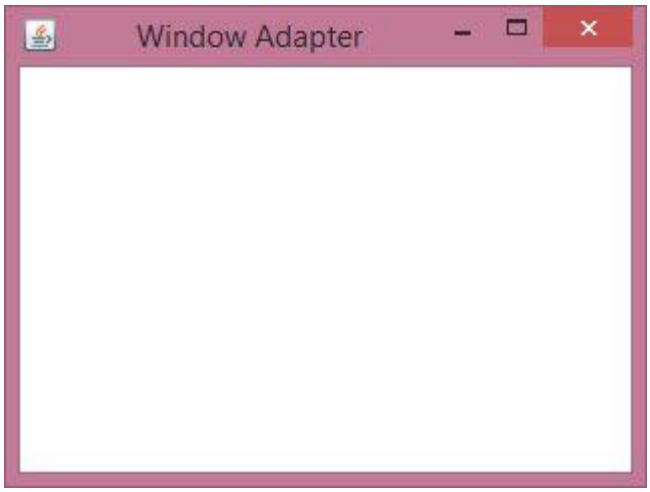
In the following example, we are implementing the WindowAdapter class of AWT and one its methods windowClosing() to close the frame window.

AdapterExample.java

```
1. // importing the necessary libraries
2. import java.awt.*;
3. import java.awt.event.*;
4.
5. public class AdapterExample {
6. // object of Frame
7.     Frame f;
8. // class constructor
9.     AdapterExample() {
10.// creating a frame with the title
```

```
11.       f = new Frame ("Window Adapter");
12.// adding the WindowListener to the frame
13.// overriding the windowClosing() method
14.//using annonymous class here
15.       f.addWindowListener (new WindowAdapter() {
16.          public void windowClosing (WindowEvent e) {
17.             f.dispose();
18.          }
19.       });
20.        // setting the size, layout and
21.       f.setSize (400, 400);
22.       f.setLayout (null);
23.       f.setVisible (true);
24.    }
25.
26.// main method
27.public static void main(String[] args) {
28.    new AdapterExample();
29.}
30.}
```

Output:



Q.Distinguish Event Listeners from Event Adapters.

Event Listeners:-

When we implement a listener interface in any class then we must have to implement all the methods declared in that interface because all the methods in an interface are abstract and must

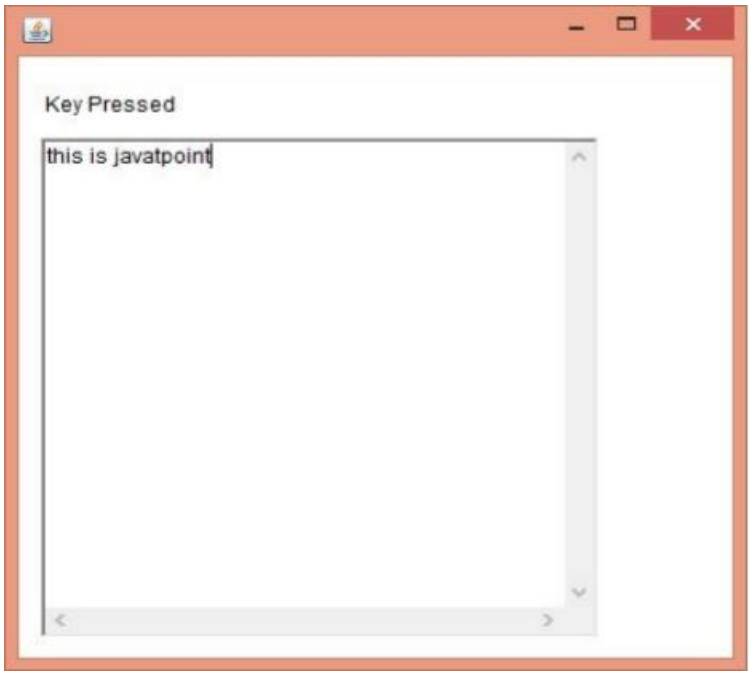
be override in class which implement it. For example consider the following program which demonstrates handling of key events by implementing listener interface.

KeyListenerExample.java

```
import java.awt.*;
  import java.awt.event.*;
  public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){
      l=new Label();
      l.setBounds(20,50,100,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
      add(l);
     add(area);
      setSize(400,400);
      setLayout(null);
      setVisible(true);
    }
    public void keyPressed(KeyEvent e) {
      l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e) {
      l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e) {
      l.setText("Key Typed");
    }

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

```
    }
}
```

OUTPUT:



Our above example for handling key events implements KeyListener interface so the class KeyEvent has to implement all the three methods listed below.

1. public void keyTyped(KeyEvent e)
2. public void keyPressed(KeyEvent e)
3. public void keyReleased(KeyEvent e)

This can be inconvenient because if we want to use only one or two methods in your program then? It is not suitable solution to implement all the methods every time even we don't need them. Adapter class makes it easy to deal with this situation. An adapter class provides empty implementations of all methods defined by that interface.

Adapter classs are very useful if you want to override only some of the methods defined by that interface. Here the names of Listener interface and corresponding interface are given which are in java.awt.event package.

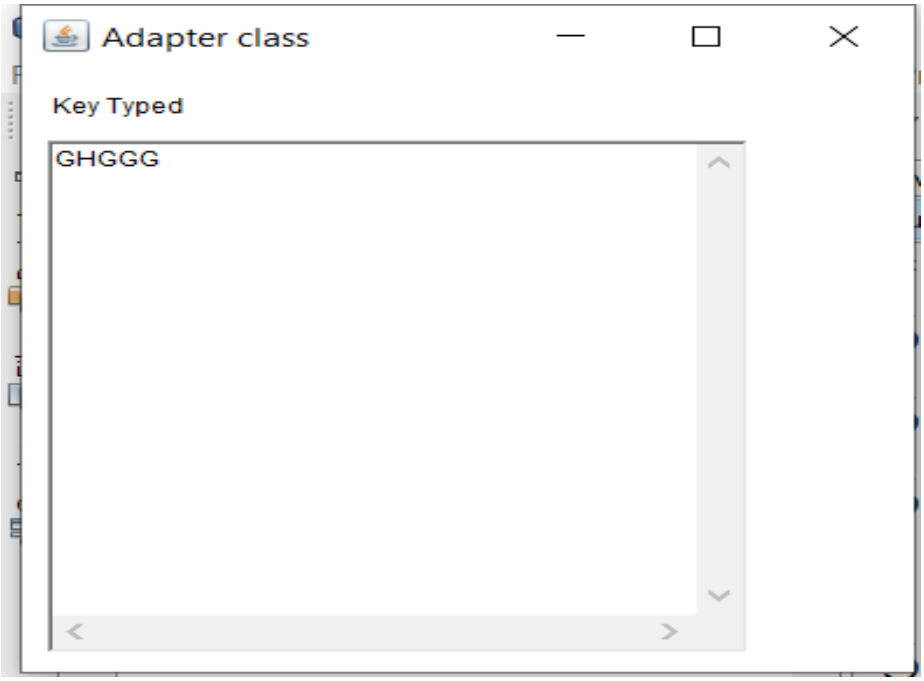
Adapter Class	Listener Interface
ComponentAdapter	ComponentListner
ContainerAdapter	ContainerListner
FocusAdapter	FocusListner
KeyAdapter	KeyListner
MouseAdapter	MouseListner
MouseMotionAdapter	MouseMotionListner
WindowAdapter	WindowListner

Now consider a situation in which we want to perform any action only when key Typed then we should have to override keyTyped() method. In this case if we use the listener interface then we must have to implements all the above three methods, the adapter class KeyAdapter will minimize programmer's work. Following example demonstrate the use of Adapter class in place of Listener interface.

KeyListenerExample.java

```
import java.awt.*;
   import java.awt.event.*;
   public class KeyListenerExample extends KeyAdapter{
      Frame f;
      Label l;
      TextArea area;
      KeyListenerExample(){
         f=new Frame("Adapter class");
         l=new Label();
         l.setBounds(20,50,100,20);
         area=new TextArea();
         area.setBounds(20,80,300, 300);
         area.addKeyListener(this);
```

```
        f.add(l);
f.add(area);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }

    public void keyTyped(KeyEvent e) {
        l.setText("Key Typed");
    }

    public static void main(String[] args) {
        new KeyListenerExample();
    }
}
```



AWT

1. **Class hierarchy**
2. **User interface components**
    a) **Labels**
    b) **Buttons**
    c) **Scrollbars**
    d) **Text components**
    e) **Checkbox**
    f) **Checkbox groups**
    g) **Choices**
    h) **Lists**
    i) **Panels**
    j) **Scroll pane**
    k) **Dialogs**
    l) **Menu bar**

AWT(Abstract Window Toolkit):-

Java AWT (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Why AWT is platform dependent?

Java AWT calls the native platform calls the native platform (operating systems) subroutine for creating API components like TextField, CheckBox, button, etc.

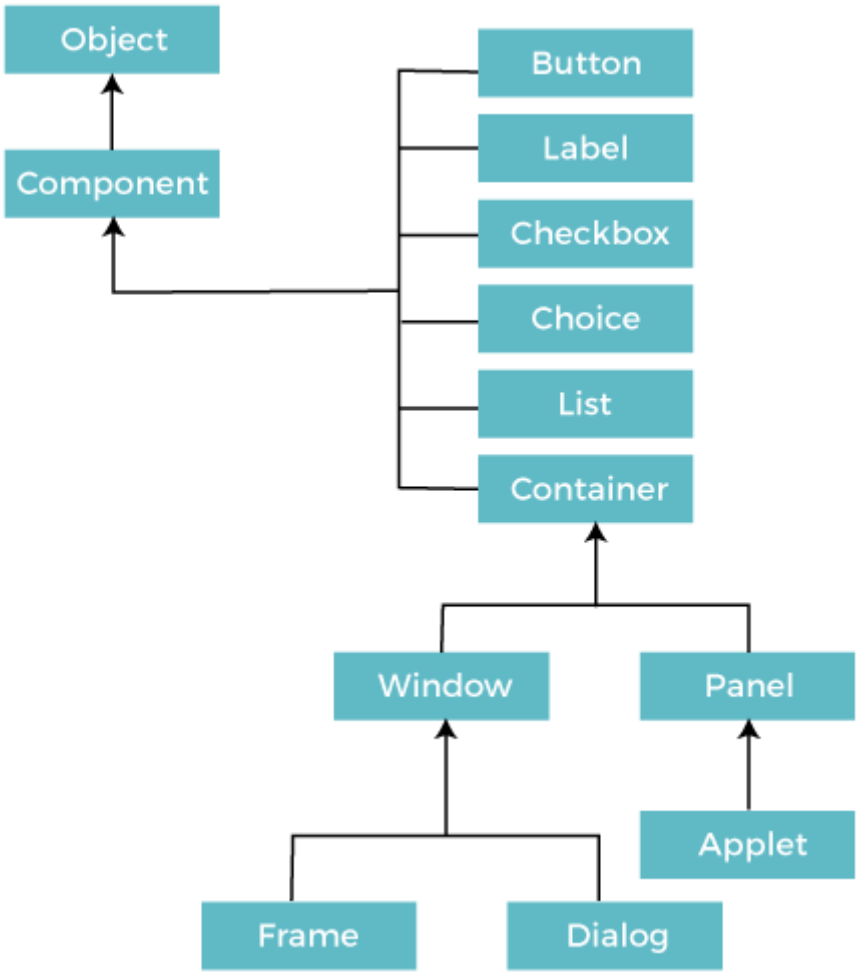
For example, an AWT GUI with components like TextField, label and button will have different look and feel for the different platforms like Windows, MAC OS, and Unix. The reason for this is the platforms

have different view for their native components and AWT directly calls the native subroutine that creates those components.

In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS.

1.Class hierarchy

The hierarchy of Java AWT classes are given below.



Components

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Note: A container itself is a component (see the above diagram), therefore we can add a container inside container.

Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

Useful Methods of Component Class

Method	Description
public void add(Component c)	Inserts a component on this component.
public void setSize(int width,int height)	Sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	Defines the layout manager for the component.
public void setVisible(boolean status)	Changes the visibility of the component, by default false.

Java AWT Example

To create simple AWT example, you need a frame. There are two ways to create a GUI using Frame in AWT.

1. By extending Frame class (**inheritance**)
2. By creating the object of Frame class (**association**)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

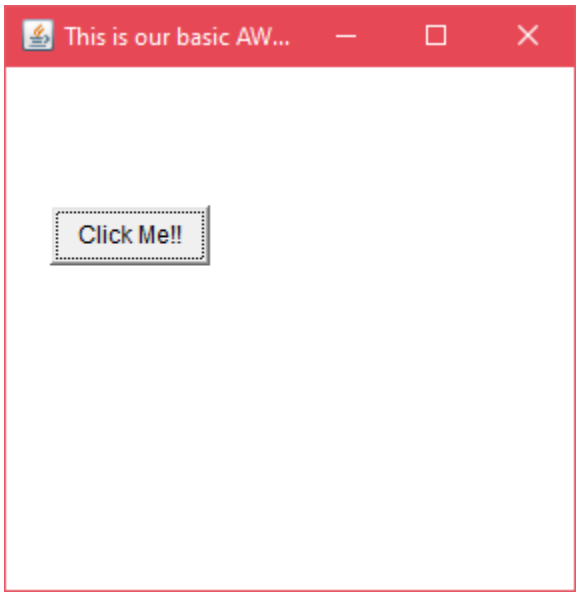
AWTExample1.java

```
// importing Java AWT class
import java.awt.*;

// extending Frame class to our class AWTExample1
public class AWTExample1 extends Frame {

  // initializing using constructor
  AWTExample1() {

    // creating a button
    Button b = new Button("Click Me!!");

    // setting button position on screen
    b.setBounds(30,100,80,30);

    // adding button into frame
    add(b);

    // frame size 300 width and 300 height
    setSize(300,300);

    // setting the title of Frame
    setTitle("This is our basic AWT example");

    // no layout manager
    setLayout(null);

    // now frame will be visible, by default it is not visible
    setVisible(true);
}

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

```
34.
35. // creating instance of Frame class
36. AWTExample1 f = new AWTExample1();
37.
38. }
39.
40. }
```

The setBounds(int x-axis, int y-axis, int width, int height) method is used in the above example that sets the position of the awt button.

Output:



AWT Example by Association

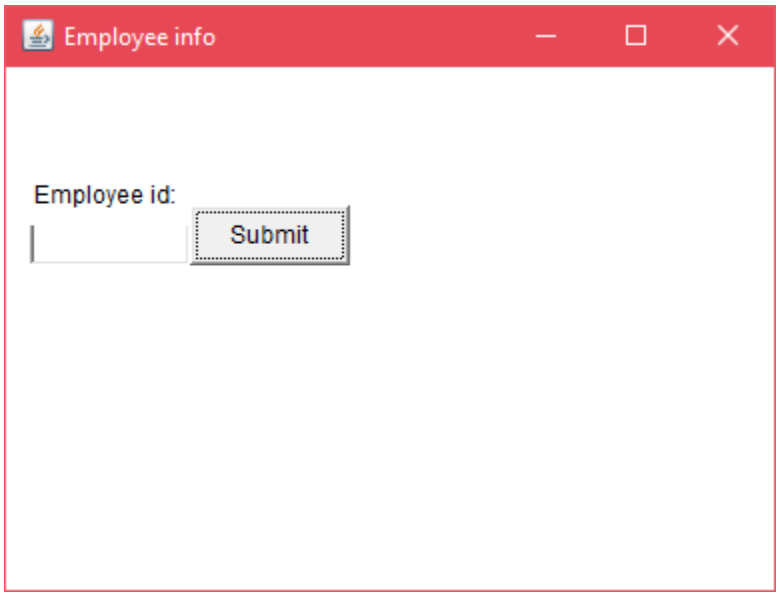
Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

AWTExample2.java

```
1. // importing Java AWT class
2. import java.awt.*;
3.
4. // class AWTExample2 directly creates instance of Frame class
5. class AWTExample2 {
6.
7.   // initializing using constructor
8.   AWTExample2() {
9.
10.    // creating a Frame
11.    Frame f = new Frame();
12.
```

```
13.      // creating a Label
14.      Label l = new Label("Employee id:");
15.
16.      // creating a Button
17.      Button b = new Button("Submit");
18.
19.      // creating a TextField
20.      TextField t = new TextField();
21.
22.      // setting position of above components in the frame
23.      l.setBounds(20, 80, 80, 30);
24.      t.setBounds(20, 100, 80, 30);
25.      b.setBounds(100, 100, 80, 30);
26.
27.      // adding components into frame
28.      f.add(b);
29.      f.add(l);
30.      f.add(t);
31.
32.      // frame size 300 width and 300 height
33.      f.setSize(400,300);
34.
35.      // setting the title of frame
36.      f.setTitle("Employee info");
37.
38.      // no layout
39.      f.setLayout(null);
40.
41.      // setting visibility of frame
42.      f.setVisible(true);
43. }
44.
45. // main method
46. public static void main(String args[]) {
47.
48. // creating instance of Frame class
49. AWTExample2 awt_obj = new AWTExample2();
50.
51. }
52.
53. }
```

Output:



3. User interface components

a)Labels

The object of the Label class is a component for placing text in a container. It is used to display a single line of **read only text**. The text can be changed by a programmer but a user cannot edit it directly.

It is called a passive control as it does not create any event when it is accessed. To create a label, we need to create the object of **Label** class.

AWT Label Class Declaration

1. **public class** Label **extends** Component **implements** Accessible

AWT Label Fields

The java.awt.Component class has following fields:

1. **static int LEFT:** It specifies that the label should be left justified.
2. **static int RIGHT:** It specifies that the label should be right justified.
3. **static int CENTER:** It specifies that the label should be placed in center.

Label class Constructors

Sr. no.	Constructor	Description

1.	Label()	It constructs an empty label.
2.	Label(String text)	It constructs a label with the given string (left justified by default).
3.	Label(String text, int alignement)	It constructs a label with the specified string and the specified alignment.

Label Class Methods

Specified

Sr. no.	Method name	Description
1.	void setText(String text)	It sets the texts for label with the specified text.
2.	Void setAlignment(int alignment)	It sets the alignment for label with the specified alignment.
3.	String getText()	It gets the text of the label
4.	int getAlignment()	It gets the current alignment of the label.

Method inherited

The above methods are inherited by the following classes:

- java.awt.Component
- java.lang.Object

Java AWT Label Example

In the following example, we are creating two labels l1 and l2 using the Label(String text) constructor and adding them into the frame.

LabelExample.java

```
1. import java.awt.*;
2.
3. public class LabelExample {
4. public static void main(String args[]){
5.
6.    // creating the object of Frame class and Label class
7.    Frame f = new Frame ("Label example");
```

```
8.      Label l1, l2;
9.
10.     // initializing the labels
11.     l1 = new Label ("First Label.");
12.     l2 = new Label ("Second Label.");
13.
14.     // set the location of label
15.     l1.setBounds(50, 100, 100, 30);
16.     l2.setBounds(50, 150, 100, 30);
17.
18.     // adding labels to the frame
19.     f.add(l1);
20.     f.add(l2);
21.
22.     // setting size, layout and visibility of frame
23.     f.setSize(400,400);
24.     f.setLayout(null);
25.     f.setVisible(true);
26. }
27. }
```

Output:

b)Buttons

Java AWT Button

A button is basically a control component with a label that generates an event when pushed. The **Button** class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

When we press a button and release it, AWT sends an instance of **ActionEvent** to that button by calling **processEvent** on the button. The **processEvent** method of the button receives the all the

events, then it passes an action event by calling its own method **processActionEvent**. This method passes the action event on to action listeners that are interested in the action events generated by the button.

To perform an action on a button being pressed and released, the **ActionListener** interface needs to be implemented. The registered new listener can receive events from the button by calling **addActionListener** method of the button.

AWT Button Class Declaration

1. **public class** Button **extends** Component **implements** Accessible

Button Class Constructors

Following table shows the types of Button class constructors

Sr. no.	Constructor	Description
1.	Button( )	It constructs a new button with an empty string i.e. it has no label.
2.	Button (String text)	It constructs a new button with given string as its label.

Button Class Methods

Sr. no.	Method	Description
1.	void setText (String text)	It sets the string message on the button
2.	String getText()	It fetches the String message on the button.
3.	void setLabel (String label)	It sets the label of button with the specified string.
4.	String getLabel()	It fetches the label of the button.
7.	void addActionListener(ActionListener l)	It adds the specified action listener to get the action events from the button.
8.	String getActionCommand()	It returns the command name of the action event fired by the button.

9.	void removeActionListener (ActionListener l)	It removes the specified action listener so that it no longer receives action events from the button.
10.	void setActionCommand(String command)	It sets the command name for the action event given by the button.

Note: The Button class inherits methods from java.awt.Component and java.lang.Object classes.

Java AWT Button Example

Example 1:

ButtonExample.java

```
1. import java.awt.*;
2. public class ButtonExample {
3. public static void main (String[] args) {
4.
5.     // create instance of frame with the label
6.     Frame f = new Frame("Button Example");
7.
8.     // create instance of button with label
9.     Button b = new Button("Click Here");
10.
11.    // set the position for the button in frame
12.    b.setBounds(50,100,80,30);
13.
14.    // add button to the frame
15.    f.add(b);
16.    // set size, layout and visibility of frame
17.    f.setSize(400,400);
18.    f.setLayout(null);
19.    f.setVisible(true);
20. }
21. }
```

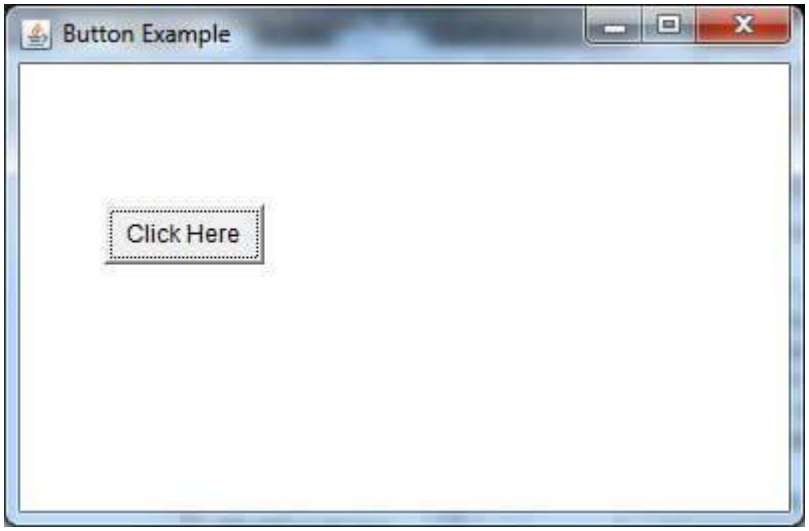
To compile the program using command prompt type the following commands

```
1. C:\Users\Anurati\Desktop\abcDemo>javac ButtonExample.java
```

If there's no error, we can execute the code using:

```
1. C:\Users\Anurati\Desktop\abcDemo>java ButtonExample
```

Output:



c)Scrollbars

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

It can be added to top-level container like Frame or a component like Panel. The Scrollbar class extends the **Component** class.

AWT Scrollbar Class Declaration

1. **public class** Scrollbar **extends** Component **implements** Adjustable, Accessible

Scrollbar Class Fields

The fields of java.awt.Image class are as follows:

- **static int HORIZONTAL** - It is a constant to indicate a horizontal scroll bar.
- **static int VERTICAL** - It is a constant to indicate a vertical scroll bar.

Scrollbar Class Constructors

Sr. no.	Constructor	Description
1	Scrollbar()	Constructs a new vertical scroll bar.
2	Scrollbar(int orientation)	Constructs a new scroll bar with the specified orientat
3	Scrollbar(int orientation, int value, int visible, int minimum, int maximum)	Constructs a new scroll bar with the specified orient value, visible amount, and minimum and maximum va

Where the parameters,

- **orientation**: specifiey whether the scrollbar will be horizontal or vertical.
- **Value**: specify the starting position of the knob of Scrollbar on its track.
- **Minimum**: specify the minimum width of track on which scrollbar is moving.
- **Maximum**: specify the maximum width of track on which scrollbar is moving.

Method Inherited by Scrollbar

The methods of Scrollbar class are inherited from the following classes:

- java.awt.Component
- java.lang.Object

Scrollbar Class Methods

Sr. no.	Method name	Description
1.	int getMaximum()	It gets the maximum value of the scroll bar.
2.	int getMinimum()	It gets the minimum value of the scroll bar.
3.	int getOrientation()	It returns the orientation of scroll bar.
4.	int getUnitIncrement()	It fetches the unit increment of the scroll bar.
5.	int getValue()	It fetches the current value of scroll bar.
6.	int getVisibleAmount()	It fetches the visible amount of scroll bar.
7.	void setBlockIncrement(int v)	It sets the block increment from scroll bar.
8.	void setMaximum (int newMaximum)	It sets the maximum value of the scroll bar.
9.	void setMinimum (int newMinimum)	It sets the minimum value of the scroll bar.
10.	void setOrientation (int orientation)	It sets the orientation for the scroll bar.
11.	void setUnitIncrement(int v)	It sets the unit increment for the scroll bar.
12.	void setValue (int newValue)	It sets the value of scroll bar with the given argument value.
13.	void setValues (int value, int visible, int minimum, int maximum)	It sets the values of four properties for scroll bar: value, visible amount, minimum and maximum.
14.	void setVisibleAmount (int newAmount)	It sets the visible amount of the scroll bar.

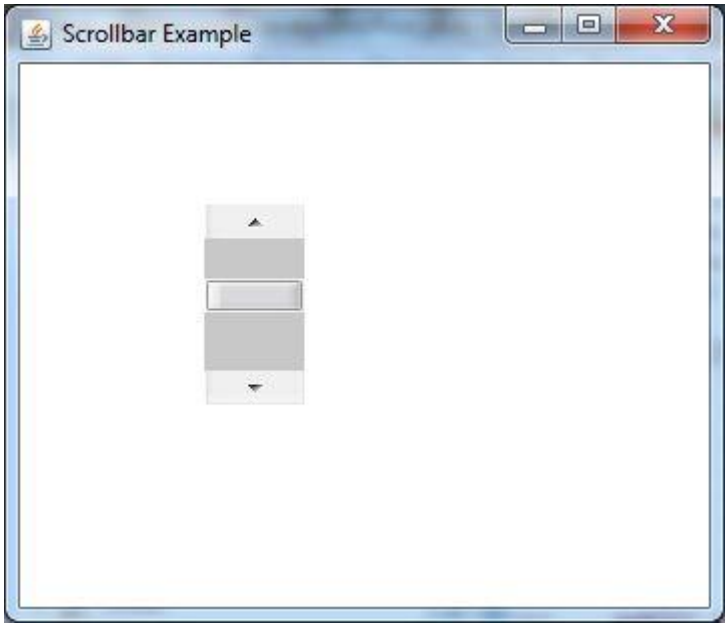
Java AWT Scrollbar Example

In the following example, we are creating a scrollbar using the Scrollbar() and adding it into the Frame.

ScrollbarExample1.java

```
1. // importing awt package
2. import java.awt.*;
3.
4. public class ScrollbarExample1 {
5.
6. // class constructor
7. ScrollbarExample1() {
8.
9.         // creating a frame
10.             Frame f = new Frame("Scrollbar Example");
11.        // creating a scroll bar
12.             Scrollbar s = new Scrollbar();
13.
14.        // setting the position of scroll bar
15.             s.setBounds (100, 100, 50, 100);
16.
17.        // adding scroll bar to the frame
18.             f.add(s);
19.
20.        // setting size, layout and visibility of frame
21.             f.setSize(400, 400);
22.             f.setLayout(null);
23.             f.setVisible(true);
24. }
25.
26. // main method
27. public static void main(String args[]) {
28.         new ScrollbarExample1();
29. }
30. }
```

Output:



d)Text components

Text Components class is extends by

1.TextField

2.TextArea

1.TextField

The object of a **TextField** class is a text component that allows a user to enter a single line text and edit it. It inherits **TextComponent** class, which further inherits **Component** class.

When we enter a key in the text field (like key pressed, key released or key typed), the event is sent to **TextField**. Then the **KeyEvent** is passed to the registered **KeyListener**. It can also be done using **ActionEvent**; if the ActionEvent is enabled on the text field, then the ActionEvent may be fired by pressing return key. The event is handled by the **ActionListener** interface.

AWT TextField Class Declaration

1. **public class** TextField **extends** TextComponent

TextField Class constructors

Sr. no.	Constructor	Description
1.	TextField()	It constructs a new text field component.

2.	TextField(String text)	It constructs a new text field initialized with the given string text to be displayed.
3.	TextField(int columns)	It constructs a new textfield (empty) with given number of columns.
4.	TextField(String text, int columns)	It constructs a new text field with the given text and given number of columns (width).

TextField Class Methods

Sr. no.	Method name	Description
1.	boolean echoCharIsSet()	It tells whether text field has character set for echoing or not.
2.	int getColumns()	It fetches the number of columns in text field.
3.	char getEchoChar()	It fetches the character that is used for echoing.
4.	Dimension getMinimumSize()	It fetches the minimum dimensions for the text field.
5.	Dimension getMinimumSize(int columns)	It fetches the minimum dimensions for the text field with specified number of columns.
6.	Dimension getPreferredSize()	It fetches the preferred size of the text field.
7.	Dimension getPreferredSize(int columns)	It fetches the preferred size of the text field with specified number of columns.
8.	void setColumns(int columns)	It sets the number of columns in text field.
9.	void setEchoChar(char c)	It sets the echo character for text field.
10.	void setText(String t)	It sets the text presented by this text component to the specified text.

Method Inherited

The AWT TextField class inherits the methods from below classes:

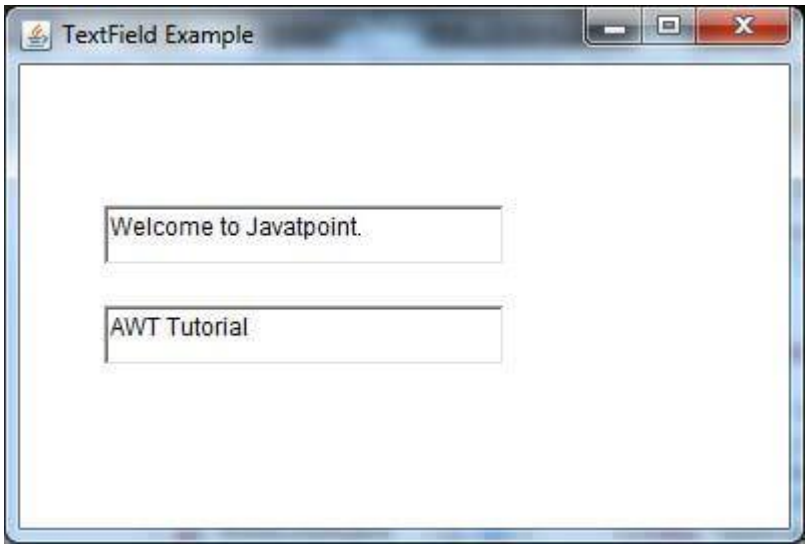
1. java.awt.TextComponent
2. java.awt.Component
3. java.lang.Object

Java AWT TextField Example

TextFieldExample1.java

```
1. // importing AWT class
2. import java.awt.*;
3. public class TextFieldExample1 {
4.     // main method
5.     public static void main(String args[]) {
6.     // creating a frame
7.     Frame f = new Frame("TextField Example");
8.
9.     // creating objects of textfield
10.    TextField t1, t2;
11.    // instantiating the textfield objects
12.    // setting the location of those objects in the frame
13.    t1 = new TextField("Welcome to Javatpoint.");
14.    t1.setBounds(50, 100, 200, 30);
15.    t2 = new TextField("AWT Tutorial");
16.    t2.setBounds(50, 150, 200, 30);
17.    // adding the components to frame
18.    f.add(t1);
19.    f.add(t2);
20.    // setting size, layout and visibility of frame
21.    f.setSize(400,400);
22.    f.setLayout(null);
23.    f.setVisible(true);
24. }
25. }
```

Output:



2.TextArea

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

The text area allows us to type as much text as we want. When the text in the text area becomes larger than the viewable area, the scroll bar appears automatically which helps us to scroll the text up and down, or right and left.

AWT TextArea Class Declaration

1. **public class** TextArea **extends** TextComponent

Fields of TextArea Class

The fields of java.awt.TextArea class are as follows:

- **static int SCROLLBARS_BOTH** - It creates and displays both horizontal and vertical scrollbars.
- **static int SCROLLBARS_HORIZONTAL_ONLY** - It creates and displays only the horizontal scrollbar.
- **static int SCROLLBARS_VERTICAL_ONLY** - It creates and displays only the vertical scrollbar.
- **static int SCROLLBARS_NONE** - It doesn't create or display any scrollbar in the text area.

Class constructors:

Sr. no.	Constructor	Description
1.	TextArea()	It constructs a new and empty text area with no text in it.
2.	TextArea (int row, int column)	It constructs a new text area with specified number of rows and columns and empty string as text.
3.	TextArea (String text)	It constructs a new text area and displays the specified text in it.
4.	TextArea (String text, int row, int column)	It constructs a new text area with the specified text in the text area and specified number of rows and columns.
5.	TextArea (String text, int row, int column, int scrollbars)	It construcst a new text area with specified text in text area and specified number of rows and columns and visibility.

Methods Inherited

The methods of TextArea class are inherited from following classes:

- java.awt.TextComponent
- java.awt.Component
- java.lang.Object

TetArea Class Methods

Sr. no.	Method name	Description
1.	int getColumns()	It returns the number of columns of text area.
2.	Dimension getMinimumSize()	It determines the minimum size of a text area.
3.	Dimension getMinimumSize(int rows, int columns)	It determines the minimum size of a text area with the given number of rows and columns.
4.	Dimension getPreferredSize()	It determines the preferred size of a text area.
5.	Dimension preferredSize(int rows, int columns)	It determines the preferred size of a text area with given number of rows and columns.
6.	int getRows()	It returns the number of rows of text area.
7.	int getScrollbarVisibility()	It returns an enumerated value that indicates which scroll bars the text area uses.
8.	void insert(String str, int pos)	It inserts the specified text at the specified position in this text area.
9.	void replaceRange(String str, int start, int end)	It replaces text between the indicated start and end positions with the specified replacement text.
10.	void setColumns(int columns)	It sets the number of columns for this text area.
11.	void setRows(int rows)	It sets the number of rows for this text area.

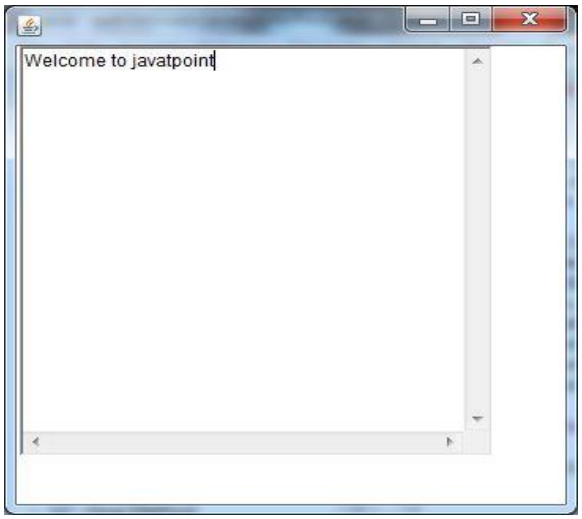
Java AWT TextArea Example

The below example illustrates the simple implementation of TextArea where we are creating a text area using the constructor TextArea(String text) and adding it to the frame.

TextAreaExample .java

```
1. //importing AWT class
2. import java.awt.*;
3. public class TextAreaExample
4. {
5. // constructor to initialize
6.     TextAreaExample() {
7. // creating a frame
8.         Frame f = new Frame();
9. // creating a text area
10.             TextArea area = new TextArea("Welcome to javatpoint");
11. // setting location of text area in frame
12.         area.setBounds(10, 30, 300, 300);
13. // adding text area to frame
14.         f.add(area);
15. // setting size, layout and visibility of frame
16.         f.setSize(400, 400);
17.         f.setLayout(null);
18.         f.setVisible(true);
19.     }
20. // main method
21. public static void main(String args[])
22. {
23.    new TextAreaExample();
24. }
25. }
```

Output:



e)Checkbox

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

AWT Checkbox Class Declaration

1. **public class** Checkbox **extends** Component **implements** ItemSelectable, Accessible

Checkbox Class Constructors

Sr. no.	Constructor	Description
1.	Checkbox()	It constructs a checkbox with no string as the label.
2.	Checkbox(String label)	It constructs a checkbox with the given label.
3.	Checkbox(String label, boolean state)	It constructs a checkbox with the given label and sets the given state.
4.	Checkbox(String label, boolean state, CheckboxGroup group)	It constructs a checkbox with the given label, set the given state in the specified checkbox group.
5.	Checkbox(String label, CheckboxGroup group, boolean state)	It constructs a checkbox with the given label, in the given checkbox group and set to the specified state.

Method inherited by Checkbox

The methods of Checkbox class are inherited by following classes:

- java.awt.Component
- java.lang.Object

Checkbox Class Methods

Sr. no.	Method name	Description
1.	void addItemListener(ItemListener IL)	It adds the given item listener to get the item events from the checkbox.
2.	CheckboxGroup getCheckboxGroup()	It determines the group of checkbox.
3.	String getLabel()	It fetched the label of checkbox.

4.	boolean getState()	It returns true if the checkbox is on, else returns off.
5.	void removeItemListener(ItemListener l)	It removes the specified item listener so that the item listener doesn't receive item events from the checkbox anymore.
6.	void setCheckboxGroup(CheckboxGroup g)	It sets the checkbox's group to the given checkbox.
7.	void setLabel(String label)	It sets the checkbox's label to the string argument.
8.	void setState(boolean state)	It sets the state of checkbox to the specified state.

Java AWT Checkbox Example

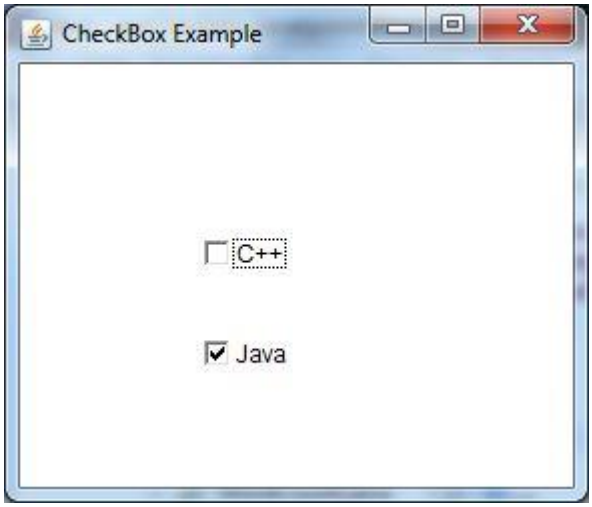
In the following example we are creating two checkboxes using the Checkbox(String label) constructo and adding them into the Frame using add() method.

CheckboxExample1.java

```
1. // importing AWT class
2. import java.awt.*;
3. public class CheckboxExample1
4. {
5. // constructor to initialize
6.     CheckboxExample1() {
7. // creating the frame with the title
8.       Frame f = new Frame("Checkbox Example");
9. // creating the checkboxes
10.       Checkbox checkbox1 = new Checkbox("C++");
11.       checkbox1.setBounds(100, 100,  50, 50);
12.       Checkbox checkbox2 = new Checkbox("Java", true);
13. // setting location of checkbox in frame
14. checkbox2.setBounds(100, 150,  50, 50);
15. // adding checkboxes to frame
16.       f.add(checkbox1);
17.       f.add(checkbox2);
18.
19. // setting size, layout and visibility of frame
20.       f.setSize(400,400);
21.       f.setLayout(null);
22.       f.setVisible(true);
```

```
23.     }
24. // main method
25. public static void main (String args[])
26. {
27.     new CheckboxExample1();
28. }
29. }
```

Output:



f)Checkbox groups

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

Note: CheckboxGroup enables you to create radio buttons in AWT. There is no special control for creating radio buttons in AWT.

AWT CheckboxGroup Class Declaration

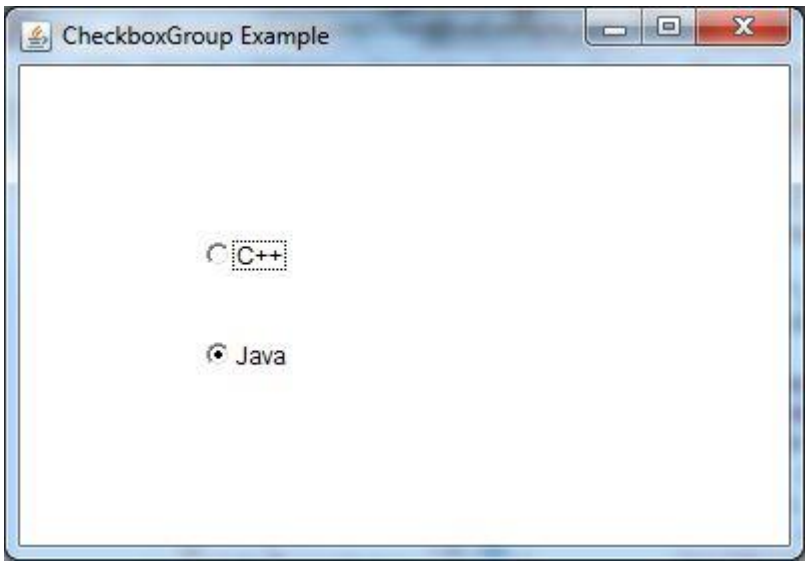
```
1. public class CheckboxGroup extends Object implements Serializable
```

Java AWT CheckboxGroup Example

```
1. import java.awt.*;
2. public class CheckboxGroupExample
3. {
4.         CheckboxGroupExample(){
5.         Frame f= new Frame("CheckboxGroup Example");
6.          CheckboxGroup cbg = new CheckboxGroup();
7.          Checkbox checkBox1 = new Checkbox("C++", cbg, false);
8.          checkBox1.setBounds(100,100, 50,50);
```

```
9.          Checkbox checkBox2 = new Checkbox("Java", cbg, true);
10.         checkBox2.setBounds(100,150, 50,50);
11.         f.add(checkBox1);
12.         f.add(checkBox2);
13.         f.setSize(400,400);
14.         f.setLayout(null);
15.         f.setVisible(true);
16.     }
17. public static void main(String args[])
18. {
19.     new CheckboxGroupExample();
20. }
21. }
```

Output:



g)Choice

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

AWT Choice Class Declaration

```
1. public class Choice extends Component implements ItemSelectable, Accessible
```

Choice Class constructor

Sr. no.	Constructor	Description
1.	Choice()	It constructs a new choice menu.

Methods inherited by class

The methods of Choice class are inherited by following classes:

- java.awt.Component
- java.lang.Object

Choice Class Methods

Sr. no.	Method name	Description
1.	void add(String item)	It adds an item to the choice menu.
2.	void addItemListener(ItemListener l)	It adds the item listener that receives item events from the choice menu.
3.	String getItem(int index)	It gets the item (string) at the given index position in the choice menu.
4.	int getItemCount()	It returns the number of items of the choice menu.
5.	int getSelectedIndex()	Returns the index of the currently selected item.
6.	String getSelectedItem()	Gets a representation of the current choice as a string.
7.	void insert(String item, int index)	Inserts the item into this choice at the specified position.
8.	void remove(int position)	It removes an item from the choice menu at the given index position.
9.	void remove(String item)	It removes the first occurrence of the item from choice menu.
10.	void removeAll()	It removes all the items from the choice menu.
11.	void removeItemListener (ItemListener l)	It removes the mentioned item listener. Thus is doesn't receive item events from the choice menu anymore.
12.	void select(int pos)	It changes / sets the selected item in the choice menu to the item at given index position.

13.	void select(String str)	It changes / sets the selected item in the choice menu to the item whose string value is equal to string specified in the argument.

Java AWT Choice Example

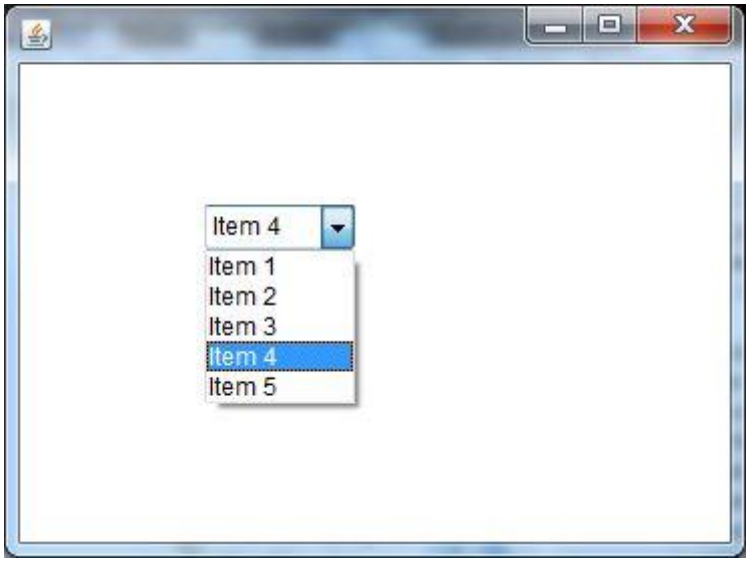
In the following example, we are creating a choice menu using Choice() constructor. Then we add 5 items to the menu using add() method and Then add the choice menu into the Frame.

ChoiceExample1.java

```
1. // importing awt class
2. import java.awt.*;
3. public class ChoiceExample1 {
4.
5.     // class constructor
6.     ChoiceExample1() {
7.
8.     // creating a frame
9.     Frame f = new Frame();
10.
11.    // creating a choice component
12.    Choice c = new Choice();
13.
14.    // setting the bounds of choice menu
15.    c.setBounds(100, 100, 75, 75);
16.
17.    // adding items to the choice menu
18.    c.add("Item 1");
19.    c.add("Item 2");
20.    c.add("Item 3");
21.    c.add("Item 4");
22.    c.add("Item 5");
23.
24.    // adding choice menu to frame
25.    f.add(c);
26.
27.    // setting size, layout and visibility of frame
28.    f.setSize(400, 400);
29.    f.setLayout(null);
30.    f.setVisible(true);
31.  }
```

```
32.
33. // main method
34. public static void main(String args[])
35. {
36.    new ChoiceExample1();
37. }
38. }
```

Output:



h)List

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

AWT List class Declaration

1. **public class** List **extends** Component **implements** ItemSelectable, Accessible

AWT List Class Constructors

Sr. no.	Constructor	Description
1.	List()	It constructs a new scrolling list.
2.	List(int row_num)	It constructs a new scrolling list initialized with the given number of rows visible.
3.	List(int row_num, Boolean multipleMode)	It constructs a new scrolling list initialized which displays the given number of rows.

Methods Inherited by the List Class

The List class methods are inherited by following classes:

- java.awt.Component
- java.lang.Object

List Class Methods

Sr. no.	Method name	Description
1.	void add(String item)	It adds the specified item into the end of scrolling list.
2.	void add(String item, int index)	It adds the specified item into list at the given index position.
3.	void addActionListener(ActionListener l)	It adds the specified action listener to receive action events from list.
4.	void addItemListener(ItemListener l)	It adds specified item listener to receive item events from list.
6.	void deselect(int index)	It deselects the item at given index position.
9.	String getItem(int index)	It fetches the item related to given index position.
10.	int getItemCount()	It gets the count/number of items in the list.
12.	String[] getItems()	It fetched the items from the list.
13.	Dimension getMinimumSize()	It gets the minimum size of a scrolling list.
14.	Dimension getMinimumSize(int rows)	It gets the minimum size of a list with given number of rows.
15.	Dimension getPreferredSize()	It gets the preferred size of list.
16.	Dimension getPreferredSize(int rows)	It gets the preferred size of list with given number of rows.
17.	int getRows()	It fetches the count of visible rows in the list.
18.	int getSelectedIndex()	It fetches the index of selected item of list.
19.	int[] getSelectedIndexes()	It gets the selected indices of the list.
20.	String getSelectedItem()	It gets the selected item on the list.

21.	String[] getSelectedItems()	It gets the selected items on the list.
22.	Object[] getSelectedObjects()	It gets the selected items on scrolling list in array of objects.
23.	int getVisibleIndex()	It gets the index of an item which was made visible by method makeVisible()
24.	void makeVisible(int index)	It makes the item at given index visible.
25.	boolean isIndexSelected(int index)	It returns true if given item in the list is selected.
26.	boolean isMultipleMode()	It returns the true if list allows multiple selections.
31.	void removeActionListener(ActionListener l)	It removes specified action listener. Thus it doesn't receive further action events from the list.
32.	void removeItemListener(ItemListener l)	It removes specified item listener. Thus it doesn't receive further action events from the list.
33.	void remove(int position)	It removes the item at given index position from the list.
34.	void remove(String item)	It removes the first occurrence of an item from list.
35.	void removeAll()	It removes all the items from the list.
36.	void replaceItem(String newVal, int index)	It replaces the item at the given index in list with the new string specified.
37.	void select(int index)	It selects the item at given index in the list.
38.	void setMultipleMode(boolean b)	It sets the flag which determines whether the list will allow multiple selection or not.

Java AWT List Example

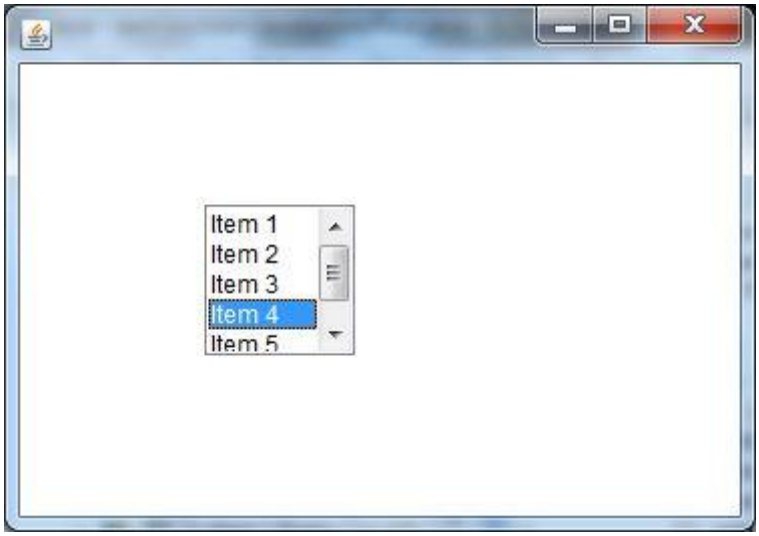
In the following example, we are creating a List component with 5 rows and adding it into the Frame.

ListExample1.java


```
1. // importing awt class
2. import java.awt.*;
3.
4. public class ListExample1
```

```
5.  {
6.      // class constructor
7.       ListExample1() {
8.       // creating the frame
9.          Frame f = new Frame();
10.       // creating the list of 5 rows
11.         List l1 = new List(5);
12.
13.         // setting the position of list component
14.         l1.setBounds(100, 100, 75, 75);
15.
16.         // adding list items into the list
17.         l1.add("Item 1");
18.         l1.add("Item 2");
19.         l1.add("Item 3");
20.         l1.add("Item 4");
21.         l1.add("Item 5");
22.
23.         // adding the list to frame
24.         f.add(l1);
25.
26.         // setting size, layout and visibility of frame
27.         f.setSize(400, 400);
28.         f.setLayout(null);
29.         f.setVisible(true);
30.     }
31.
32. // main method
33. public static void main(String args[])
34. {
35.    new ListExample1();
36. }
37. }
```

Output:



i)Panel

The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

It doesn't have title bar.

AWT Panel class declaration

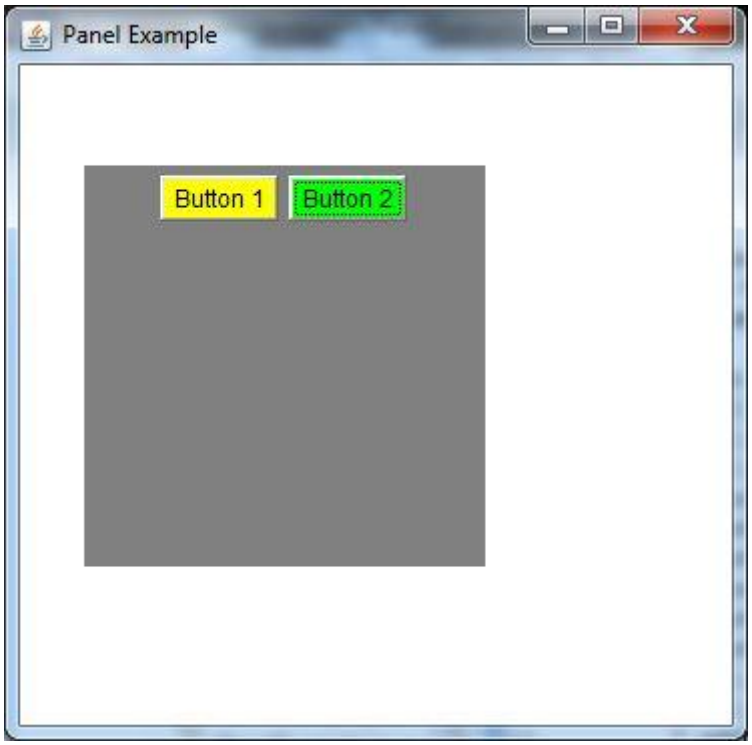
```
public class Panel extends Container implements Accessible
```

Java AWT Panel Example

```
import java.awt.*;
public class PanelExample {
    PanelExample()
      {
      Frame f= new Frame("Panel Example");
      Panel panel=new Panel();
      panel.setBounds(40,80,200,200);
      panel.setBackground(Color.gray);
      Button b1=new Button("Button 1");
      b1.setBounds(50,100,80,30);
      b1.setBackground(Color.yellow);
      Button b2=new Button("Button 2");
      b2.setBounds(100,100,80,30);
      b2.setBackground(Color.green);
      panel.add(b1); panel.add(b2);
      f.add(panel);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
```

```
20.     }
21.     public static void main(String args[])
22.     {
23.     new PanelExample();
24.     }
25. }
```

Output:



j)Scroll pane

When screen real estate is limited, use a scroll pane to display a component that is large or one whose size can change dynamically.

Fields

Modifier and Type	Field and Description
`static int`	**SCROLLBARS_ALWAYS** Specifies that horizontal/vertical scrollbars should always be shown regardless of the respective sizes of the scrollpane and child.
`static int`	**SCROLLBARS_AS_NEEDED** Specifies that horizontal/vertical scrollbar should be shown only when the size of the child exceeds the size of the

	scrollpane in the horizontal/vertical dimension.
static int	**SCROLLBARS_NEVER** Specifies that horizontal/vertical scrollbars should never be shown regardless of the respective sizes of the scrollpane and child.

Constructors

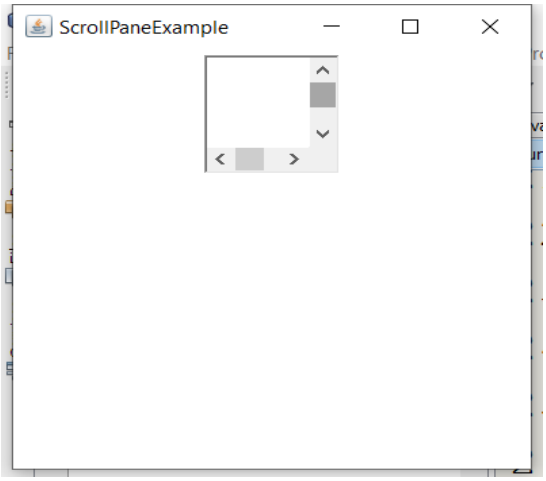
Constructor and Description
ScrollPane() Create a new scrollpane container with a scrollbar display policy of "as needed".
ScrollPane(int scrollbarDisplayPolicy) Create a new scrollpane container.

Example program:-

```
import java.awt.*;

public class ScrollPaneEx extends Frame{
   ScrollPaneEx()
   {
      setSize(400,400);

      setTitle("ScrollPaneExample");
      setVisible(true);

      this.setLayout(new FlowLayout());

      ScrollPane p=new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);

      add(p);
```

```
    }
    public static void main(String args[])
    {
        ScrollPaneEx e=new ScrollPaneEx();
    }
}
```

Output:-



k)Dialog

The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize buttons.

Frame vs Dialog

Frame and Dialog both inherits Window class. Frame has maximize and minimize buttons but Dialog doesn't have.

AWT Dialog class declaration

```
1. public class Dialog extends Window
```

Java AWT Dialog Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class DialogExample {
4.     private static Dialog d;
5.     DialogExample() {
6.         Frame f= new Frame();
```

```
7.          d = new Dialog(f , "Dialog Example", true);
8.          d.setLayout( new FlowLayout() );
9.          Button b = new Button ("OK");
10.         b.addActionListener ( new ActionListener()
11.         {
12.             public void actionPerformed( ActionEvent e )
13.             {
14.                 DialogExample.d.setVisible(false);
15.             }
16.         });
17.         d.add( new Label ("Click button to continue."));
18.         d.add(b);
19.         d.setSize(300,300);
20.         d.setVisible(true);
21.     }
22.     public static void main(String args[])
23.     {
24.         new DialogExample();
25.     }
26. }
```

Output:

L)Menu bar

- A menu bar can be created using **MenuBar** class.
- A menu bar may contain one or multiple menus, and these menus are created using **Menu** class.
- A menu may contain one of multiple menu items and these menu items are created using **MenuItem** class.

Signature: public class MenuBar extends MenuComponent implements MenuContainer, Accessible.

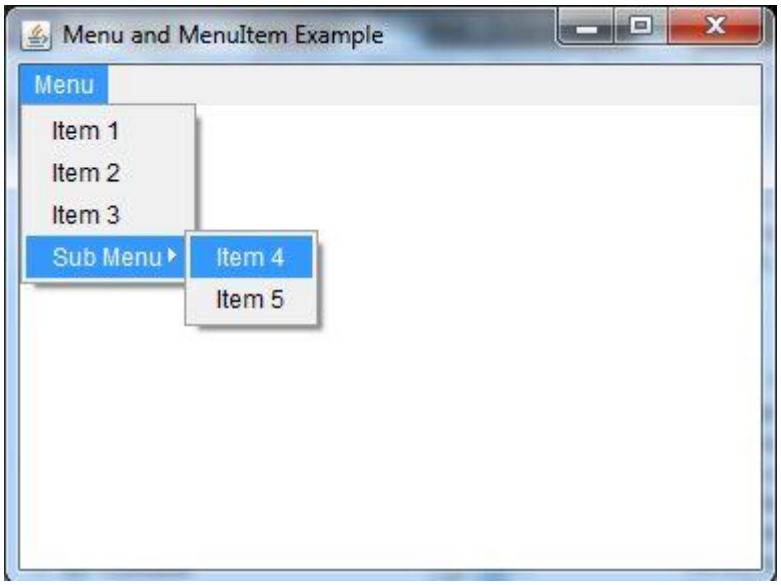
Simple constructors of MenuBar, Menu and MenuItem

Constructor	Description
public MenuBar()	Creates a menu bar to which one or many menus are added.
public Menu(String title)	Creates a menu with a title.
public MenuItem(String title)	Creates a menu item with a title.

Menu bar example program;-

```
1. import java.awt.*;
2. class MenuExample
3. {
4.     MenuExample(){
5.         Frame f= new Frame("Menu and MenuItem Example");
6.         MenuBar mb=new MenuBar();
7.         Menu menu=new Menu("Menu");
8.         Menu submenu=new Menu("Sub Menu");
9.         MenuItem i1=new MenuItem("Item 1");
10.        MenuItem i2=new MenuItem("Item 2");
11.        MenuItem i3=new MenuItem("Item 3");
12.        MenuItem i4=new MenuItem("Item 4");
13.        MenuItem i5=new MenuItem("Item 5");
14.        menu.add(i1);
15.        menu.add(i2);
16.        menu.add(i3);
17.        submenu.add(i4);
18.        submenu.add(i5);
19.        menu.add(submenu);
20.        mb.add(menu);
21.        f.setMenuBar(mb);
22.        f.setSize(400,400);
23.        f.setLayout(null);
24.        f.setVisible(true);
25. }
26. public static void main(String args[])
27. {
28. new MenuExample();
29. }
30. }
```

Output:


Menu and MenuItem Example
Menu
Item 1
Item 2
Item 3
Sub Menu
Item 4
Item 5

Layout Managers:-

1.Flow Layout

2.Border Layout

3.Grid Layout

4.Card Layout

5.Grid Bag Layout

Layout Managers:-

The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Types of LayoutManager

There are 6 layout managers in Java

- **FlowLayout**: It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**
- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.
- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to righ**t and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.
- **GridBagLayout**: It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect ration of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.

- **BoxLayout**: It arranges multiple components in either **vertically or horizontally**, but not both. The components are arranged from **left to right or top to bottom**. If the components are aligned **horizontally**, the height of all components will be the same and equal to the largest sized components. If the components are aligned **vertically**, the width of all components will be the same and equal to the largest width components.
- **CardLayout**: It arranges two or more components having the same size. The components are **arranged in a deck**, where all the cards of the same size and the **only top card are visible at any time**. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either **horizontally or vertically.**

1.FlowLayout:-

- The Java Flow Layout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.
- Flow Layout puts Components(Such as text fields, buttons, labels etc) in a row, if horizontal space is not enough to hold all components then flow layout adds them in a next row and so on.
- All rows in Flow Layout are **center** aligned by default.
- The default horizontal and vertical gap between components is 5 pixels.
- Flow Layout class is present in **java.awt** package.

Class declaration(Syntax):-

Following is the declaration for **java.awt.FlowLayout** class:

```
Public class FlowLayout extends Object implements LayoutManager, Serializable
{
------

------
}
```

Field or Variable

Following are the fields for **java.awt.BorderLayout** class:

- **static int CENTER** -- This value indicates that each row of components should be centered.
- **static int LEADING** -- This value indicates that each row of components should be justified to the leading edge of the container's orientation, for example, to the left in left-to-right orientations.
- **static int LEFT** -- This value indicates that each row of components should be left-justified.
- **static int RIGHT** -- This value indicates that each row of components should be right-justified.
- **static int TRAILING** -- This value indicates that each row of components should be justified to the trailing edge of the container's orientation, for example, to the right in left-to-right orientations.

Class constructors

S.N.	Constructor & Description
1	**FlowLayout()** Constructs a new FlowLayout with a centered alignment and a default 5-unit horizontal and vertical gap.
2	**FlowLayout(int align)** Constructs a new FlowLayout with the specified alignment and a default 5-unit horizontal and vertical gap.
3	**FlowLayout(int align, int hgap, int vgap)**

	Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

Class methods

S.N.	Method & Description
1	**int getAlignment()** Gets the alignment for this layout.
2	**int getHgap()** Gets the horizontal gap between components.
3	**int getVgap()** Gets the vertical gap between components.
4	**void setAlignment(int align)** Sets the alignment for this layout.
5	**void setHgap(int hgap)** Sets the horizontal gap between components.
6	**void setVgap(int vgap)** Sets the vertical gap between components.

Methods inherited

This class inherits methods from the following classes:

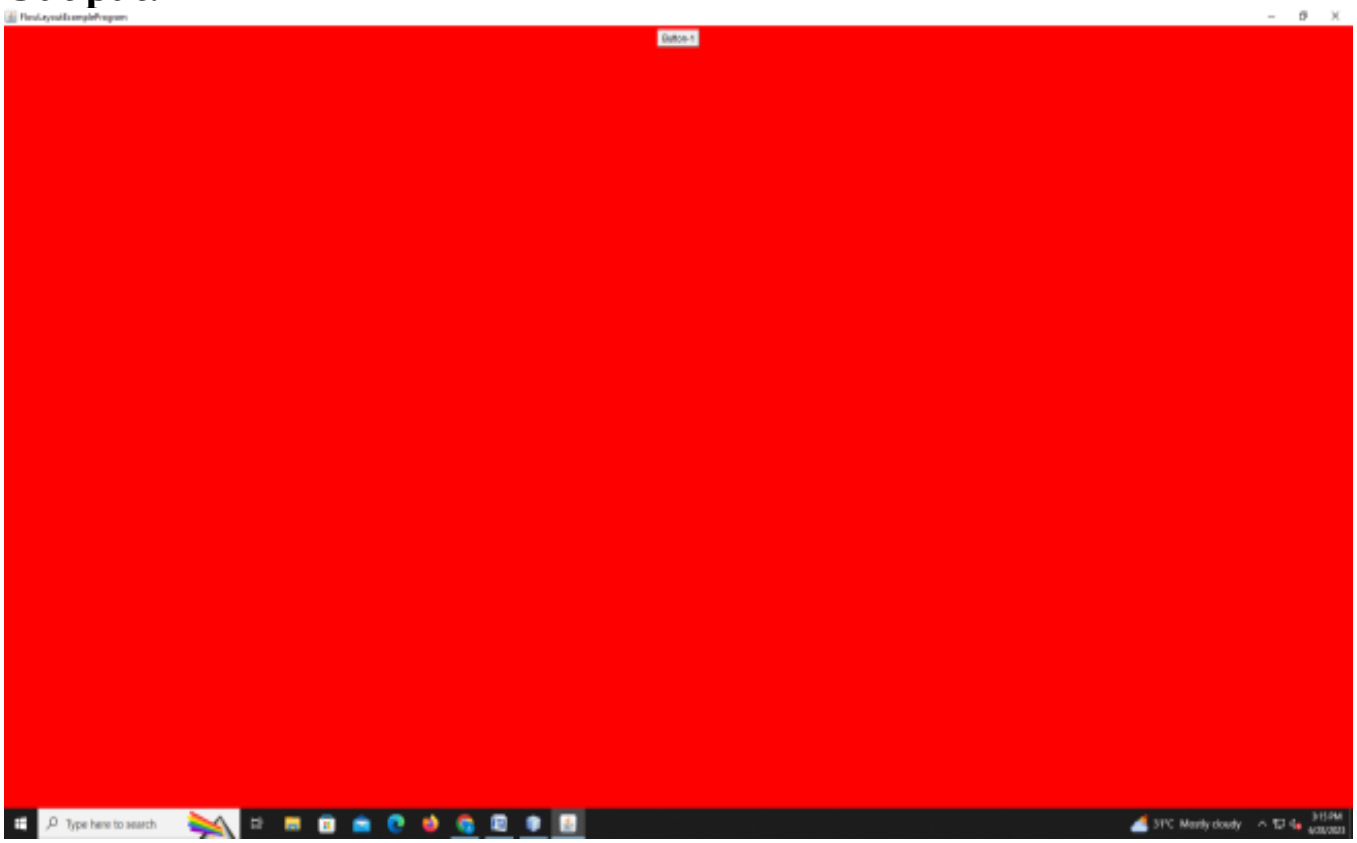
- java.lang.Object

Example program:-

```
import java.awt.*;
import java.awt.event.*;
public class FlowLayoutExample extends Frame{
   public FlowLayoutExample()
   {


      this.setSize(400,400);

        this.setVisible(true);
      this.setTitle("FlowLayoutExampleProgram");
      this.setBackground(Color.red);
      this.addWindowListener(new WindowAdapter()
```

```
        {
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
    });
       Button b1=new Button("Button-1");

       this.add(b1);

       FlowLayout f=new FlowLayout();
       this.setLayout(f);
    }
    public static void main(String args[])
    {
       FlowLayoutExample s=new FlowLayoutExample();
    }

}
```

Out put:-



2.BorderLayout:-

- The class **BorderLayout** arranges the components to fit in the five regions: east, west, north, south and center.
- Each region (area) may contain one component only.
- It is the default layout of a frame or window.
- It is present in java.awt package.

Class declaration(Syntax):-

Following is the declaration for **java.awt.BorderLayout** class:

Public class BorderLayout extends object implements LayoutManager2, Serializable

{

}

Field or Variable:-

Following are the fields for **java.awt.BorderLayout** class:

- **static String NORTH** -- The north layout constraint (top of container).
- **static String SOUTH** -- The south layout constraint (bottom of container).
- **static String WEST** -- The west layout constraint (left side of container).
- **static String EAST** -- The east layout constraint (right side of container).
- **static String CENTER** -- The center layout constraint (middle of container).

Class constructors

S.N.	Constructor & Description
1	**BorderLayout()** Constructs a new border layout with no gaps between components.
2	**BorderLayout(int hgap, int vgap)** Constructs a border layout with the specified gaps between components.

Class methods

S.N.	Method & Description
1	**void add(Component compObj, Object region)** Here, compObj is the component to be added, and region specifies where the component will be added.
2	**int getHgap()** Gets the horizontal gap between components.
3	**int getVgap()** Gets the vertical gap between components.

4	**void setHgap(int hgap)** Sets the horizontal gap between components.
5	**void setVgap(int vgap)** Sets the vertical gap between components.

Methods inherited

This class inherits methods from the following classes:

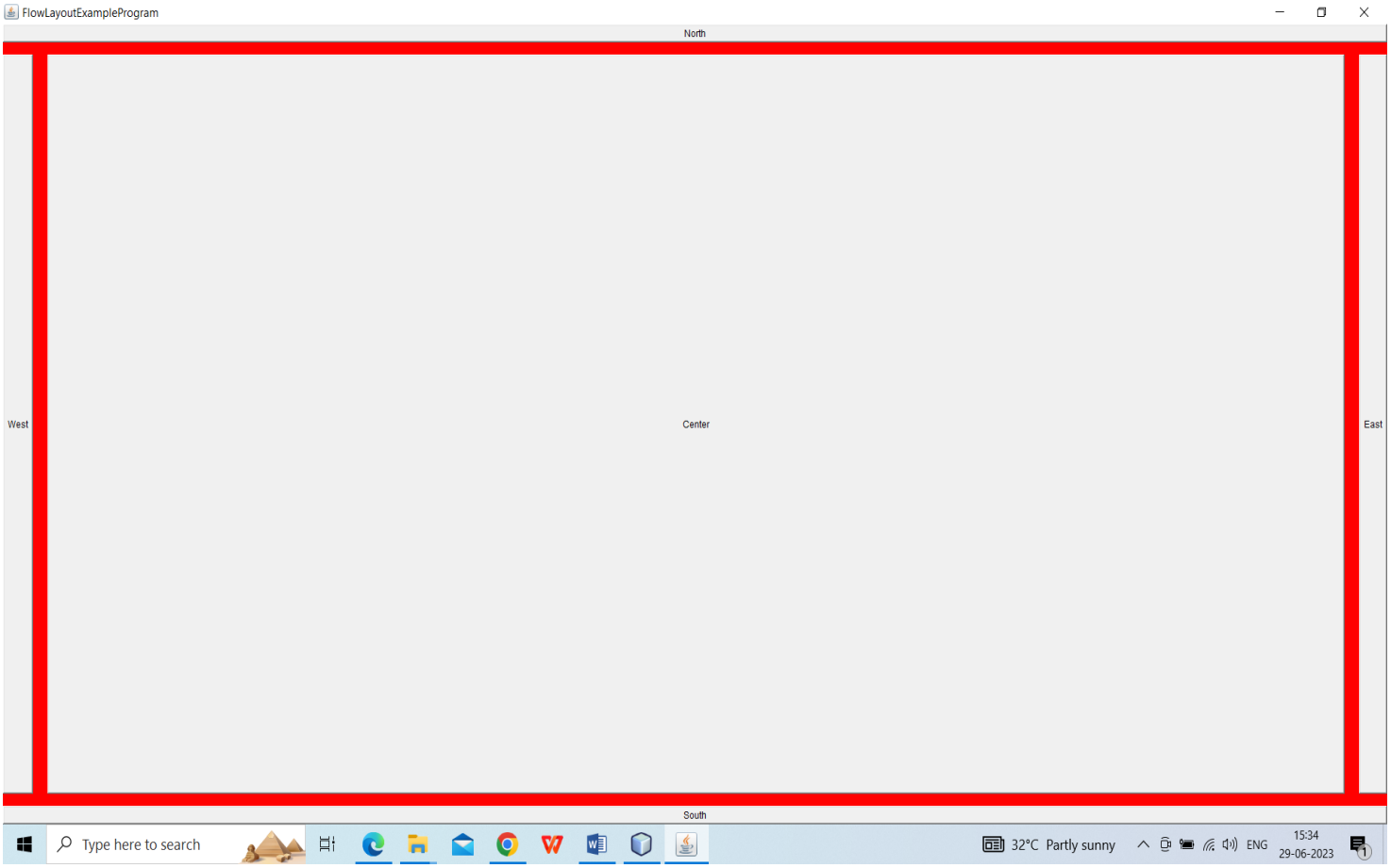
- java.lang.Object

Example Program:-

```
import java.awt.*;
import java.awt.event.*;
public class BorderLayoutExample extends Frame{
   public BorderLayoutExample()
   {


      this.setSize(400,400);
       this.setVisible(true);
      this.setTitle("FlowLayoutExampleProgram");
      this.setBackground(Color.red);
      this.addWindowListener(new WindowAdapter()
      {
       public void windowClosing(WindowEvent e)
       {
          System.exit(0);
       }
   });
       BorderLayout b=new BorderLayout(20,15);
       this.setLayout(b);
      Button b1=new Button("East");
      Button b2=new Button("West");
      Button b3=new Button("North");
      Button b4=new Button("South");
      Button b5=new Button("Center");

      this.add(b1,"East");
      this.add(b2,"West");
      this.add(b3,"North");
      this.add(b4,"South");
      this.add(b5,"Center");

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

```
    {
BorderLayoutExample s=new BorderLayoutExample();
    }

}
```

OutPut:-



3.Grid Layout

The Java GridLayout class is used to arrange the components in a rectangular grid.
One component is displayed in each rectangle. GridLayout class is present in java.awt package. GridLayout is useful to divide the container into a 2D grid form that contains several rows and columns. The container is divided into equal-sized rectangle; and one component is placed in each rectangle.

Class declaration(Syntax):-

Following is the declaration for **java.awt.GridLayout** class:

```
Public class GridLayout extends Object implements LayoutManager, Serializable
{
------

------
}
```

Class Constructors

Sr.No.	Constructor & Description
1	**GridLayout()** Creates a grid layout with a default of one column per component, in a single row.
2	**GridLayout(int rows, int cols)** Creates a grid layout with the specified number of rows and columns but no gaps between the components.
3	**GridLayout(int rows, int cols, int hgap, int vgap)** Creates a grid layout with the specified number of rows and columns along with given horizontal and vertical gaps.

Class Methods

Sr.No.	Method & Description
1	**int getColumns()** Gets the number of columns in this layout.
2	**int getHgap()** Gets the horizontal gap between the components.
3	**int getRows()** Gets the number of rows in this layout.
4	**int getVgap()** Gets the vertical gap between the components.
5	**void setColumns(int cols)** Sets the number of columns in this layout to the specified value.
6	**void setHgap(int hgap)** Sets the horizontal gap between the components to the specified value.

7	**void setRows(int rows)** Sets the number of rows in this layout to the specified value.
8	**void setVgap(int vgap)** Sets the vertical gap between the components to the specified value.

Methods Inherited

This class inherits methods from the following class −

- java.lang.Object

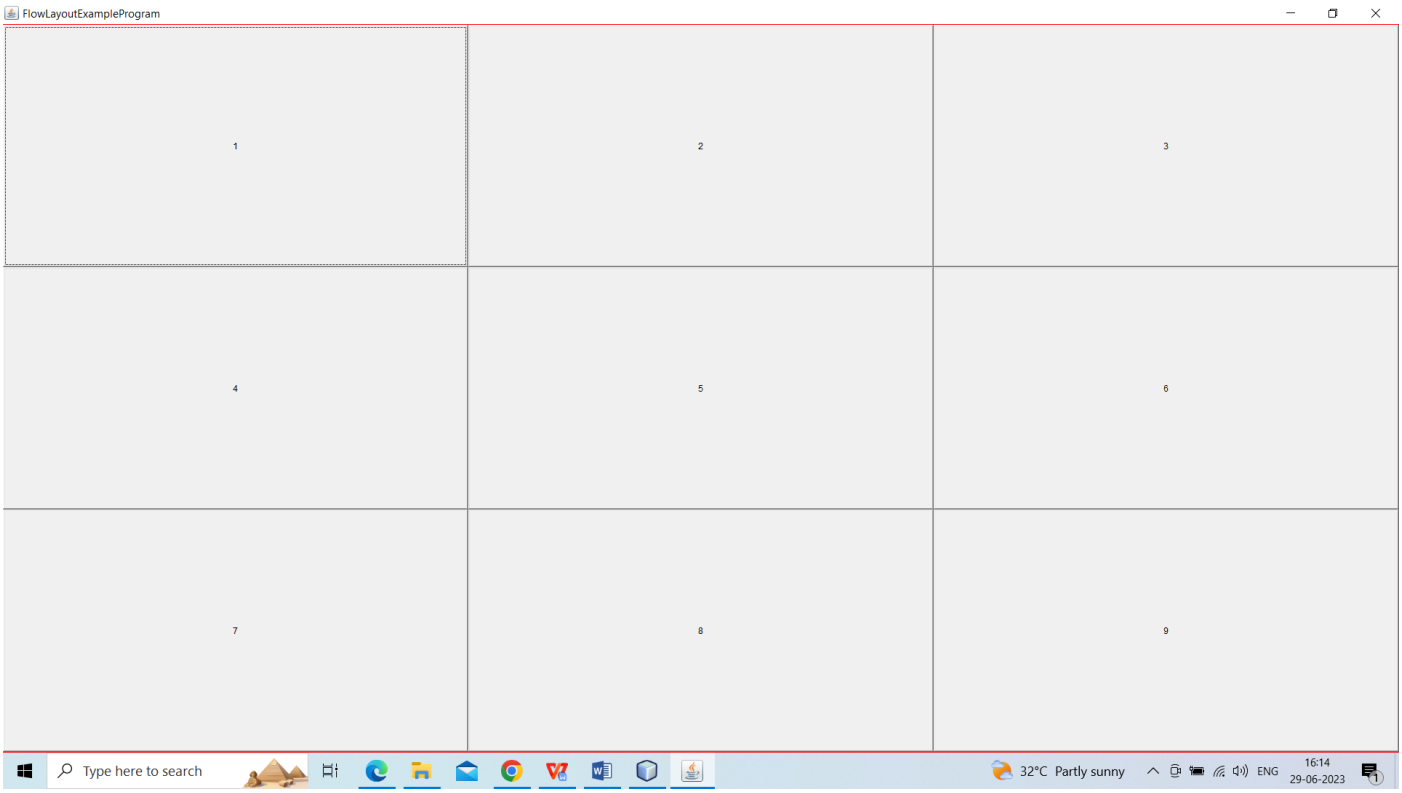
Example Program:-

```
import java.awt.*;
import java.awt.event.*;
public class GridLayoutExample extends Frame{
   public GridLayoutExample()
   {



      this.setSize(500,500);


       this.setVisible(true);
      this.setTitle("FlowLayoutExampleProgram");
      this.setBackground(Color.red);
      this.addWindowListener(new WindowAdapter()
      {
       public void windowClosing(WindowEvent e)
       {
          System.exit(0);
```

```
    }
});


    GridLayout g=new GridLayout(3,3);

     this.setLayout(g);

    Button b1=new Button("1");
    Button b2=new Button("2");
    Button b3=new Button("3");
    Button b4=new Button("4");
    Button b5=new Button("5");
    Button b6=new Button("6");
    Button b7=new Button("7");
    Button b8=new Button("8");
    Button b9=new Button("9");
    this.add(b1);
    this.add(b2);
    this.add(b3);
    this.add(b4);
    this.add(b5);
    this.add(b6);
    this.add(b7);
    this.add(b8);
    this.add(b9);
```

```
        //b.setHgap(50);
        //System.out.println(b.getHgap());
        //f.setAlignment(FlowLayout.RIGHT);

    }
    public static void main(String args[])
    {
GridLayoutExample s=new GridLayoutExample();
    }

}
```

OutPut:-



4.Card Layout

- The CardLayout class manages the components in such a way that only one component is visible at a time.
- It treats each component as a card in the container.
- Only one card is visible at a time, and the container acts as a stack of cards.
- The first component added to a CardLayout object is the visible component, when the container is first displayed.
- CardLayout class is found in java.awt package.

Class declaration(Syntax):-

Following is the declaration for **java.awt.CardLayout** class:

Public class CardLayout extends Object implements LayoutManager2, Serializable

{

}

Class Constructors

Sr.No.	Constructor & Description
1	**CardLayout()** Creates a new card layout with the gaps of size zero.
2	**CardLayout(int hgap, int vgap)** Creates a new card layout with the specified horizontal and vertical gaps.

Class Methods

Sr.No.	Method & Description
1	**void first(Container parent)** Flips to the first card of the container.
2	**int getHgap()** Gets the horizontal gap between the components.

3	**int getVgap()** Gets the vertical gap between the components.
4	**void last(Container parent)** Flips to the last card of the container.
5	**void next(Container parent)** Flips to the next card of the specified container.
6	**void previous(Container parent)** Flips to the previous card of the specified container.
7	**void setHgap(int hgap)** Sets the horizontal gap between the components.
8	**void setVgap(int vgap)** Sets the vertical gap between the components.
9	**void show(Container parent, String name)** Flips to the component that was added to this layout with the specified name, using addLayoutComponent.

Methods Inherited

This class inherits methods from the following class −

- java.lang.Object

Example:-

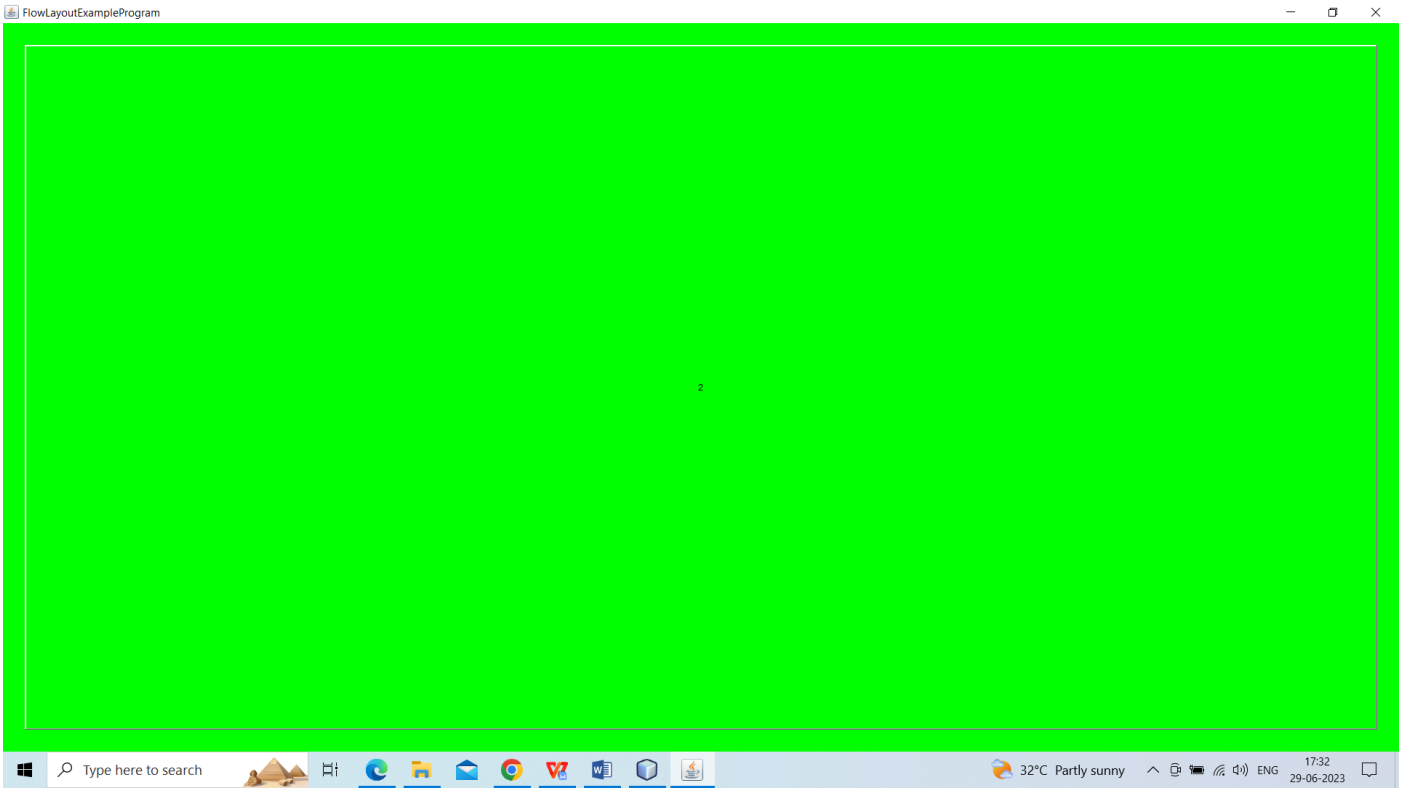
```
import java.awt.*;
import java.awt.event.*;
public class CardLayoutExample extends Frame implements ActionListener{
   CardLayout c;
   public CardLayoutExample()
   {


      this.setSize(500,500);

        this.setVisible(true);
```

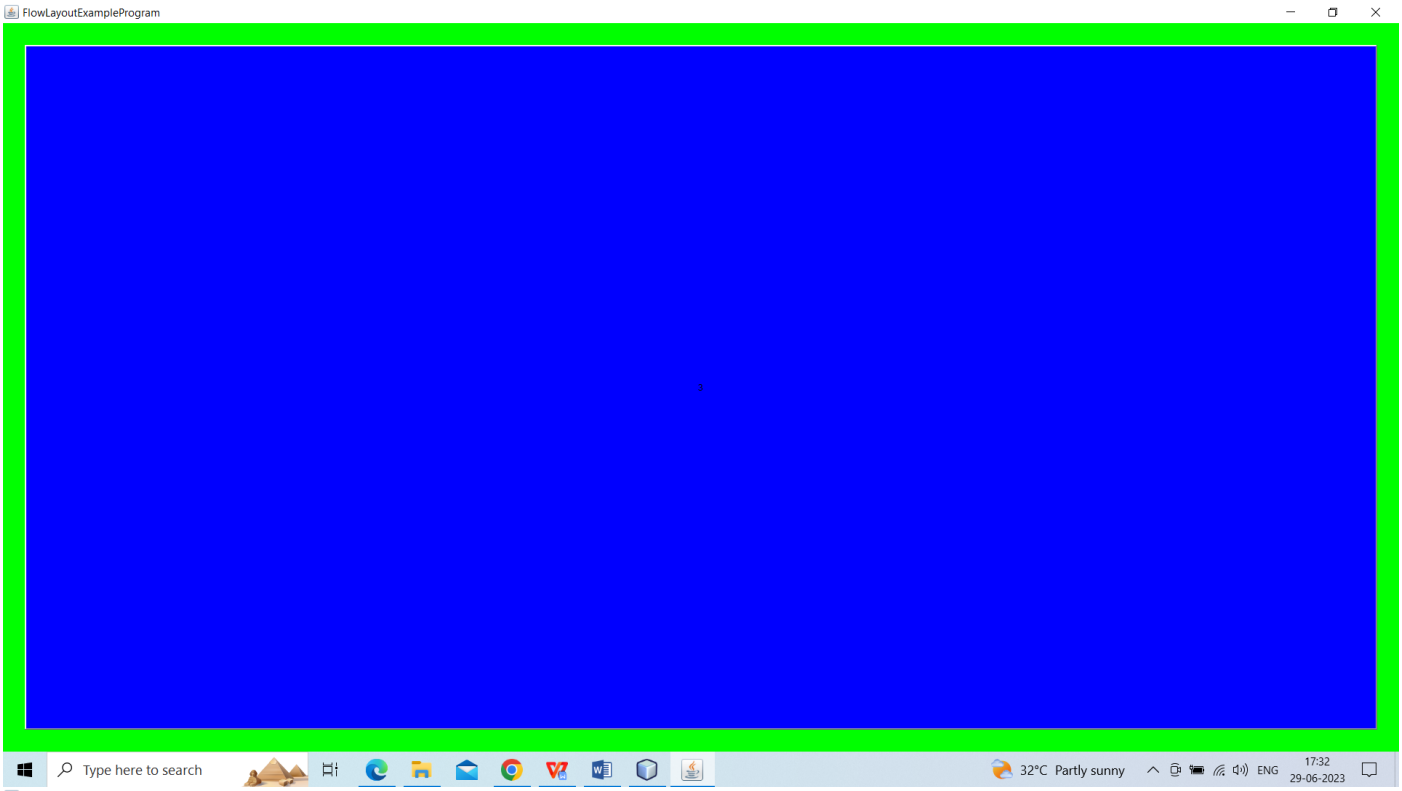
```
        this.setTitle("FlowLayoutExampleProgram");
        this.setBackground(Color.green);
        this.addWindowListener(new WindowAdapter()
        {
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
    });


        c=new CardLayout(30,30);

         this.setLayout(c);

        Button b1=new Button("1");
        b1.setBackground(Color.red);
        Button b2=new Button("2");
        b2.setBackground(Color.green);
        Button b3=new Button("3");
        b3.setBackground(Color.blue);
        Button b4=new Button("4");
        b4.setBackground(Color.yellow);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);

        this.add(b1);
        this.add(b2);
        this.add(b3);
        this.add(b4);

    }
    public void actionPerformed(ActionEvent e)
    {
        c.next(this);
    }
    public static void main(String args[])
    {
CardLayoutExample s=new CardLayoutExample();
    }
```

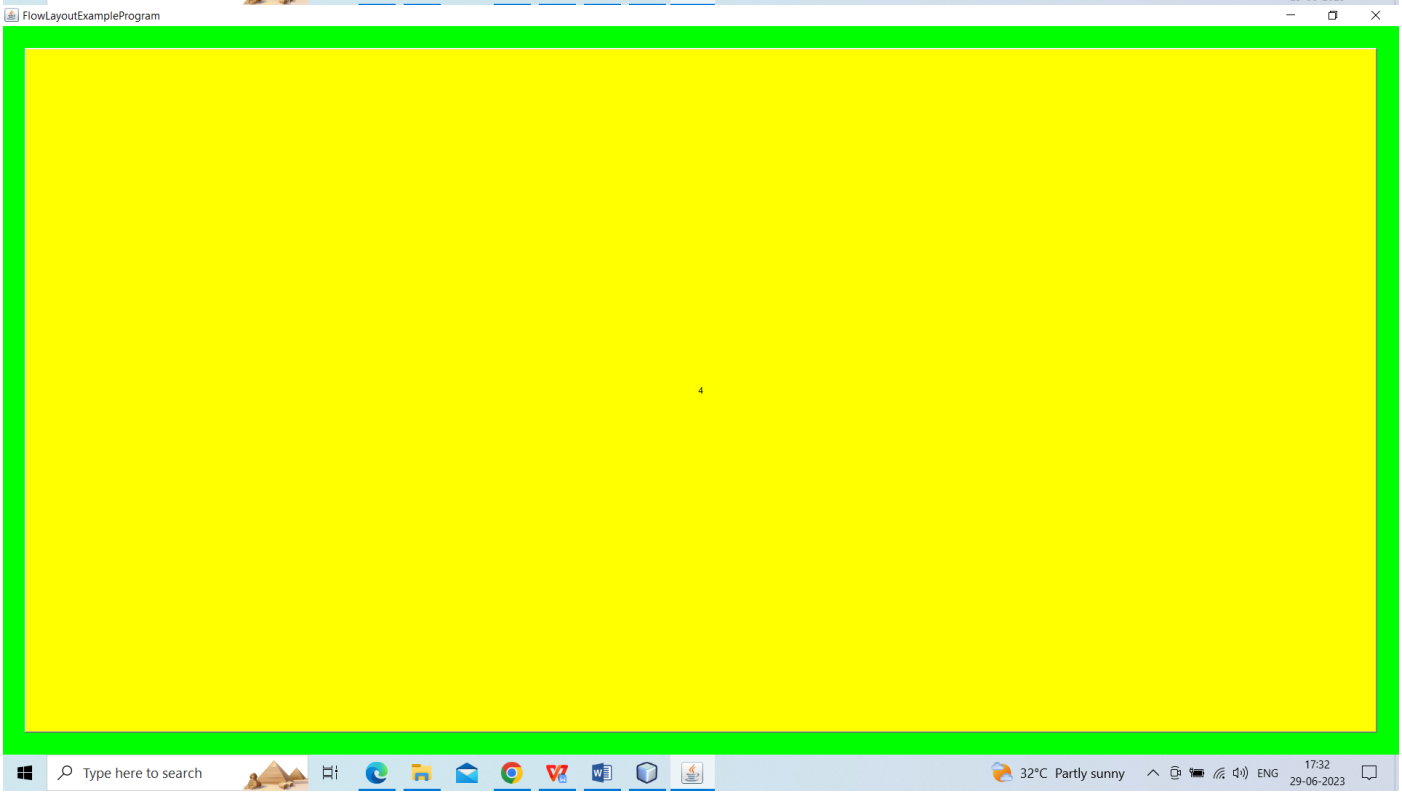
}

OutPut:-

When you click button 1 it will display next button 2 and click again display 3, 4.




FlowLayoutExampleProgram
3
Type here to search
32°C Partly sunny
ENG
17:32
29-06-2023


FlowLayoutExampleProgram
4
Type here to search
32°C Partly sunny
ENG
17:32
29-06-2023

5.Grid Bag Layout

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline. A GridBagLayout class represents grid bag layout manager where the components are arranged in rows and columns. In this layout the component can span more than one row or column and the size of the component can be adjusted to fit in the display area.

When positioning the components by using grid bag layout, it is necessary to apply some constraints or conditions on the components regarding their position, size and place in or around the components etc. Such constraints are specified using GridBagConstraints class.

In order to create GridBagLayout, we first instantiate the GridBagLayout class by using its only no-argument constructor

GridBagLayout layout=new GridBagLayout();

setLayout(layout);

and defining it as the current layout manager.

To apply constraints on the components, we should first create an object to GridBagConstrinats class, as

GridBagConstrinats gbc =new GridBagConstrinats();

This will create constraints for the components with default value. The other way to specify the constraints is by directly passing their values while creating the

GridBagConstrinats as

GridBagConstrinats gbc= new GridBagConstrinats(

int gridx, int gridy, int gridwidth, int gridheight, double weightx,

double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady);

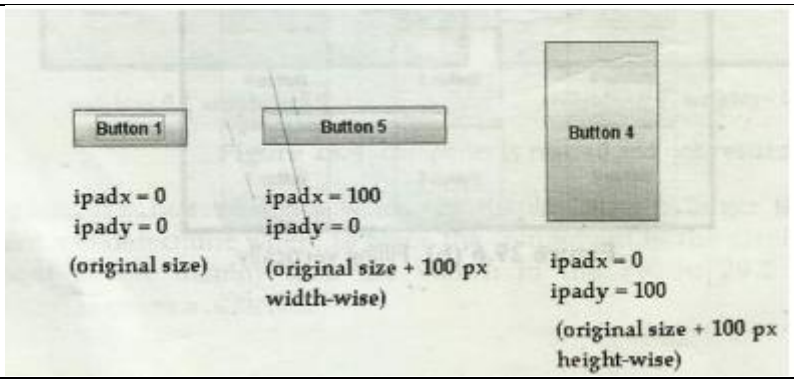
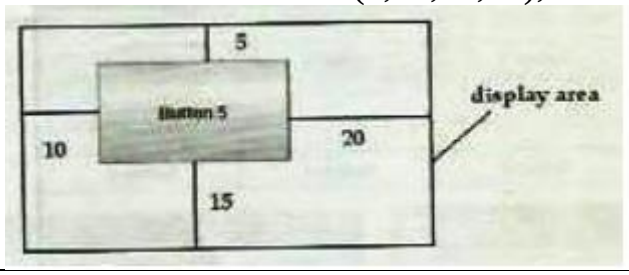
To set the constraints use setConstraints() method in GridBagConstrinats class and its

prototype

void setConstraints(Component comp, GridBagConstraints cons);

Constraint fields Defined by GridBagConstraints:

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER. Others are □ GridBagConstraints.EAST □ GridBagConstraints.WEST □ GridBagConstraints.SOUTH □ GridBagConstraints.NORTH □ GridBagConstraints.NORTHEAST □ GridBagConstraints.NORTHWEST □ GridBagConstraints.SOUTHEAST □ GridBagConstraints.SOUTHWEST
int gridx	Specifies the X coordinate of the cell to which the component will be added.
int gridy	Specifies the Y coordinate of the cell to which the component will be added.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated.
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.

	
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are □ GridBagConstraints.NONE (the default) □ GridBagConstraints.HORIZONTAL □ GridBagConstraints.VERTICAL □ GridBagConstraints.BOTH.
Insets insets	Small amount of space between the container that holds your components and the window that contains it. Default insets are all zero. Insets(int top,int left,int bottom, int right) Ex. Insets i=new Insets(5,10,15,20); 

Example:

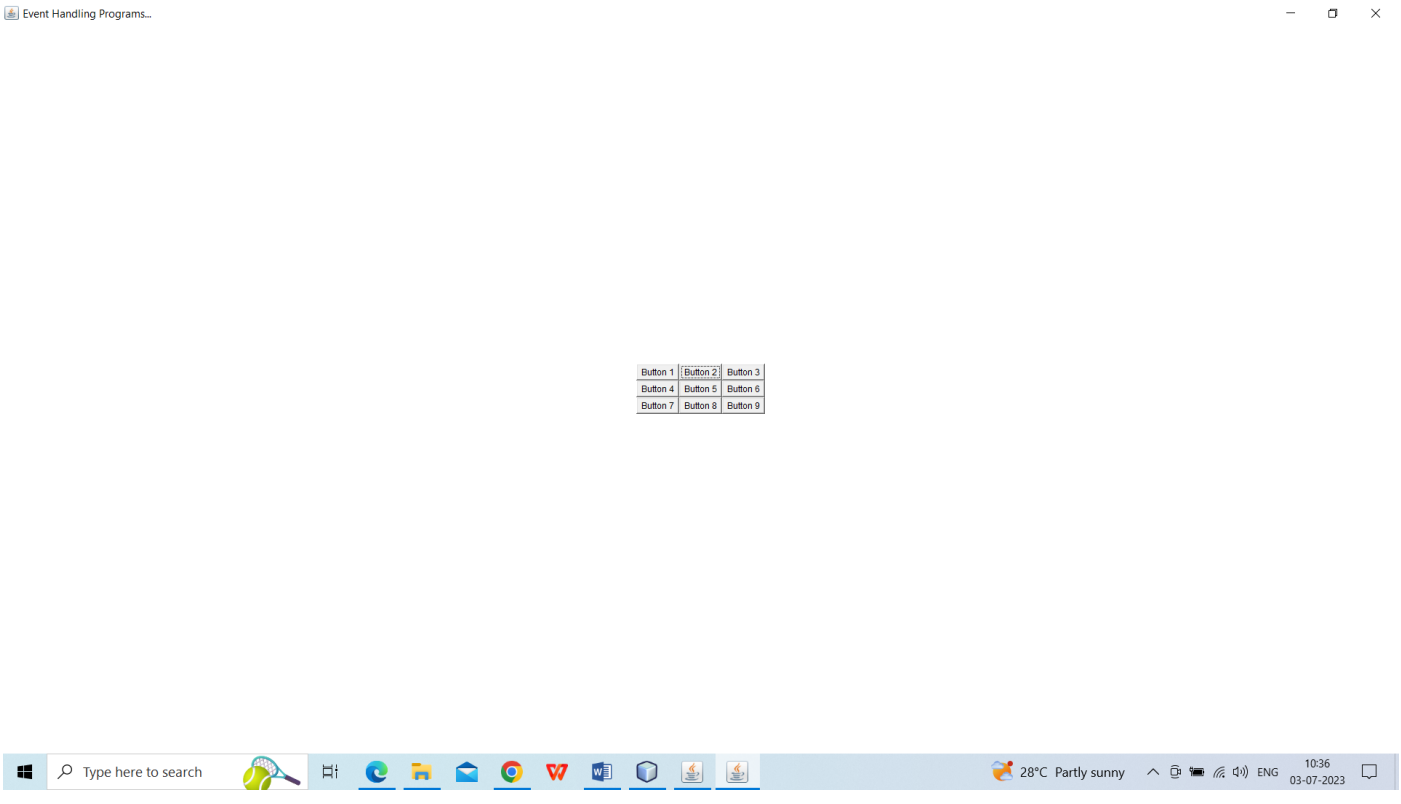
```
// Use GridBagLayout.

import java.awt.*;

import java.awt.event.*;

class MyFrame extends Frame

{

Button b1,b2,b3,b4,b5,b6,b7,b8,b9;
```

```
MyFrame()
{
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
b1=new Button("Button 1");
b2=new Button("Button 2");
b3=new Button("Button 3");
b4=new Button("Button 4");
b5=new Button("Button 5");
b6=new Button("Button 6");
b7=new Button("Button 7");
b8=new Button("Button 8");
b9=new Button("Button 9");
gbc.gridx=0;
gbc.gridy=0;
gbag.setConstraints(b1,gbc);
gbc.gridx=1;
gbc.gridy=0;
gbag.setConstraints(b2,gbc);
gbc.gridx=2;
gbc.gridy=0;
gbag.setConstraints(b3,gbc);
gbc.gridx=0;
gbc.gridy=1;
gbag.setConstraints(b4,gbc);
gbc.gridx=1;
```

```
gbc.gridy=1;
gbag.setConstraints(b5,gbc);
gbc.gridx=2;
gbc.gridy=1;
gbag.setConstraints(b6,gbc);
gbc.gridx=0;
gbc.gridy=2;
gbag.setConstraints(b7,gbc);
gbc.gridx=1;
gbc.gridy=2;
gbag.setConstraints(b8,gbc);
gbc.gridx=2;
gbc.gridy=2;
gbag.setConstraints(b9,gbc);
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
add(b6);
add(b7);
add(b8);
add(b9);
}
}
class GridBagDemo2
{
```

```
public static void main(String arg[])
{
MyFrame f=new MyFrame();
f.setSize(400,400);
f.setTitle("GridBagLayout Example...");
f.setVisible(true);
f.addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
});
}
}
```

OutPut:-



Unit-5

Applets:-

1. **Concepts of Applets**
2. **Difference between applets and applications**
3. **Life cycle of an applet**
4. **Types of Applets**
5. **Creating applets**
6. **Passing parameters to applets**

1.Concepts of Applets

Defination:-

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

Important points :

1. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
2. Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.
3. An applet is a Java class that extends the java.applet.Applet class.
4. A main() method is not invoked on an applet, and an applet class will not define main().
5. Applets are designed to be embedded within an HTML page.
6. When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
7. A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
8. The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
9. Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
10. Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Why we need Applet?

1. When we need to include something dynamic to include in our web page.
2. When we require some flash output like applet to produce some sound, animations or some special effects when displaying some certain pages.
3. When we want to create a program and it to make available on internet so that it could be used by others.

Advantage of Applet

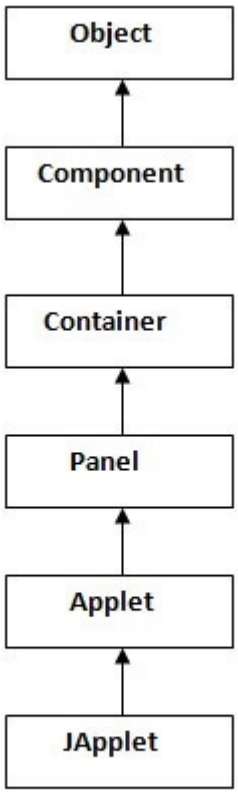
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

2.Difference between applets and applications

Parameters	Java Application	Java Applet
Definition	Applications are just like a Java program that can be executed independently without using the web browser.	Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution.
main () method	The application program requires a main() method for its execution.	The applet does not require the main() method for its execution instead init() method is required.
Compilation	The “javac” command is used to compile application programs, which are then executed using the “java” command.	Applet programs are compiled with the “javac” command and run using either the “appletviewer” command or the web browser.
File access	Java application programs have full access to the local file system and network.	Applets don’t have local disk and network access.
Access level	Applications can access all kinds of resources available on the system.	Applets can only access browser-specific services. They don’t have access to the local system.
Installation	First and foremost, the installation of a Java application on the local computer is required.	The Java applet does not need to be installed beforehand.
Execution	Applications can execute the programs from the local system.	Applets cannot execute programs from the local machine.
Program	An application program is needed to perform some tasks directly for the user.	An applet program is needed to perform small tasks or part of them.
Run	It cannot run on its own; it needs JRE to execute.	It cannot start on its own, but it can be executed using a Java-enabled web browser.

Parameters	Java Application	Java Applet
Connection with servers	Connectivity with other servers is possible.	It is unable to connect to other servers.
Read and Write Operation	It supports the reading and writing of files on the local computer.	It does not support the reading and writing of files on the local computer.
Security	Application can access the system's data and resources without any security limitations.	Executed in a more restricted environment with tighter security. They can only use services that are exclusive to their browser.
Restrictions	Java applications are self-contained and require no additional security because they are trusted.	Applet programs cannot run on their own, necessitating the maximum level of security.

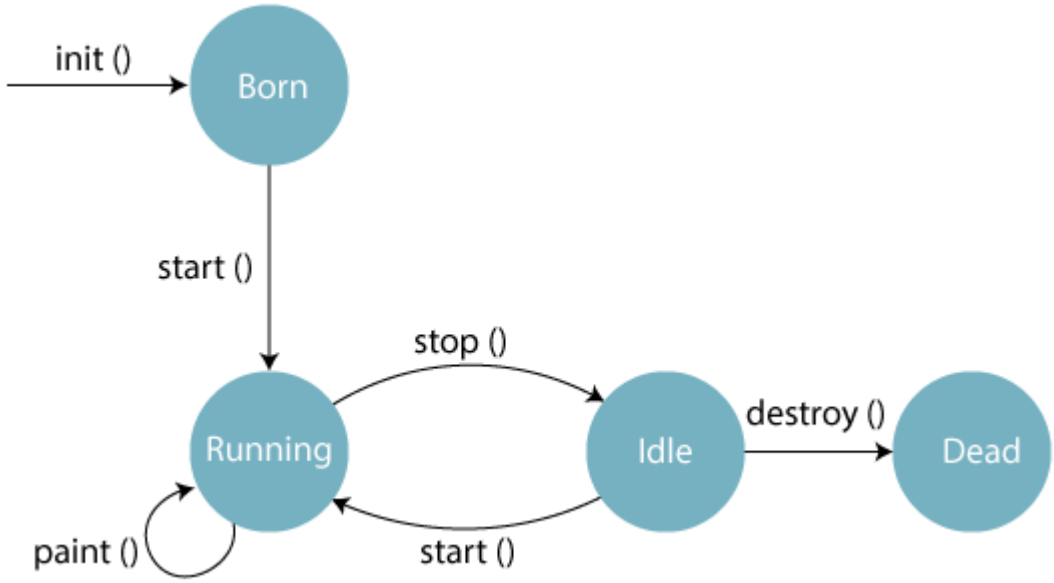
3.Life cycle of an applet

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

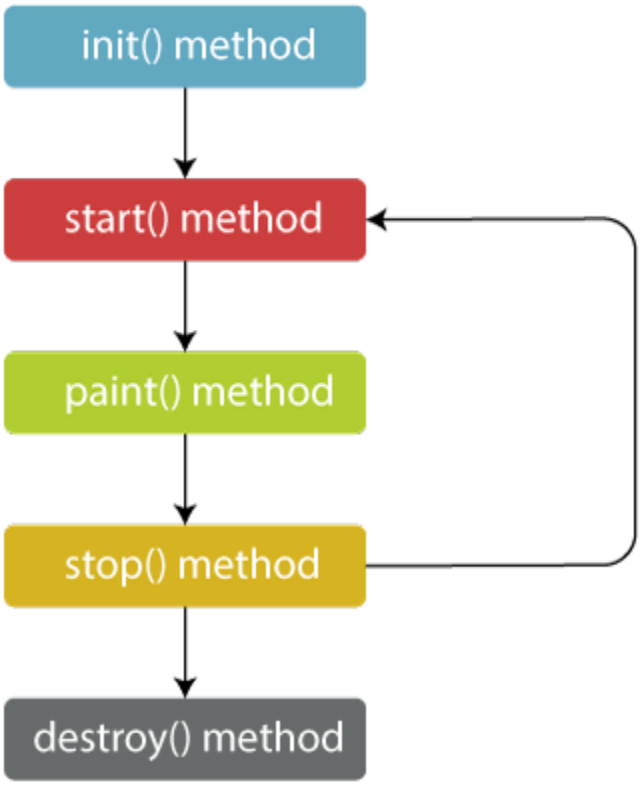


There are five methods of an applet life cycle, and they are:

- **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.

Flow of Applet Life Cycle:

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



Syntax of entire Applet Life Cycle in Java

```
1. class TestAppletLifeCycle extends Applet {
2. public void init() {
3. // initialized objects
4. }
5. public void start() {
6. // code to start the applet
7. }
8. public void paint(Graphics graphics) {
9. // draw the shapes
10.}
11.public void stop() {
12.// code to stop the applet
13.}
14.public void destroy() {
15.// code to destroy the applet
16.}
17.}
```

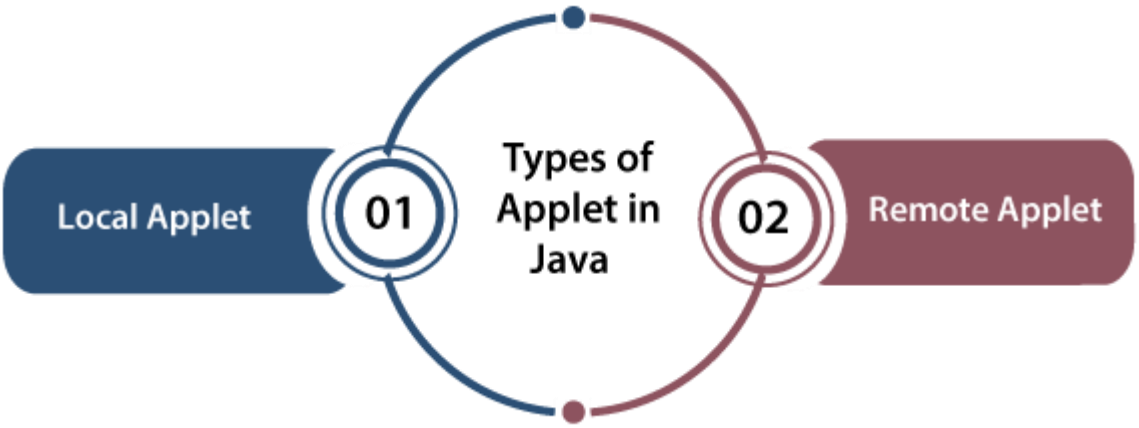
Who is responsible to manage the life cycle of an applet?

Java Plug-in software.

4.Types of Applets

A special type of Java program that runs in a Web browser is referred to as **Applet**. It has less response time because it works on the client-side. It is much secured executed by the browser under any of the platforms such as Windows, Linux and Mac OS etc. There are two types of applets that a web page can contain.

1. **Local Applet**
2. **Remote Applet**



Let's understand both types of Applet one by one:

Local Applet

Local Applet is written on our own, and then we will embed it into web pages. Local Applet is developed locally and stored in the local system. A web page doesn't need the get the information from the internet when it finds the local Applet in the system. It is specified or defined by the file name or pathname. There are two attributes used in defining an applet, i.e., the **codebase** that specifies the path name and **code** that defined the name of the file that contains Applet's code.

Specifying Local applet

```
<applet
    codebase = "tictactoe"
    code = "FaceApplet.class"
    width = 120
    height = 120>
</applet>
```

1. First, we will create a Local Applet for embedding in a web page.
2. After that, we will add that Local Applet to the web page.

FaceApplet.java

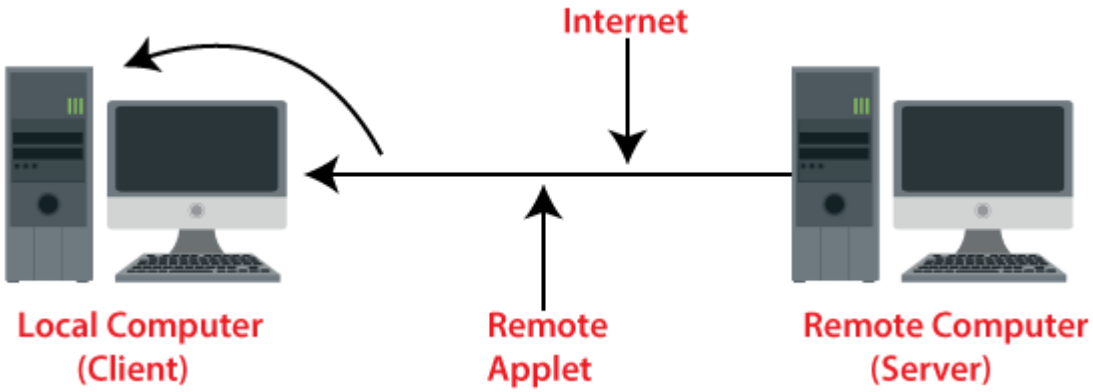
```
//Import packages and classes
```

```
2. import java.applet.*;
3. import java.awt.*;
4. //Creating FaceApplet class that extends Applet
5. public class FaceApplet extends Applet
6. {
7.     //paint() method starts
8.     public void paint(Graphics g){
9.         //Creating graphical object
10.        g.setColor(Color.red);
11.        g.drawString("Welcome", 50, 50);
12.        g.drawLine(20, 30, 20, 300);
13.        g.drawRect(70, 100, 30, 30);
14.        g.fillRect(170, 100, 30, 30);
15.        g.drawOval(70, 200, 30, 30);
16.        g.setColor(Color.pink);
17.        g.fillOval(170, 200, 30, 30);
18.        g.drawArc(90, 150, 30, 30, 30, 270);
19.        g.fillArc(270, 150, 30, 30, 0, 180);
20.    }
21.}
```

Execute the above code by using the following commands:

Remote Applet

A remote applet is designed and developed by another developer. It is located or available on a remote computer that is connected to the internet. In order to run the applet stored in the remote computer, our system is connected to the internet then we can download run it. In order to locate and load a remote applet, we must know the applet's address on the web that is referred to as Uniform Recourse Locator(URL).



Specifying Remote applet

```
<applet
    codebase = "http://www.myconnect.com/applets/"
    code = "FaceApplet.class"
    width = 120
    height =120>
</applet>
```

Difference Between Local Applet and Remote Applet

The following table describes the key differences between Local applet and Remote applet.

Local Applet	Remote Applet
There is no need to define the Applet's URL in Local Applet.	We need to define the Applet's URL in Remote Applet.
Local Applet is available on our computer.	Remote Applet is not available on our computer.
In order to use it or access it, we don't need Internet Connection.	In order to use it or access it on our computer, we need an Internet Connection.
It is written on our own and then embedded into the web pages.	It was written by another developer.
We don't need to download it.	It is available on a remote computer, so we need to download it to our system.

5.Creating applets

Creating Hello Geeks applet :

There are two ways to run an applet

1. **By using Web Browser**
2. **By using Applet Viewer**

1.Executing the applet within a Java-compatible Web browser:
Suppose we have a GfgApplet.java file in which we have our java code.

```
// Java program to run the applet

// using the web browser

import java.awt.*;

import java.applet.*;

public class GfgApplet extends Applet

{

    public void paint(Graphics g)

    {

        g.drawString("Hello Geeks,Welcome to GeeksforGeeks",20,20);

    }

}
```

- Compiling: javac GfgApplet.java
- Create an Html file and embed the Applet tag in the HTML file.

To run an applet in a web browser, we must create an HTML text file with a tag that loads the applet. For this, we may use the APPLET or OBJECT tags. Here is the HTML file that runs HelloWorld with APPLET:

Attributes in applet tag:

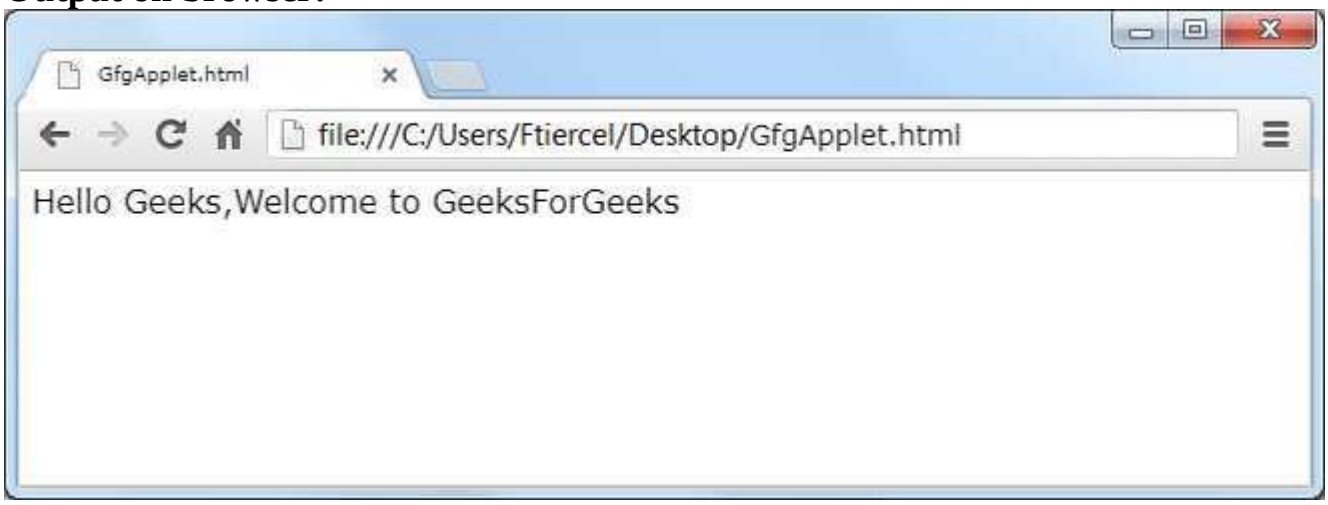
- **Code:** It specifies the name of the applet class to load into the browser.
- **Width:** It specifies the width of an applet.
- **Height:** It sets the height of an applet.

GfgApplet.html

```
<html>
```

```
<body>
<applet code="GfgApplet.class" width=300 height=100></applet>
</body>
</html>
```

Output on browser:



The applet GfgApplet.class is loaded into the browser when you access GfgApplet.html.

To load applet programs, the browser must have java enabled.

Double click on GfgApplet.html

This **won't work** on browser as we don't have the **proper plugins.**

You can run directly applet viewer using appletviewer tool

- **Javac GfgApplet.java**
- **Appletviewer GfgApplet.html**

2. Using an Applet Viewer to run the applet:
It is a java application that allows you to view applets. It's similar to a mini-browser that will enable you to see how an applet might appear in a browser. It recognizes the APPLET tag and uses it during the creation process. The APPLET tag should be written in the source code file, with comments around it.

- Write HTML APPLET tag in comments in the source file.
- Compile the applet source code using javac.
- Use applet viewer ClassName.class to view the applet.

```
// Java program to run the applet

// using the applet viewer

import java.awt.*;

import java.applet.*;

public class GfgApplet extends Applet

{

    public void paint(Graphics g)

    {

        g.drawString(“Hello Geeks,Welcome to GeeksforGeeks”,20,20);

    }

   }

/*

<applet code="GfgApplet" width=300 height=100>

</applet>

*/
```

To use the applet viewer utility to run the applet, type the following at the command prompt:

c:\>javac GfgApplet.java
c:\>appletviewer GfgApplet.java
Output:

6.Passing parameters to applets

The APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the getParameter( ) method. It returns the value of the specified parameter in the form of a String object.

Param Tag

The <param> tag is a sub tag of the <applet> tag. The <param> tag contains two attributes: *name* and *value* which are used to specify the name of the parameter and the value of the parameter respectively. For example, the param tags for passing name and age parameters looks as shown below:

<param name=”name” value=”Ramesh” />
<param name=”age” value=”25″ />

Now, these two parameters can be accessed in the applet program using the *getParameter()* method of the *Applet* class.

getParameter() method:-

The *getParameter()* method of the *Applet* class can be used to retrieve the parameters passed from the HTML page. The syntax of *getParameter()* method is as follows:

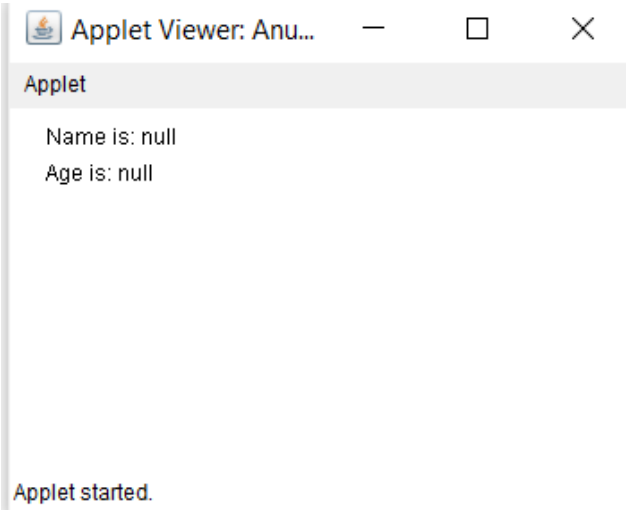
String getParameter(String param-name)

Let’s look at a sample program which demonstrates the <param> HTML tag and the *getParameter()* method:

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet
{
    String n;
    String a;
    public void init()
    {
        n = getParameter("name");
        a = getParameter("age");
    }
```

```
        public void paint(Graphics g)
        {
                g.drawString("Name is: " + n, 20, 20);
                g.drawString("Age is: " + a, 20, 40);
        }
}
/*
        <applet code="MyApplet" height="300" width="500">
                <param name="name" value="Ramesh" />
                <param name="age" value="25" />
        </applet>
*/
```

OutPut:-



Next Topic:-

The HTML APPLET Tag:

< APPLET

[CODEBASE = codebaseURL]

CODE = appletFile

[ALT = alternateText]

[NAME = appletInstanceName]

WIDTH = pixels

HEIGHT = pixels

[ALIGN = alignment]

[VSPACE = pixels]

[HSPACE = pixels]

>

[< PARAM NAME = AttributeName VALUE = AttributeValue>]

[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]

. . .

[HTML Displayed in the absence of Java]

</APPLET>

Let’s take a look at each part now.

CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet’s executable class file (specified by the CODE tag).

CODE :CODE is a required attribute that gives the name of the file containing your applet’s compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT :The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can’t currently run Java applets.

NAME: NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them.

WIDTH AND HEIGHT :WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet.The possible values:LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP,

ABSMIDDLE,and ABSBOTTOM.

VSPACE AND HSPACE :These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet.

PARAM NAME AND VALUE: The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method.

What is the use of repaint ()?

repaint(): This method cannot be overridden. It controls the update() -> paint() cycle. We can call this method to get a component to repaint itself. the repaint() method is invoked to refresh the viewing area i.e. you call it when you have new things to display.

Unit-5

Swings

1. Introduction
2. Limitations of AWT
3. MVC Architecture
4. Components
5. Containers
6. Exploring Swing:-
    a. JApplet
    b. JFrame
    c. JComponent
    d. ImageIcon
    e. JLabel
    f. JTextfield
    g. JButton
    h. JCheckBox
    i. JList
    j. JRadiobutton
    k. JComboBox
    l. JTabbedPane
    m. JScrollPane

1.Introduction

AWT is used for creating GUI in Java. However, the AWT components are internally depends on native methods like C functions and operating system equivalent and hence problems related to portability arise (look and feel. Ex. Windows window and MAC window). And, also AWT components are heavy weight. It means AWT components take more system resources like memory and processor time.

Due to this, Java soft people felt it is better to redevelop AWT package without internally taking the help of native methods. Hence all the classes of AWT are extended to form new classes and a new class library is created. This library is called JFC (Java Foundation Classes).

Java Foundation Classes (JFC):

JFC is an extension of original AWT. It contains classes that are completely portable, since the entire JFC is developed in pure Java. Some of the features of JFC are:

1. JFC components are light-weight: Means they utilize minimum resources.
2. JFC components have same look and feel on all platforms. Once a component is

created, it looks same on any OS.

3. JFC offers “pluggable look and feel” feature, which allows the programmer to change look and feel as suited for platform. For, ex if the programmer wants to display window-style button on Windows OS, and Unix style buttons on Unix, it is possible.

4. JFC does not replace AWT, but JFC is an extension to AWT. All the classes of JFC are derived from AWT and hence all the methods in AWT are also applicable in JFC.

So, JFC represents class library developed in pure Java which is an extension to AWT and swing is one package in JFC, which helps to develop GUIs and the name of the package is

import javax.swing.*;

Here x represents that it is an ‘extended package’ whose classes are derived from AWT package.

Swing Defination:-

Swing is a framework or API that is used to create GUI (or) window-based applications.It is an advanced version of AWT(Abstract Window Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, jRadioButton, JCheckbox, jMenu, JColorChooser etc.

Swing is a Java Foundation Classes [JFC] library and an extension of the Abstract Window Toolkit [AWT]. Swing offers much-improved functionality over AWT, new components, expanded components features, and excellent event handling with drag-and-drop support.

2.Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are platform-dependent.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.
- AWT components is converted by the native code of the operating system.
- AWT components do not support features like icons and tool-tips.
- AWT doesn’t follows MVC(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.

No.	AWT	Swing
1)	AWT components are **platform-dependent**.	Swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser and` etc.
4)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

3.MVC Architecture

The Model-View-Controller Architecture

Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

Model

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable “thumb,” its minimum and maximum values, and the thumb’s width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component’s visual representation.

View

The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the

close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

Figure 1-6 shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scrollbar, within the limits defined by the endpoints, and that pushing on the end buttons is acceptable as well. The result is a fully functional MVC scrollbar.

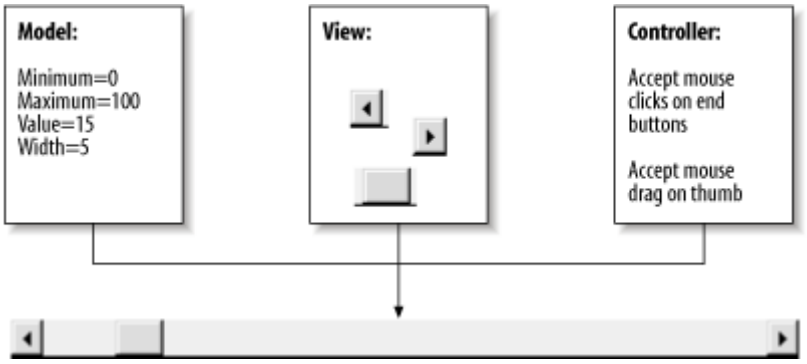


Figure 1-6. The three elements of a model-view-controller architecture

MVC Interaction

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated. Let’s continue discussing the scrollbar component.

We already know that the view cannot render the scrollbar correctly without obtaining information from the model first. In this case, the scrollbar does not know where to draw its “thumb” unless it can obtain its current position and width relative to the minimum and

maximum. Likewise, the view determines if the component is the recipient of user events, such as mouse clicks. (For example, the view knows the exact width of the thumb; it can tell whether a click occurred over the thumb or just outside of it.) The view passes these events on to the controller, which decides how to handle them. Based on the controller's decisions, the values in the model may need to be altered. If the user drags the scrollbar thumb, the controller reacts by incrementing the thumb's position in the model. At that point, the whole cycle repeats. The three elements, therefore, communicate their data as shown in Figure 1-7.

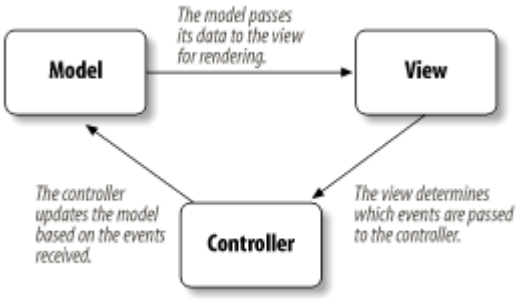


Figure 1-7. Communication through the model-view-controller architecture

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 1-8.

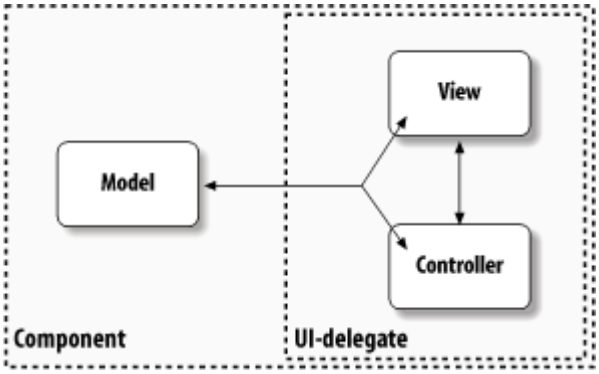


Figure 1-8. With Swing, the view and the controller are combined into a UI-delegate object

So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table, you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly In the same manner, separating the delegate from the model gives the user the added benefit of choosing what a component looks like without affecting any of its data. By using this approach, in conjunction with the lightweight design, Swing can provide each component with its own pluggable L&F.

components and containers.

However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a **component is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components.** Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container.

4.Components:

Defination:-

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of Jcomponents are the JButtons, JCheckBoxes, and JScrollbars of a typical graphical user interface.

In general, Swing components are derived from the JComponent class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. All of Swing's components are represented by classes defined within the package javax.swing. The following figure shows hierarchy of classes of javax.swing.

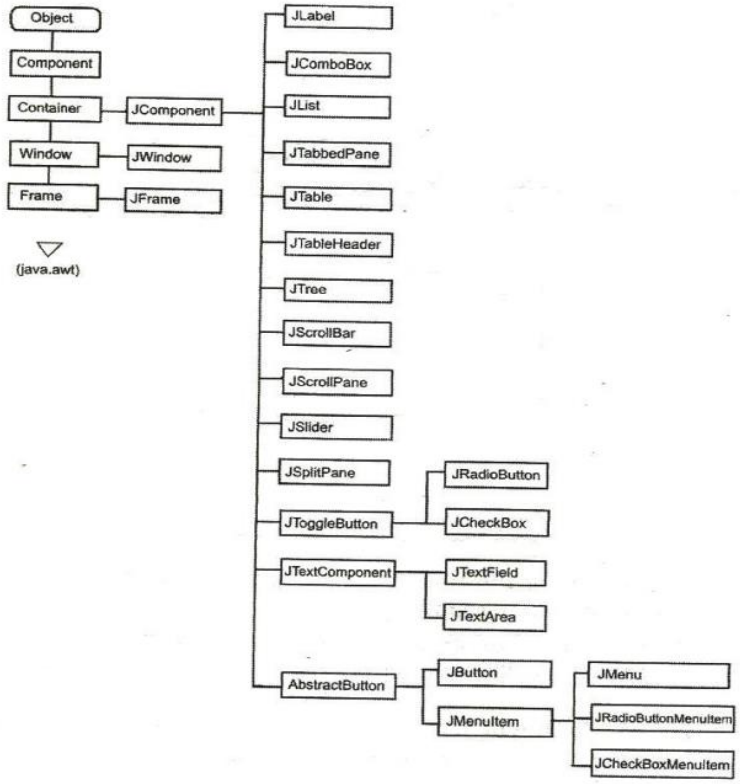


Fig. Swing Class Hierarchy

5.Containers:

Swing defines two types of containers.

1.Top-level Container

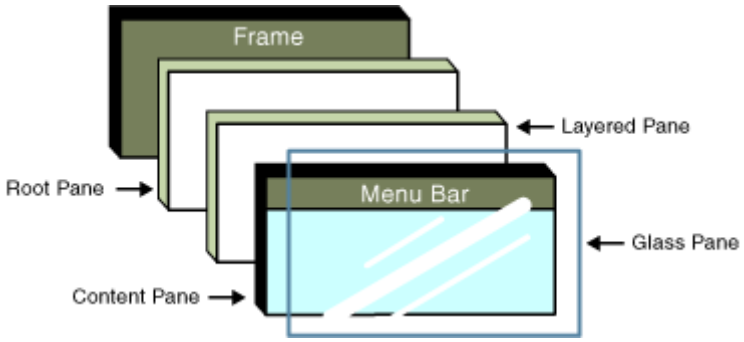
2.Lightweight Container

1. **Top-level containers**/ Root containers: JFrame, JApplet,JWindow, and JDialog.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container.

Furthermore, every containment hierarchy must begin with a top-level container.

The one most commonly used for applications are JFrame and JApplet. Unlike Swing's other components, the top-level containers are heavyweight. Because they inherit AWT classes Component and Container. Whenever we create a top level container four sub-level containers are automatically created:



- Glass pane (JGlass)
- Root pane (JRootPane)
- Layered pane (JLayeredPane)
- Content pane

Glass pane: This is the first pane and is very close to the monitor's screen. Any components to be displayed in the foreground are attached to this glass pane. To reach this glass pane we use getGlassPane() method of JFrame class, which return Component class object.

Conent pane: This pane is below the glass pane. most of all, Individual components are attached to this pane. To reach this pane, we can call getContentPane() method of JFrame class which returns Container class object.

Layered pane: This pane is below the content pane. When we want to take several components as a group, we attach them in the layered pane. We can reach this pane by calling getLayeredPane() method of JFrame class which returns JLayeredPane class object.

Root Pane: This is bottom most pane. Any components to be displayed in the background are displayed in this frame. To go to the root pane, we can use getRootPane() method of JFrame class, which returns JRootPane object.

2. Lightweight containers – containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components.

7. Exploring Swing:-

1.JApplet

1. Swing Based applets are similar to AWT-based applets, but with an important difference:A swing applet extends JApplet rather than Applet.
2. JApplet is derived from Applet.Thus,JApplet includes all of the functionality found in Applet and adds support for swing.
3. JApplet is a top-level Swing Container, which means that it is not derived from JComponent.
4. Swing applets also follow same applet life cycle methods init(), start(), paint(),stop() and destroy().
5. JApplet is rich with functionality that is not found in Applet. For example, JApplet supports various "panes," such as the content pane, the glass pane, and the root pane.
6. One difference between Applet and JApplet is, When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object.
7. The content pane can be obtained via the method shown here:

   Container getContentPane( )

8. The add( ) method of Container can be used to add a component to a content pane.

Its form is shown here:

void add(comp)

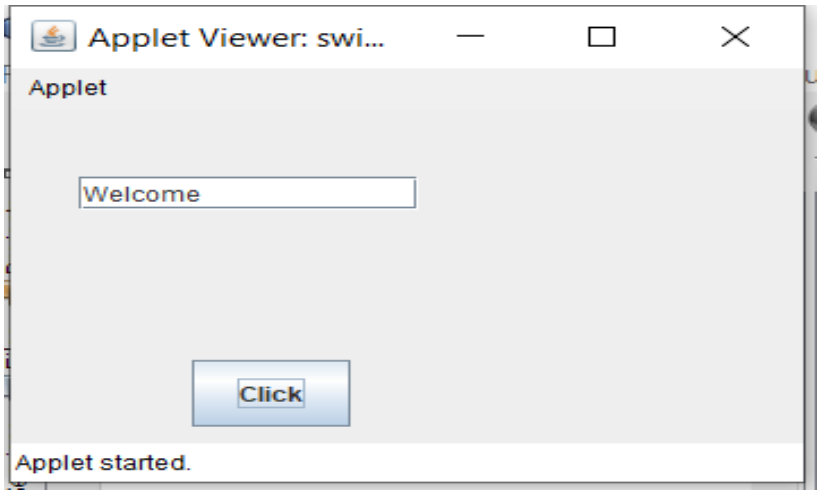
Here, comp is the component to be added to the content pane.

Example Program Executing appletviewer:-

```
1. import java.applet.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. /*<applet code="EventJApplet.class" width="300" height="300">  </applet> */
5. public class EventJApplet extends JApplet implements ActionListener{
6. JButton b;
7. JTextField tf;
8. public void init(){
9.
10.tf=new JTextField();
11.tf.setBounds(30,40,150,20);
12.
13.b=new JButton("Click");
14.b.setBounds(80,150,70,40);
15.
16.add(b);add(tf);
17.b.addActionListener(this);
18.
19.setLayout(null);
20.}
21.
22.public void actionPerformed(ActionEvent e){
23.tf.setText("Welcome");
24.}
25.}
```

In the above example, we have created all the controls in init() method because it is invoked only once.

Output:-



2.JFrame

Frame represents a window with a title bar and borders. Frame becomes the basis for creating the GUIs for an application because all the components go into the frame. The Frame in Java Swing is defined in class javax.swing.JFrame. JFrame class inherits the java.awt.Frame class. JFrame is like the main window of the GUI application using swing.

To create a frame, we have to create an object to JFrame class in swing as

JFrame jf=new JFrame(); // create a frame without title
JFrame jf=new JFrame("title"); // create a frame with title
To close the frame, use setDefaultCloseOperation() method of JFrame class
setDefaultCloseOperation(constant)

where constant values are

Constant	Description
JFrame.EXIT_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DO_NOTHING_ON_CLOSE	This will not perform any operation upon clicking close button
JFrame.DISPOSE_ON_CLOSE	This closes the application upon clicking the close button

JFrame.HIDE_ON_CLOSE	This hides the frame upon clicking close button

We can create a JFrame window object using two approaches:

#1) By Extending The JFrame Class

The first approach is creating a new class to construct a Frame. This class inherits from the JFrame class of the javax.swing package.

The following program implements this approach.

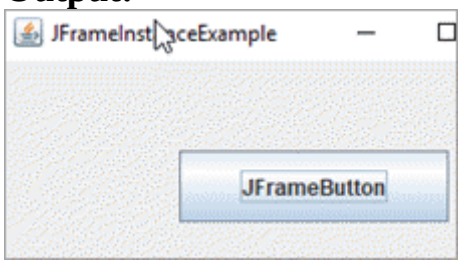
```
import javax.swing.*;

class FrameInherited extends JFrame{    //inherit from JFrame class

   JFrame f;

   FrameInherited(){

      JButton b=new JButton("JFrame_Button");//create button object

      b.setBounds(100,50,150, 40);


      add(b);//add button on frame

      setSize(300,200);

      setLayout(null);

      setVisible(true);

   }

}

public class Main {

   public static void main(String[] args) {

     new FrameInherited(); //create an object of FrameInherited class

   }

}
```

Output:

#2) By Instantiating The JFrame Class

```
import javax.swing.*;

public class Main {


   public static void main(String[] args) {
      JFrame f=new JFrame("JFrameInstanceExample");//create a JFrame object


      JButton b=new JButton("JFrameButton");//create instance of JButton
      b.setBounds(100,50,150, 40);//dimensions of JButton object


      f.add(b);//add button in JFrame


      f.setSize(300,200);//set frame width = 300 and height = 200
      f.setLayout(null);//no layout manager specified
      f.setVisible(true);//make the frame visible
   }
}
```

Output:



c. JComponent

The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

The JComponent class extends the Container class which itself extends Component. The Container class has support for adding components to the container.

Constructor: JComponent();

The following are the JComponent class's methods to manipulate the appearance of the component.

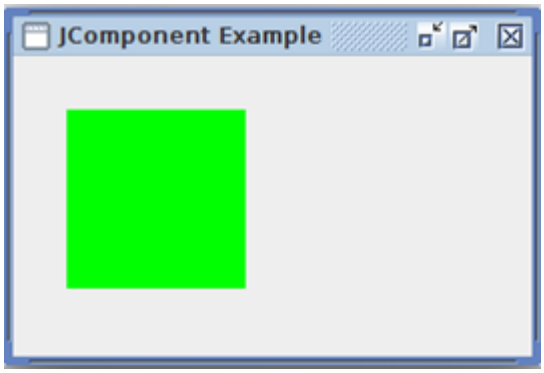
public int getWidth ()	Returns the current width of this component in pixel.
public int getHeight ()	Returns the current height of this component in pixel.
public int getX()	Returns the current x coordinate of the component's top-left corner.
public int getY ()	Returns the current y coordinate of the component's top-left corner.
public java.awt.Graphics getGraphics()	Returns this component's Graphics object you can draw on. This is useful if you want to change the appearance of a component.
public void setBackground (java.awt.Color bg)	Sets this component's background color.
public void setEnabled (boolean enabled)	Sets whether or not this component is enabled.
public void setFont (java.awt.Font font)	Set the font used to print text on this component.
public void setForeground (java.awt.Color fg)	Set this component's foreground color.
public void setToolTipText(java.lang.String text)	Sets the tool tip text.
public void setVisible (boolean visible)	Sets whether or not this component is visible.

Example program:-

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
class MyJComponent extends JComponent {
    public void paint(Graphics g) {
      g.setColor(Color.green);
      g.fillRect(30, 30, 100, 100);
    }
}
public class JComponentEx {
    public static void main(String[] arguments) {
```

```
        MyJComponent com = new MyJComponent();
        // create a basic JFrame
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("JComponent Example");
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // add the JComponent to main frame
        frame.add(com);
        frame.setVisible(true);
      }
}
```

Output:-



d. ImageIcon

The class **ImageIcon** is an implementation of the Icon interface that paints Icons from Images.

Class Declaration

Following is the declaration for **javax.swing.ImageIcon** class −

```
public class ImageIcon
   extends Object
      implements Icon, Serializable, Accessible
```

An implementation of the Icon interface that paints Icons from Images. Images that are created from a URL, filename or byte array are preloaded using MediaTracker to monitor the loaded state of the image. For further information and examples of using image icons, see How to Use Icons in The Java Tutorial.

Class Constructors

Sr.No.	Constructor & Description

1	**ImageIcon()** Creates an uninitialized image icon.
2	**ImageIcon(byte[] imageData)** Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
3	**ImageIcon(byte[] imageData, String description)** Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
4	**ImageIcon(Image image)** Creates an ImageIcon from an image object.
5	**ImageIcon(Image image, String description)** Creates an ImageIcon from the image.
6	**ImageIcon(String filename)** Creates an ImageIcon from the specified file.
7	**ImageIcon(String filename, String description)** Creates an ImageIcon from the specified file.
8	**ImageIcon(URL location)** Creates an ImageIcon from the specified URL.
9	**ImageIcon(URL location, String description)** Creates an ImageIcon from the specified URL.

Class Methods

Sr.No.	Method & Description
1	**String getDescription()** Gets the description of the image.
2	**int getIconHeight()**

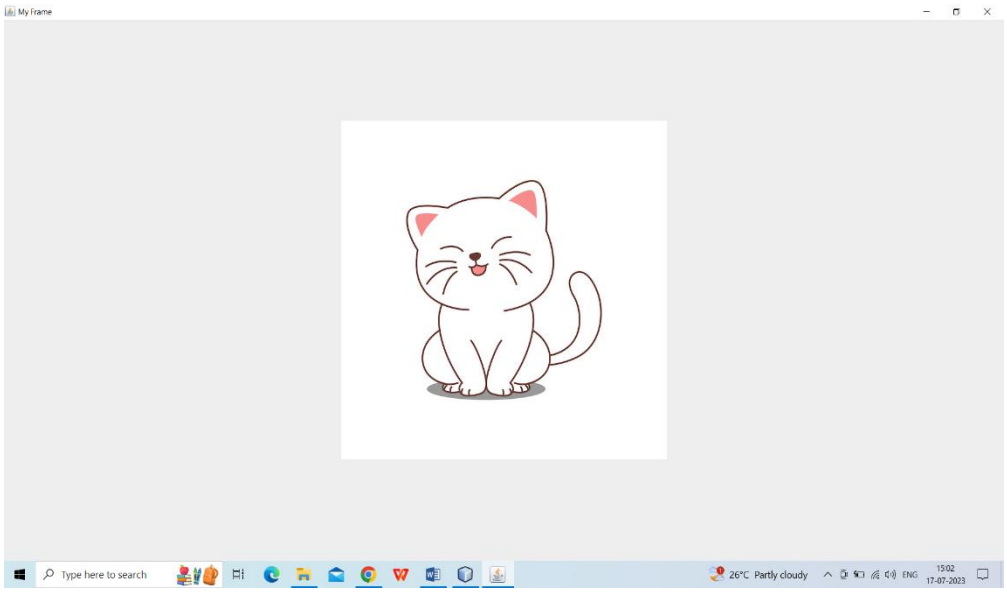
	Gets the height of the icon.
3	**int getIconWidth()** Gets the width of the icon.
4	**Image getImage()** Returns this icon's Image.
5	**void paintIcon(Component c, Graphics g, int x, int y)** Paints the icon.
6	**void setDescription(String description)** Sets the description of the image.
7	**void setImage(Image image)** Sets the image displayed by this icon.

Example program:-

```
import java.awt.BorderLayout;

import javax.swing.ImageIcon;

import javax.swing.JFrame;

import javax.swing.JLabel;

   public class ImageIconEx {

      public static void main(String args[])

      {

         TimeFrame frame = new TimeFrame();
```

```
    }

}

class TimeFrame extends JFrame

{

    //Image icon = Toolkit.getDefaultToolkit().getImage("me.jpg");

    ImageIcon icon = new
ImageIcon("C:\\Users\\Admin\\Pictures\\pictures\\cat.jpg","Anusha");

    JLabel label = new JLabel(icon);

    public TimeFrame(){

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setTitle("My Frame");

        setSize(500,400);

        //this.setIconImage(icon);

        add(label);

        setVisible(true);

    }

}
```

Output:-



e. JLabel

JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus. JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.

Fields:-

Public static final int LEFT

Public static final int RIGHT

Public static final int CENTER

Public static final int LEADING

Public static final int TRAILING

Constructor of the class are :

JLabel() : creates a blank label with no text or image in it.

JLabel(String s) : creates a new label with the string specified.

JLabel(Icon i) : creates a new label with a image on it.

JLabel(String s, Icon i, int align) : creates a new label with a string, an image and a specified horizontal alignment

Commonly used methods of the class are :

getIcon() : returns the image that the label displays

setIcon(Icon i) : sets the icon that the label will display to image i

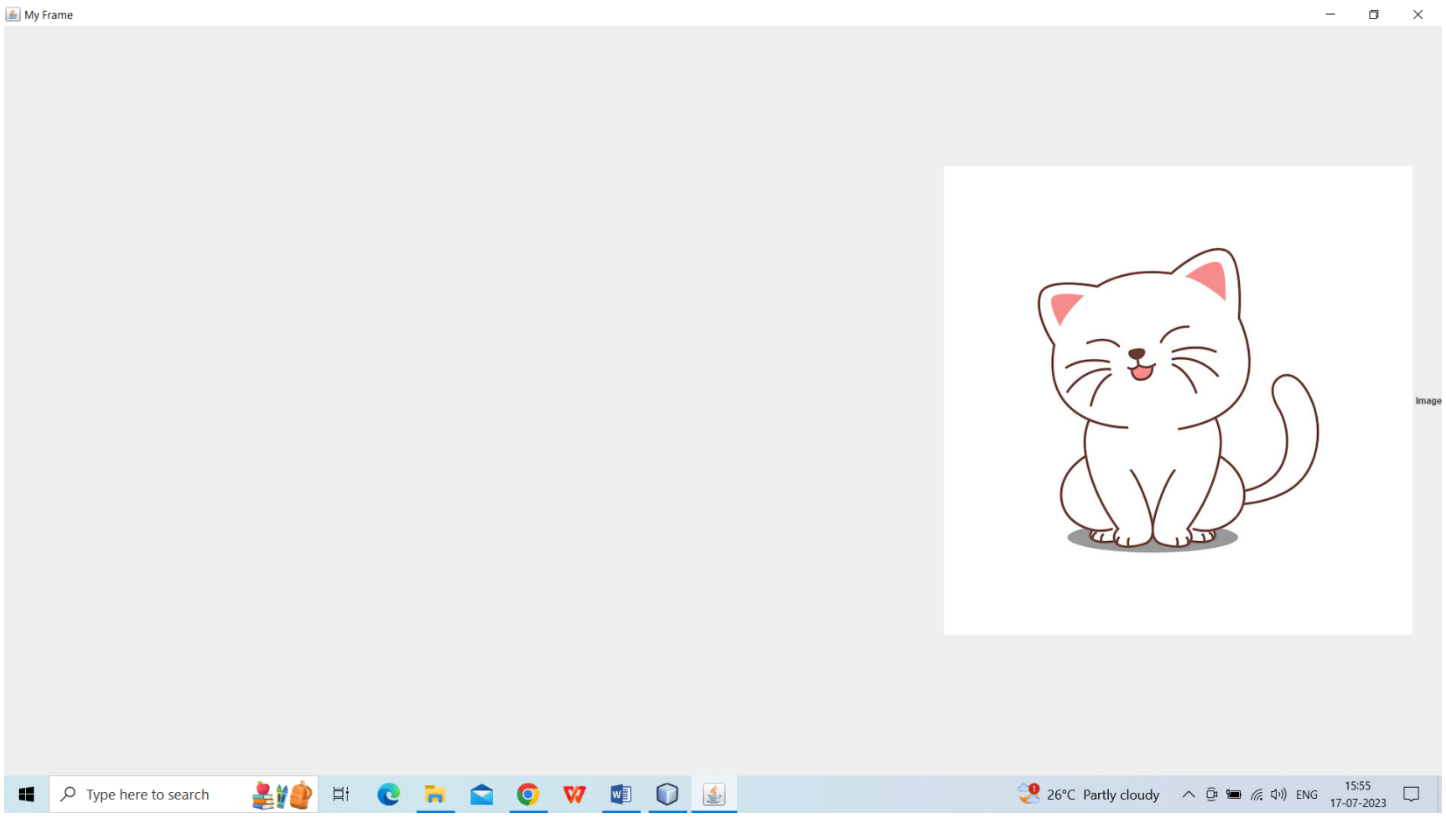
getText() : returns the text that the label will display

setText(String s) : sets the text that the label will display to string s

Example program:-

```
import java.awt.*;
import javax.swing.ImageIcon;
import javax.swing.*;
public class JLabelEx extends JFrame{
  public JLabelEx(){
     ImageIcon icon = new ImageIcon("C:\\Users\\Admin\\Pictures\\pictures\\cat.jpg");
JLabel label = new JLabel("Image",icon,JLabel.RIGHT);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setTitle("My Frame");
setSize(500,400);
add(label);
setVisible(true);

       }
  public static void main(String args[])
{
 JLabelEx frame = new JLabelEx();
}

   }
```

Output:-



f.JTextfield

The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField( )

JTextField(int cols)

JTextField(String s, int cols)

JTextField(String s)

Here, s is the string to be presented, and cols is the number of columns in the text field.

The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a

JTextField object is created and is added to the content pane.

Example:

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add text field to content
 jtf = new JTextField(10);
contentPane.add(jtf);
}
}
```

Output:-

g.JButton Class

The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

JButton(Icon i)

JButton(String s)

JButton(String s, Icon i)

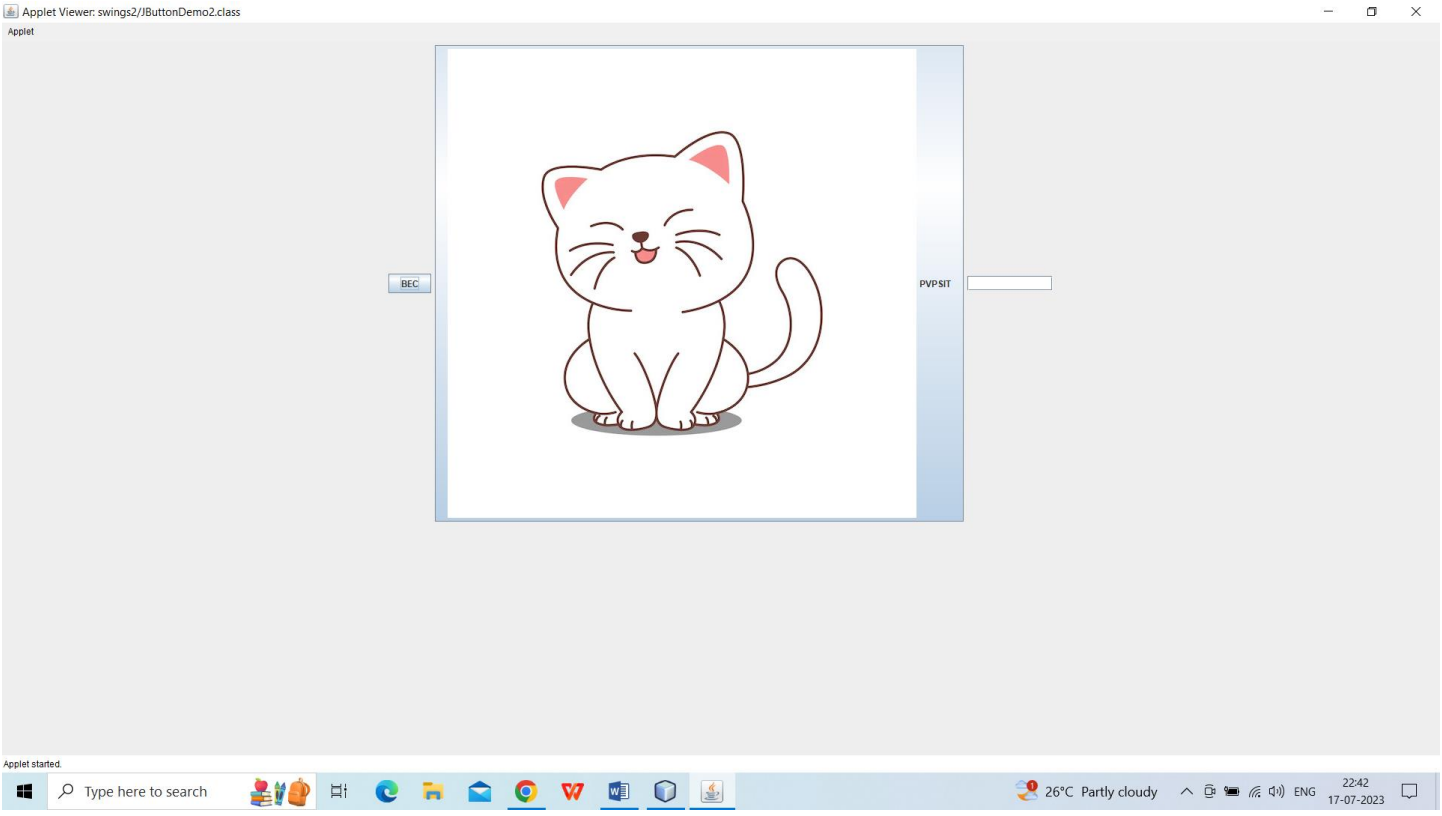
Here, s and i are the string and icon used for the button.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo2" width=250 height=300>
</applet>
*/
public class JButtonDemo2 extends JApplet implements ActionListener
{
JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
JButton jb1 = new JButton("BEC");
jb1.addActionListener(this);
contentPane.add(jb1);
// Add buttons to content pane
ImageIcon pvp = new ImageIcon("pvp.jpg");
JButton jb2 = new JButton("PVPSIT",pvp);
jb2.setActionCommand("PVPSIT");
```

```
jb2.addActionListener(this);
contentPane.add(jb2);
jtf = new JTextField(10);
contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae)
{
jtf.setText(ae.getActionCommand());
}
}
```

Output:-



h.Check Boxes:

The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. Its immediate super class is JToggleButton, which provides support for two-state buttons (true or false). Some of its constructors are shown here:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
```

JCheckBox(String s, Icon i)

JCheckBox(String s, Icon i, boolean state)

Here, i is the icon for the button. The text is specified by s. If state is true, the check box is initially selected. Otherwise, it is not. The state of the check box can be changed via the following method:

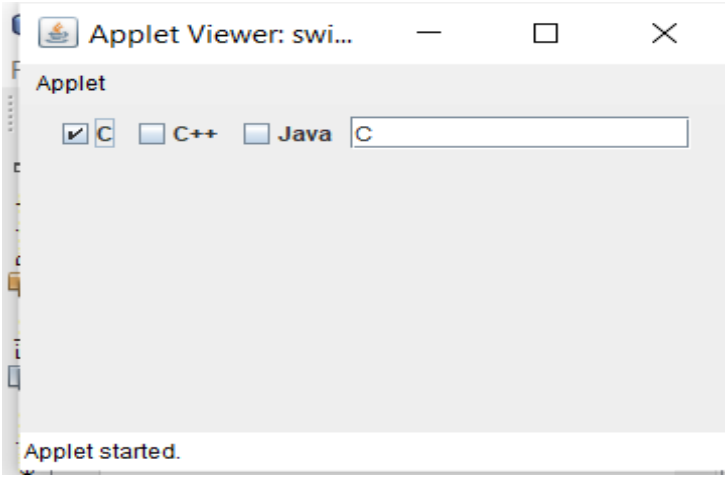
void setSelected(boolean state)

Here, state is true if the check box should be checked. When a check box is selected or deselected, an item event is generated. This is handled by itemStateChanged( ). Inside itemStateChanged( ), the getItem( ) method gets the JCheckBox object that generated the event. The getText( ) method gets the text for that check box and uses it to set the text inside the text field.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo2" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo2 extends JApplet implements ItemListener
{
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
JCheckBox cb = new JCheckBox("C", true);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("C++");
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java");
cb.addItemListener(this);
```

```
contentPane.add(cb);
// Add text field to the contentpane
 jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```



i.JList :

JList is part of Java Swing package . JList is a component that displays a set of Objects and allows the user to select one or more items . JList inherits JComponent class. JList is a easy way to display an array of Vectors .

Constructor for JList are :

JList(): creates an empty blank list

JList(E [ ] l) : creates an new list with the elements of the array.

JList(ListModel d): creates a new list with the specified List Model

JList(Vector l) : creates a new list with the elements of the vector

Commonly used methods are :

method explanation

getSelectedIndex() returns the index of selected item of the list

getSelectedValue() returns the selected value of the element of the list

setSelectedIndex(int i) sets the selected index of the list to i

getSelectedValuesList() returns a list of all the selected items.

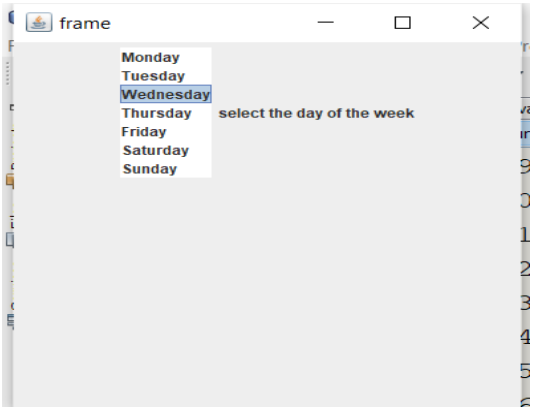
getSelectedIndices() returns an array of all of the selected indices, in increasing order

Example program:-

```
// java Program to create a simple JList
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class JListEx
{

 //frame
 static JFrame f;

 //lists
 static JList b;

 //main class
 public static void main(String[] args)
 {
 //create a new frame
 f = new JFrame("frame");

 //create a object
 //JListEx s=new JListEx();

 //create a panel
 JPanel p =new JPanel();

 //create a new label
 JLabel l= new JLabel("select the day of the week");
```

```
//String array to store weekdays
String week[]= { "Monday","Tuesday","Wednesday",
"Thursday","Friday","Saturday","Sunday"};

//create list
b= new JList(week);

//set a selected index
b.setSelectedIndex(2);

//add list to panel
p.add(b);
p.add(l);

f.add(p);

//set the size of frame
f.setSize(400,400);
f.setVisible(true);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//f.show();
 }
}
```

Output:-



j.Radio Buttons:

Radio buttons are supported by the JRadioButton class, which is a concrete implementation of AbstractButton. Its immediate superclass is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

JRadioButton(Icon i)

JRadioButton(Icon i, boolean state)

JRadioButton(String s)

JRadioButton(String s, boolean state)

JRadioButton(String s, Icon i)

JRadioButton(String s, Icon i, boolean state)

Here, i is the icon for the button. The text is specified by s. If state is true, the button is initially selected. Otherwise, it is not.

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The ButtonGroup class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton ab)

Here, ab is a reference to the button to be added to the group.

Radio button presses generate action events that are handled by actionPerformed( ).

The getActionCommand( ) method gets the text that is associated with a radio button and

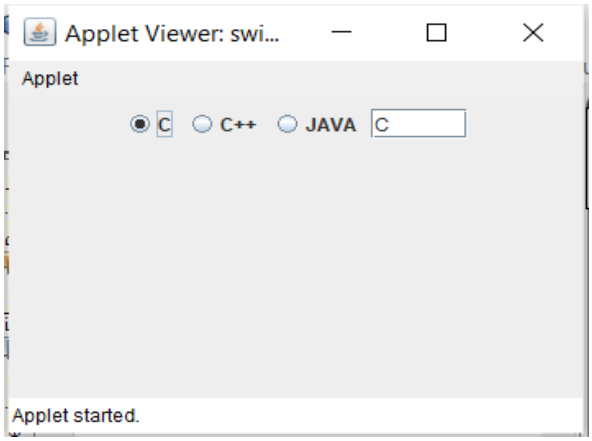
uses it to set the text field.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements ActionListener
{
JTextField tf;
```

```
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton c = new JRadioButton("C");
c.addActionListener(this);
contentPane.add(c);
JRadioButton cpp = new JRadioButton("C++");
cpp.addActionListener(this);
contentPane.add(cpp);
JRadioButton java = new JRadioButton("JAVA");
java.addActionListener(this);
contentPane.add(java);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(c);
bg.add(cpp);
bg.add(java);
// Create a text field and add it to the content pane
tf = new JTextField(5); contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae)
{ tf.setText(ae.getActionCommand());
}
}
```

Output:-



k.JComboBox:

Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent.

A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.

Two of JComboBox's constructors are shown here:

JComboBox( )

JComboBox(Vector v)

Here, v is a vector that initializes the combo box. Items are added to the list of

choices via the addItem( ) method, whose signature is shown here:

void addItem(Object obj)

Here, obj is the object to be added to the combo box.

By default, a JComboBox component is created in read-only mode, which means the user can only pick one item from the fixed options in the drop-down list. If we want to allow the user to provide his own option, we can simply use the setEditable() method to make the combo box editable.

The following example contains a combo box and a label. The label displays an icon.

The combo box contains entries for "PVPSIT", "BEC", and "VRSEC". When a college is

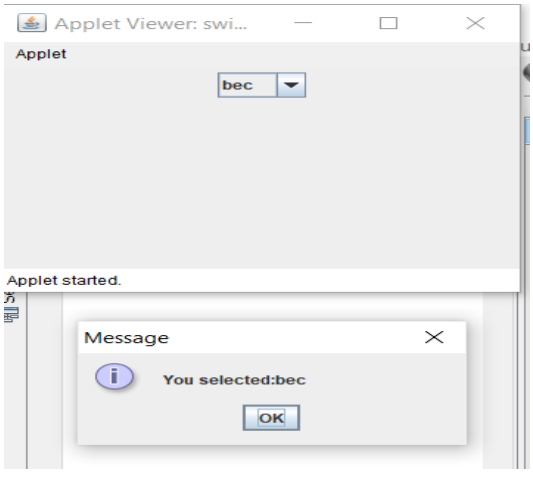
selected, the label is updated to display the flag for that country

Example:

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

/* <applet code="JComboBoxDemo" width=300 height=100>

</applet>

*/
```

```
public class JComboBoxDemo extends JApplet implements ItemListener
{
Container contentPane;
JComboBox jc;
public void init()
{
// Get content pane
contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it to the
jc = new JComboBox();
jc.addItem("pvp");
jc.addItem("bec");
jc.addItem("vrsec");
jc.addItemListener(this);
contentPane.add(jc);
}
public void itemStateChanged(ItemEvent ie)
{
String s = (String)jc.getItemAt(jc.getSelectedIndex());
JFrame f=new JFrame();
JOptionPane.showMessageDialog(f,"You selected:"+s);
}}
```

Output:-



l.JTabbedPane:

A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. We will use its default constructor. Tabs are defined via the following method:

void addTab(String str, Component comp)

Here, str is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added.

The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a JTabbedPane object.
2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane. The first tab is titled“Cities” and contains four buttons. Each button displays the name of a city. The second tab is titled “Colors” and contains three check boxes. Each check box displays the name of a color. The third tab is titled “Language” and contains radio buttons.

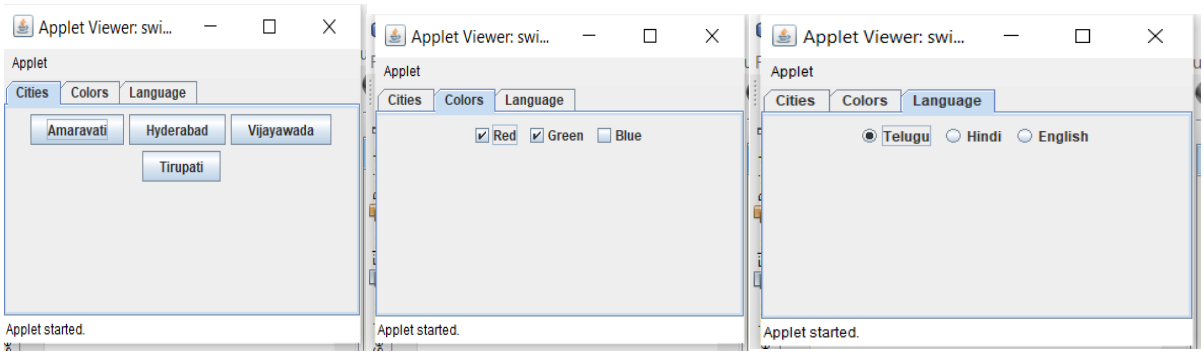
Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400
height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet
{
public void init()
```

```
{
Container contentPane = getContentPane();
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Language", new LanguagesPanel());
contentPane.add(jtp);
}
}
class CitiesPanel extends JPanel
{
public CitiesPanel()
{
JButton b1 = new JButton("Amaravati");
add(b1);
JButton b2 = new JButton("Hyderabad");
add(b2);
JButton b3 = new JButton("Vijayawada");
add(b3);
JButton b4 = new JButton("Tirupati");
add(b4);
}
}
class ColorsPanel extends JPanel
{
public ColorsPanel()
{
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
```

```
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}
class LanguagesPanel extends JPanel
{
public LanguagesPanel()
{
    ButtonGroup bg = new ButtonGroup();

JRadioButton rb1 = new JRadioButton("Telugu");
bg.add(rb1);
JRadioButton rb2 = new JRadioButton("Hindi");
bg.add(rb2);
JRadioButton rb3 = new JRadioButton("English");
bg.add(rb3);
add(rb1);
add(rb2);
add(rb3);
}
}
```

Output:-



m.JScrollPane

JScrollPane provides a **scrollable view of a component, where a component maybe an image, table, tree etc**. A JScrollPane can be added to a top-level container like JFrame or a component like JPanel. JScrollPane is another **lightweight component** which extends **JComponent** class.

Constructors of JScrollPane

Constructor	Description
public JScrollPane()	Creates an empty JScrollPane with no viewport, where both the horizontal and vertical scrollbars appearing when required.
public JScrollPane(Component view)	Creates a JScrollPane that displays a component within it. This JScrollPane also shows the horizontal and vertical bar, only when its component's contents are larger than the viewing area.

Methods of JScrollPane class

Methods	Description
public setPreferredSize(Dimension d)	Sets the preferred size of JScrollPane.
public int setLayout(LayoytManager managaer)	Sets the layout manager for JScrollPane.

Example Program:-

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


public class JScrollPaneEx
{
public static void main(String... ar)
{
SwingUtilities.invokeLater(new Runnable() {
public void run()
{
new A();
}
});
```

```
}//Closing the main method
}//Closing the class Combo


class A //implements ActionListener
{

JFrame jf;
JPanel jp;
JLabel label1;

A()
{
jf = new JFrame("JScrollPane");
label1 = new JLabel("Displaying a picture ",JLabel.CENTER);

//Creating an ImageIcon object to create a JLabel with image
ImageIcon image = new ImageIcon("nature.jpg");
JLabel label = new JLabel(image, JLabel.CENTER);

//Creating a JPanel and adding JLabel that contains the image
jp = new JPanel(new BorderLayout());
jp.add( label, BorderLayout.CENTER );

//Adding JPanel to JScrollPane
JScrollPane scrollP = new JScrollPane(jp);

//Adding JLabel and JScrollPane to JFrame
jf.add(label1,BorderLayout.NORTH);
jf.add(scrollP,BorderLayout.CENTER);
```

```
jf.setSize(350,270);
jf.setVisible(true);
}

}
```

Output:-