

PDFZilla – Unregistered

PDFZilla - Unregistered

PDFZilla - Unregistered

Module 4

Packages and Interfaces:

Packages, Access Protection, Importing Packages. Interfaces.

Multi Threaded Programming:

What are threads? How to make the classes threadable ; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, producer consumer problems.

Interface

Interface's are another form of creating abstract classes in java.

Abstract classes cannot be used to achieve multiple inheritance, but **interfaces can be used to achieve multiple inheritance in java.**

Interface is a mechanism to achieve abstraction (specifying the task's to be done by derived classes, but interface is not going to implement one)

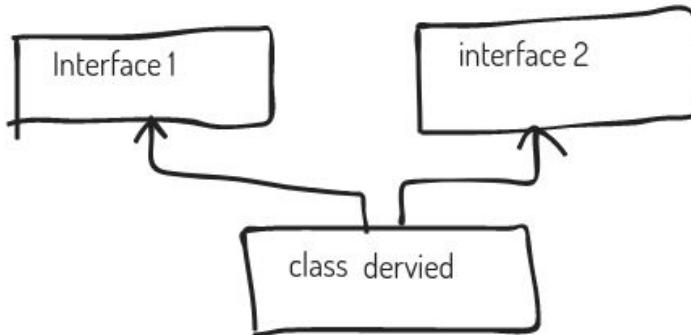
Interface implements abstraction in its basic form, by **not defining any method within it.**

Abstract classes will not implement abstraction to its fullest, because some methods can be defined within it.

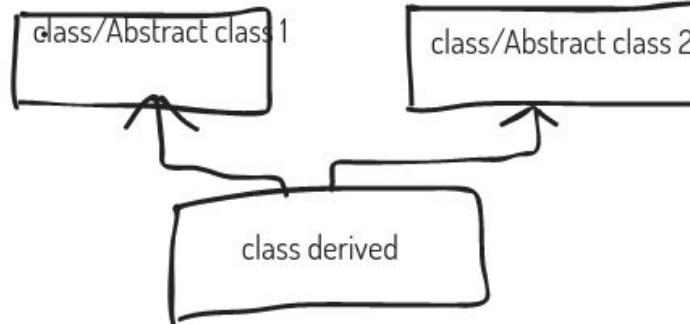
There **can be only abstract methods in the interface.**

Interface

ALLOWED IN JAVA



NOT ALLOWED IN JAVA



Interface

Instance cannot be created for an interface, but a reference can be created.

Java Abstract class & Interface represents “**IS-A**” relationship.

A single class can inherit multiple interfaces. A single class can inherit from only one Abstract class/Non-abstract class.

“**implements**” is the keyword used to inherit the interface.

G.F: <access-specifier> **interface** <interface-name>

{

return-type method-name1(parameter-list); //method declaration

return-type method-name2(parameter-list);

.....

type final varname1 = value; // final and static variables which must be initialized

type final varname2 = value; }

Interface

G.F: of implementing interface

```
class <calss-name> [extends classname] [implements interface [,interface...]]  
{  
....//class body  
}
```

[] indicates optional

If a class **implements** an interface, then all the methods in that must be compulsorily defined by the inherited class.

All methods and variables are implicitly public in an interface.

NOTE: An interface can extend (inherit) another interface.

Interface

Ex: **interface** disp {
 public void display();
 int *i*=90; // by default i is public final static
}

```
class cmp implements disp {  
    public void cmp_method()  
    { System.out.println("method of cmp"); }
```

```
@Override  
public void display()  
{ System.out.println("in cmp display"); }  
}
```

Interface

```
class student implements disp {  
    public void student_method() { System.out.println("method of student"); }  
    @Override  
    public void display() { System.out.println("in student display"); }  
}  
class test {  
    public static void main(String[] args) {  
        disp d = new student();           d.display();  
        //d.student_method(); // CTE  
  
        d = new cmp();                 d.display();  
        //d.cmp_method(); // CTE  
    }  
}
```

Interface : Another example

```
import java.lang.*;  
interface stack  
{  
    void push(Object obj);  
    Object pop();  
    int size=3;  
}
```

```
class intstack implements stack  
{  
    private int top,a[];  
  
    public intstack()  
    { top=-1; a = new int[size]; }
```

Interface : Another example

```
public void push(Object obj){  
    System.out.println(obj.getClass());  
    if (top==size-1)  
        {System.out.println("Integer stack full"); return;}  
  
    top=top+1;  
    a[top] = ((Integer) obj).intValue();  
}  
  
public Object pop() {  
    if (top==-1)  
        {System.out.println("Integer stack empty"); return null; }  
  
    Object obj = new Integer(a[top]);  
    top--;  
    return obj;  
} }
```

Interface : Another example

```
class fstack implements stack {
    int top;      float a[];  
  
    public fstack()
    {   top=-1; a = new float[size]; }  
  
    public void push(Object obj)
    {
        if (top==size-1)
        {System.out.println("Integer stack full"); return;}  
  
        top=top+1;
        a[top] = ((Float)obj).floatValue();
    }
}
```

Interface : Another example

```
public Object pop() {  
    if (top== -1)  
    { System.out.println("Integer stack empty"); return null; }
```

```
Object obj = new Float(a[top]);  
top--;  
return obj;  
} }
```

```
class test {  
    public static void main(String args[])  
    {  
        intstack i = new intstack();  
        i.push( 10 );  
        System.out.println( i.pop() );
```

Interface : Another example

```
fstack j = new fstack();
j.push(10f);
System.out.println( j.pop() );
```

```
System.out.println("One interface Multiple method invocation");
stack s = i; // base class reference pointing to its derived class
s.push(20);
System.out.println( s.pop() );
s = j;
s.push(30f);
System.out.println( s.pop() );
```

Interface : Another example

```
i.push(90); i.push(10);
```

```
int k = i.pop() + i.pop(); //CTE
```

```
k = ((Integer)i.pop()).intValue() + ((Integer)i.pop()).intValue();
```

```
System.out.println(k);
```

```
}
```

```
}
```

i.pop() calls toString() of Integer wrapper class to obtain stringed integer value.

((Integer)i.pop()).intValue() retrieves value of type int.

IMPLEMENTING MULTIPLE INTERFACE

```
interface accept_value
```

```
{ void accept(); }
```

```
interface display_value
```

```
{ void display(); }
```

```
class test implements accept_value, display_value {
```

```
    public void accept() { System.out.println("In accept"); }
```

```
    public void display() { System.out.println("In display"); }
```

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

```
        test a = new test();
```

```
        a.accept();
```

```
        a.display();
```

```
    } }
```

INTERFACE EXTENDING ANOTHER INTERFACE

```
interface accept_value  
{ void accept(); }
```

```
interface display_value extends accept_value  
{ void display(); }
```

```
class test implements display_value {  
    public void accept() { System.out.println("In accept"); }  
    public void display() { System.out.println("In display"); }  
  
    public static void main (String[] args) {  
        test a = new test(); a.accept();  
        a.display();  
    } }
```

PACKAGE

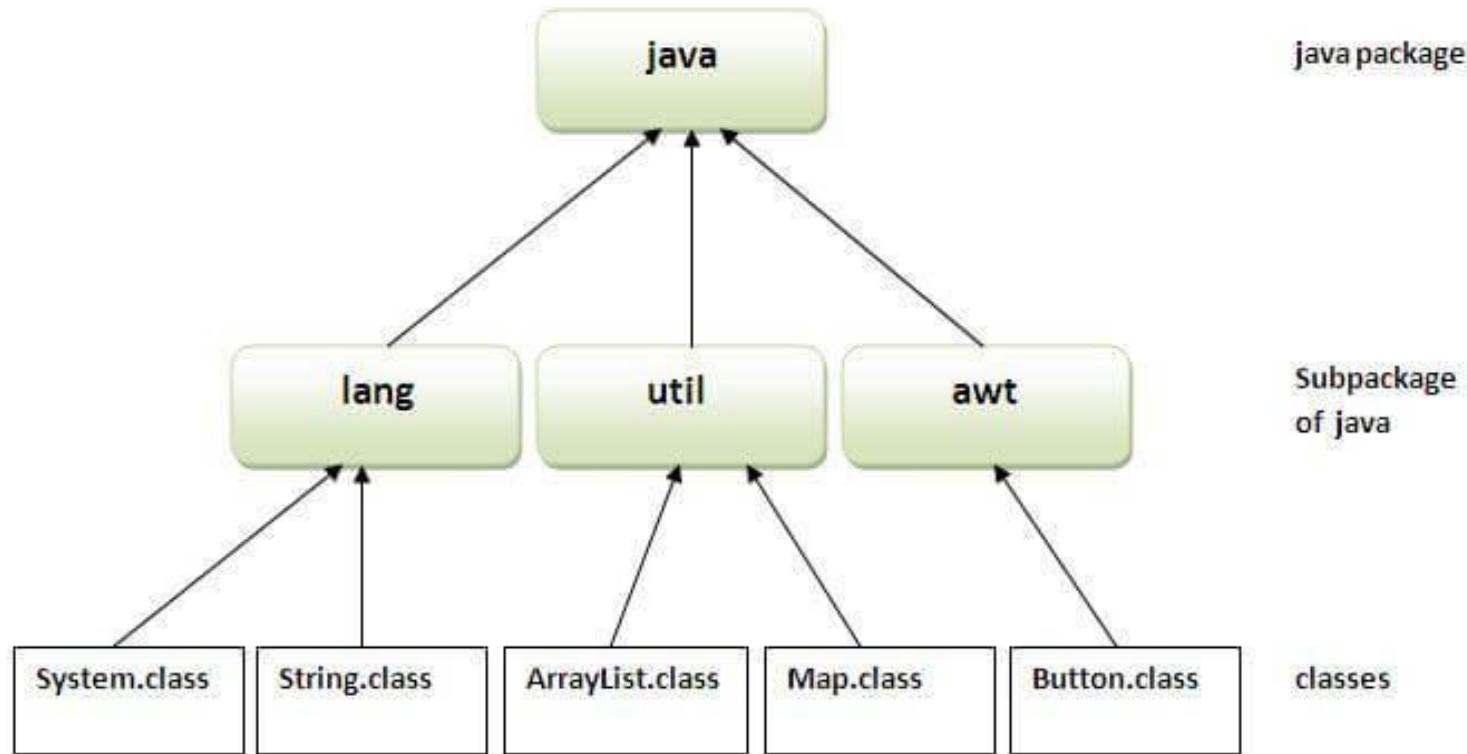
Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts.
Ex: there can be two classes with similar names in two packages.
- Providing controlled access:
protected and default have package level access control.
A protected member is accessible by classes in the same package and its subclasses (outside the package).
A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation or data-hiding.
- Usage of classes, interfaces will be easier by using package names.

Package in java can be categorized in two forms, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

PACKAGE



PACKAGE

Creating a package actually creates a directory with the same name as the package name.
(in Eclipse a default package is created for each project, if a suitable name is provided for the package, then a directory of the same name will be created to hold on to the classes, which are inside the package)

In console based implementation of packages, directory name which matches package name has to be created manually and respective classes must be kept in that directory.

Ex: package college.engineering.cse

college is the directory and main package name

engineering is the sub directory of college and intermediate package name

cse is the subdirectory of engineering and last package name

college ---> engineering ----> cse (Hierarchy of packages)

Any class to be added to this package must be stored in the “cse” directory.

PACKAGE : Creating Packages

```
package college.engineering;  
public class cse  
//public specifier for class, to access CSE class outside the package //class.engineering.cse  
{  
    public static void print()  
    {System.out.println("In college.engineering package class CSE"); }  
}
```

**ONLY ONE PUBLIC CLASS IS ALLOWED IN ONE PACKAGE
ANY OTHER CLASS IN THE SAME PACKAGE MUST NOT BE PUBLIC.**

```
class ise  
{  
    public static void print()  
    { System.out.println("In rns.engineering package class ISE"); }  
}
```

PACKAGE : Creating Packages

```
package college.pucollege;
```

```
public class PUC {  
    public static void print()  
    { System.out.println("In rns.pucollege package class PUC"); }  
}
```

```
//Test.java  
import college.engineering.CSE;  
import college.pucollege.*;  
class Test {  
    public static void main(String args[]) {  
        CSE.print();  
        PUC.print();  
    }  
}
```

PACKAGE : Access Protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

PACKAGE : Access Protection

Any member declared as public can be accessed from anywhere.

Any member declared as private cannot be accessed outside its class.

Any member declared as protected can be accessed outside the current package, but only inside the subclasses . Non-subclasses cannot access protected members.

When a member has a default access specifier, it is visible to subclasses as well as to other classes in the same package.

A class in a package can have only **public, final or default access specifiers**.

When a class is **declared as public**, it is accessible anywhere in the project. (Project is a combination of several packages).

PACKAGE : Access Protection

When a class is public, it must be the **only public class** declared in the file, and the **file must have the same name as the class**.

If a class has **default access**, then it can only be accessed by other classes within the same package.

Ex:

File path: rns\engineering\CSE.java

```
package rns.engineering;  
public class CSE {  
    private int ipri;  
    protected int ipro;  
    public int ipub;  
    int inomod;  
}
```

PACKAGE : Access Protection

File path: rns\engineering\NonSubClassSamePackage.java

```
package rns.engineering;
```

```
public class NonSubClassSamePackage {
```

```
    public void access()
```

```
{
```

```
    System.out.println("In Same Package Non-Subclass");
```

```
    CSE obj = new CSE();
```

```
    //obj.ipri = 90; // CTE
```

```
    obj.ipro = 900;
```

```
    obj.ipub = 9000;
```

```
    obj.inomod = 89;
```

```
}
```

```
}
```

PACKAGE : Access Protection

File path: rns\engineering\SubClassSamePackage.java

```
package rns.engineering;
```

```
public class SubClassSamePackage extends CSE
```

```
{
```

```
    public void access()
```

```
{
```

```
    System.out.println("In Same Package Sub class");
```

```
//ipri=90; // CTE
```

```
ipro = 900; ipub = 9000;
```

```
inomod = 89;
```

```
}
```

```
}
```

PACKAGE : Access Protection

File path: rns\pucollege\NonSubClassDifferentPackage.java

```
package rns.pucollege;
```

```
import rns.engineering.CSE;
```

```
public class NonSubClassDifferentPackage {
```

```
    public void access()
```

```
{
```

```
    System.out.println("In Different Package non-sub-class");
```

```
    CSE cse = new CSE();
```

```
//cse.ipri=90; //CTE
```

```
//cse.ipro = 90; //CTE
```

```
//cse.inomod = 90; // CTE
```

```
cse.ipub = 90;
```

```
}
```

```
}
```

PACKAGE : Access Protection

File path: rns\pucollege\SubClassDifferentPackage..java

```
package rns.pucollege;
import rns.engineering.CSE;
public class SubClassDifferentPackage extends CSE
{
    public void acess()
    {
        System.out.println("In Different Package Sub class");
        //ipri=90; // CTE : not visible
        ipro = 900;
        ipub = 9000;
        //inomod = 89; // CTE : not visible
    }
}
```

PACKAGE : Access Protection

File path: src\Test.java

```
package Test;  
import rns.engineering.*;  
import rns.pucollege.*;  
class Test {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        SubClassSamePackage p = new SubClassSamePackage();  
        NonSubClassSamePackage q = new NonSubClassSamePackage();  
        SubClassDifferentPackage r = new SubClassDifferentPackage();  
        NonSubClassDifferentPackage s = new NonSubClassDifferentPackage();  
        p.access();          q.access();  
        r.access();          s.access();  
    }  
}
```

PACKAGE : Importing Packages

Packages are a way to categorize different classes from each other.

All built in java classes are stored in packages, there is not a single built in java class that is present in the unnamed-default package.

All standard classes are stored in some named package.

It is a tedious task to type long dot-separated package path names for each and every class present in the package to use it.

Ex: class test {

```
public static void main(String args[]) {  
    java.util.Scanner ip = new java.util.Scanner(java.lang.System.in);  
    java.lang.System.out.println("Afd");  
}
```

Java includes import statements to bring certain classes, or entire packages, into visibility.

PACKAGE : Importing Packages

Once imported, a class can be referred to directly, using only its name. The import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

G.F: **import pkg1[.pkg2].(classname|*);**

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, either an explicit class name or a star (*) must be specified, which indicates that the Java compiler must import the entire package or all the classes in the package.

Ex: `import java.util.Date;`
`import java.io.*;`

Multithreaded Programming

All modern operating systems support multitasking. There are two different types of multitasking.

1. Process-based
2. Thread-based.

Process-based multitasking

A *process* is a program under execution.

If a system allows more than one process to execute simultaneously, it is termed as *process-based multitasking*.

Ex: A process-based multitasking facilitates to run the Java compiler at the same time that you are using a text editor.

Processes are heavyweight tasks that require their own separate **address spaces**.

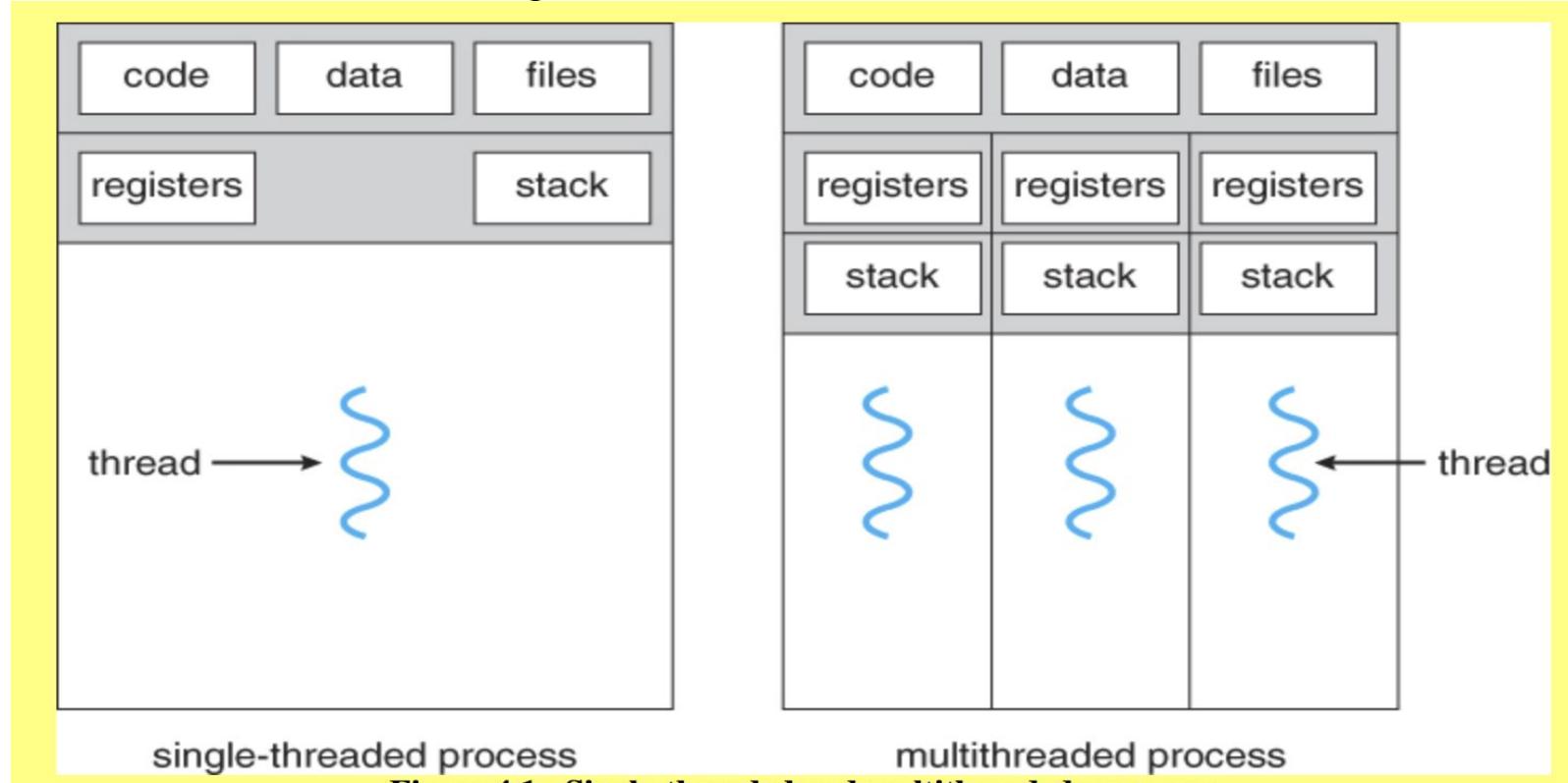
Interprocess communication is expensive and limited.

Context switching from one process to another is also costly. (Costly in terms of CPU time wastage)

Multithreaded Programming

Thread-based multitasking

Thread is the smallest unit of dispatchable/executable code.



Multithreaded Programming

Thread-based multitasking

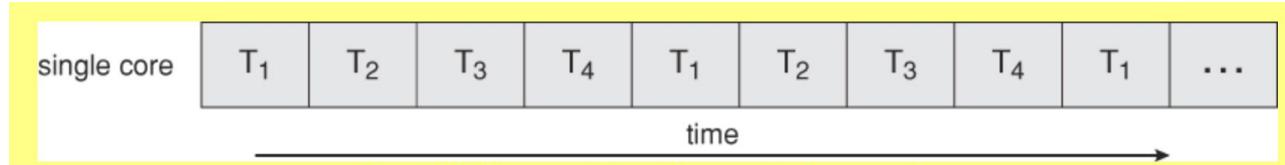
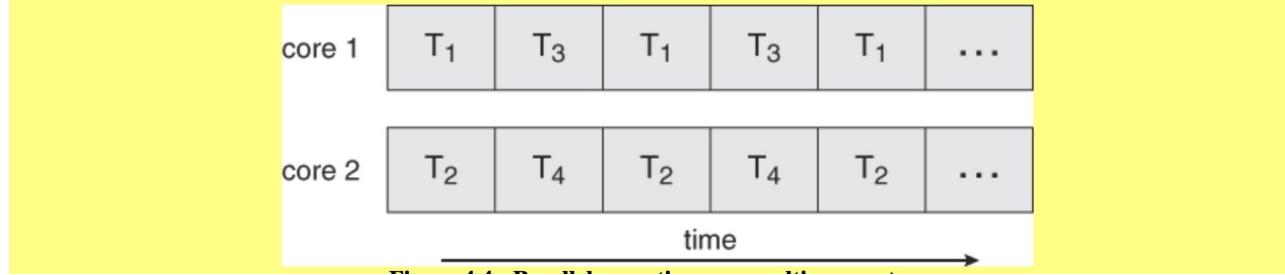


Figure 4.3 - Concurrent execution on a single-core system.



A single program can perform two or more tasks simultaneously.

Ex: A text editor can format text at the same time that it is printing. Printing and formatting will be done by two separate threads which belong to a single process.

“Multitasking threads” require less overhead than “multitasking processes”.

Multithreaded Programming

Thread-based multitasking

Threads are a lightweight process.

They share the same address space of a heavyweight process.

Inter Thread communication is inexpensive.

Context switching from one thread to the next is low cost.

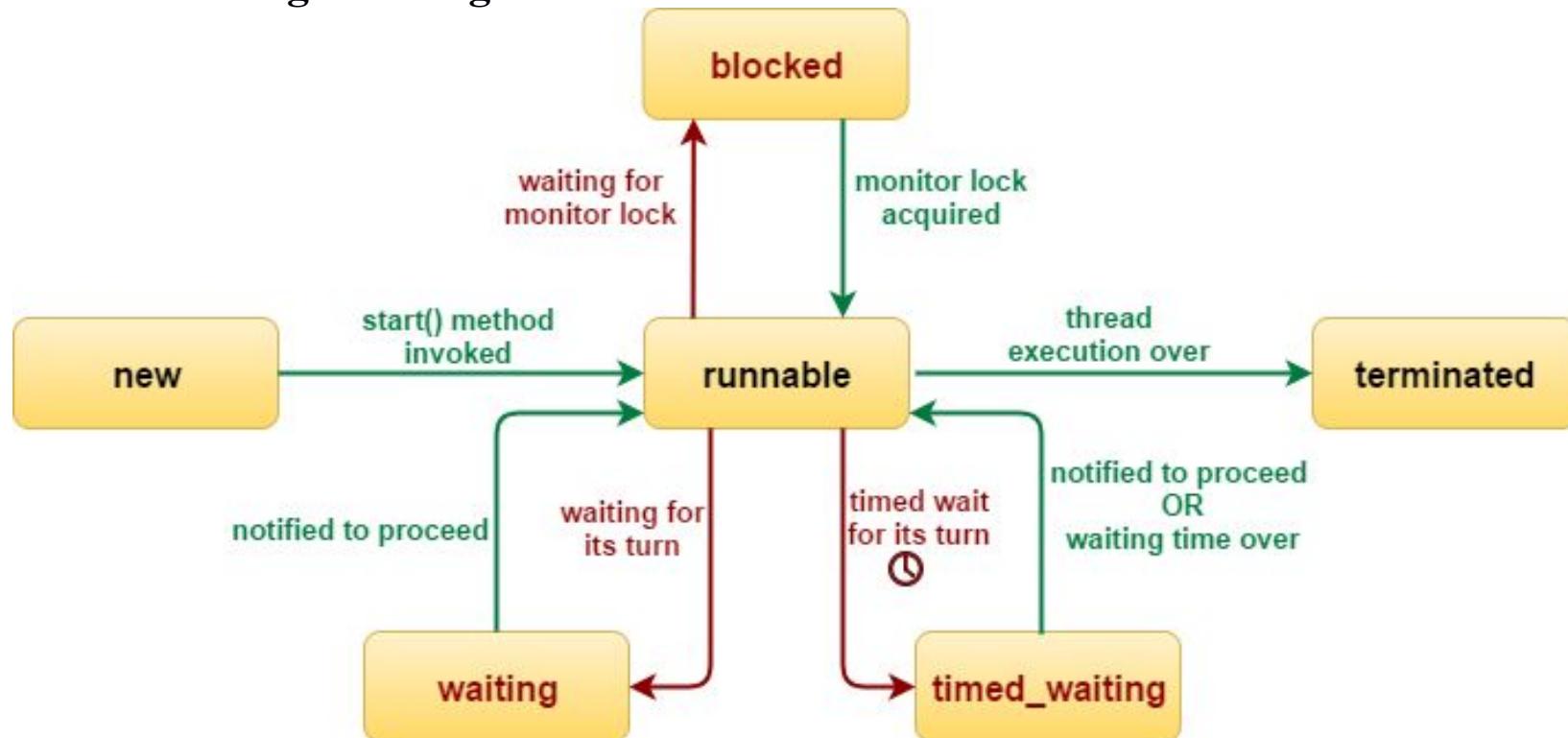
Java provides built-in support for *multithreaded programming*.

A multithreaded program contains two or more parts of the program that can run **concurrently**. Each part of such a program is called a *thread*, and each thread defines a separate path of execution.

Multithreading is a specialized form of multitasking.

While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.

Multithreaded Programming : Thread states



Multithreaded Programming : Thread class

Multithreading in java is built on the **Thread** class methods, and its associated **Runnable** interface.

To create a new thread, the program will either **extend Thread** or **implement the Runnable interface**.

The Thread class defines several methods that help manage threads.

getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>

All java programs by default have a **single thread termed as main**.

Multithreaded Programming : Thread class

When a Java program begins execution, one thread begins running immediately i.e main function.

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

```
class Thread //built-in class
{
    public static Thread currentThread();
    public final void setName(String threadName);
    public final String getName();
    public static void sleep(long milliseconds) throws InterruptedException
}
```

Multithreaded Programming : Thread class

currentThread() returns a reference to the thread in which it is called.

setName() is used to change the internal name of the thread, **threadName** specifies the name of the thread.

Ex: main() thread can be controlled through a Thread object.

A reference for main() has to be obtained using currentThread().

```
class CurrentThreadDemo {  
    public static void main(String args[]) {  
        Thread t = Thread.currentThread();  
        System.out.println("Current thread: " + t);  
  
        t.setName("My Thread"); // changing the name of the thread  
  
        System.out.println("After name change: " + t);  
    }  
}
```

Multithreaded Programming : Thread class

```
try {      // becz, Thread.sleep( ) throws InterruptedException
    for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.out.println("Main thread interrupted");
}
}
```

Output:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5 4 3 2 1

Multithreaded Programming : Thread class

Numerical values are displayed from 5 to 0 with a sleep interval of 1 second by calling sleep() method.

Exceptions will be generated automatically by Thread.sleep(), if another thread tries to interrupt a sleeping thread.

Thread information is displayed in the following manner.

the name of the thread, (which is main)

its priority, (default priority is 5)

and the name of its group. (main is the name of the group)

Multithreaded Programming : Creating a Thread using Runnable interface

Java has two ways of creating a thread:

- implement the Runnable interface.
- extend the Thread class.

Implementing Runnable

Generating a thread in java, using Runnable interface is a two step process.

1) To generate a thread, is to create a class that implements the Runnable interface.

```
interface Runnable  
{  
    void run();  
}
```

Multithreaded Programming : Creating a Thread using Runnable interface

run() is the entry point for a thread in the program.

Statements that are supposed to be executed in thread must be coded in run() method. Even function calls can also be made within the run () method.

2) An object of type Thread must be instantiated from within the class which implements Runnable interface.

Thread defines several constructors, one of them is

```
class Thread {  
    Thread( Runnable threadOb,      String threadName);  
    Thread (Runnable threadOb);  
}
```

threadOb: is an instance of a class that implements the Runnable interface.

threadName: name of new thread.

After a new thread object is created, it will not start executing until the **start()** method is called, which is declared within the Thread class.

```
class stack implements Runnable {
```

```
    @Override
```

```
    public run() { .....
```

```
Thread t;
```

```
public stack () {
```

```
    t = new Thread(this, "FT");
```

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

```
    .....
```

```
}
```

```
}
```

```
class test {
```

```
    Public static void main(String [] argos) {
```

```
        stack s = new stack();
```

```
}
```

```
}
```

Multithreaded Programming : Creating a Thread using Runnable interface

In essence, start() executes a call to run().

```
class Thread {  
    public void start() throws IllegalThreadStateException  
        //if the thread was already started.  
}
```

Thread process must start its execution only once and not many times.

Ex: class NewThread implements Runnable {

```
    Thread t;
```

*//Since, start() and run() is present only in Thread class, an instance of Thread must be
//created here.*

Multithreaded Programming : Creating a Thread using Runnable interface

```
NewThread()
{
    // Thread object is created, which is used to call start( ) method.
    t = new Thread(this, "Demo Thread");

    /*
        "this" is a reference of type NewThread which implements Runnable interface.
        Hence, it is passed as the first parameter.
    */
    System.out.println("Child thread: " + t);

    // Create a new, second thread. Becz main is considered as the 1st thread
    t.start(); // Begins thread execution
}
```

Multithreaded Programming : Creating a Thread using Runnable interface

```
// This is the entry point for thread.  
@Override  
public void run()  
{  
    for(int i = 5; i > 0; i--)  
    {  
        System.out.println("Child Thread: " + i);  
        try {  
            Thread.sleep(500);  
        }  
        catch (InterruptedException e)  
        { System.out.println("Child interrupted."); }  
    }  
    System.out.println("Exiting child thread.");  
}  
}/End of class NewThread
```

Multithreaded Programming : Creating a Thread using Runnable interface

```
class ThreadDemo {  
    public static void main(String args[]) {  
        new NewThread(); // create a new thread  
  
        //main thread resumes its execution  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Main Thread: " + i);  
                Thread.sleep(1000);  
            } }  
        catch (InterruptedException e)  
        {System.out.println("Main thread interrupted.");}  
  
        System.out.println("Main thread exiting.");  
    }  
}//End of class ThreadDemo
```

Multithreaded Programming : Creating a Thread using Runnable interface

a new Thread object is created by the following statement inside constructor

```
t = new Thread(this, "Demo Thread");
```

Passing “**this**” as the first argument indicates that the new thread has to be called on the object referenced by this.

Next, start() is called, which starts the thread execution beginning from the run() method.

After calling start(), NewThread’s constructor returns to main(), then main thread resumes its execution, and begins executing for loop.

Both threads (main & Demo Thread) continue executing, sharing the CPU, until their statements get executed.

Multithreaded Programming : Creating a Thread using Runnable interface

Output may vary based on processor speed and task load

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

Multithreaded Programming : Creating a Thread using Runnable interface

In a multithreaded program, often the main thread must be the last thread to finish running.
(automatically the main thread will not conclude the execution at last, but forcibly it will be made to conclude the execution after all the child threads completes its execution.

In case if any resources that are acquired by the main thread and in turn they are used by the child threads, then the main thread has to wait until the child thread concludes its execution.

Ex: Scanner instance can be instantiated in main thread and in turn it can be used by child thread. Only after the usage of child thread is over, the scanner instance can be closed in the main thread. Closing of scanner instance in child thread, will avoid further usage of the same in main thread.

Habit, of acquiring resources will be done in main because it stays for a longer period of time, since the execution starting point is main and the usual exit of control is from main.)

The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Multithreaded Programming : Creating a Thread using Extending Thread

Another way to create a thread is to create an user defined class that extends the “Thread” class, and then to create an instance of it.

Extending class must override the run() method, which is the entry point for the new thread.

It must also call start() to begin execution of the new thread.

Ex: **class** NewThread **extends** Thread {

```
NewThread() {  
    // Calling Thread class constructor      public Thread(String threadName)  
    super("Demo Thread");  
    System.out.println("Child thread: " + this);  
    start(); // Start the thread, which will in turn call run( ) method.  
}
```

Multithreaded Programming : Creating a Thread using Extending Thread

```
// This is the entry point for the second thread. Overriding run method
```

```
@Override
```

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

```
try {
```

```
for(int i = 5; i > 0; i--) {
```

```
System.out.println("Child Thread: " + i);
```

```
Thread.sleep(500);
```

```
}
```

```
}
```

```
catch (InterruptedException e)
```

```
{ System.out.println("Child interrupted."); }
```

```
System.out.println("Exiting child thread.");
```

```
}
```

Multithreaded Programming : Creating a Thread using Extending Thread

```
class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread, unreferenced instance
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            {System.out.println("Main thread interrupted.");}
            System.out.println("Main thread exiting.");
        } } //End of class ExtendThread
```

Output will be the same as before.

Multithreaded Programming : Creating a Thread using Extending Thread

Whether to use *implements (Runnable)* or *extends (Thread)* to create a thread

Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run() to create a thread. This is, of course, the same method required by Runnable.

Thread class must be extended only when they are being enhanced or modified in some way.

If none of the other thread methods are overridden, other than run() method, then it is best implement Runnable.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>

Consider

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Multithreaded Programming : Creating a Multiple Threads

```
//Create multiple threads.  
class NewThread implements Runnable  
{  
  
    String name; // name of thread  
    Thread t;  
  
    NewThread(String threadname) {  
        name = threadname;  
        t = new Thread(this, name);  
        System.out.println("New thread: " + t);  
        t.start(); // Start the thread  
    }  
}
```

Multithreaded Programming : Creating a Multiple Threads

```
@Override  
public void run()  
{  
    try {  
        for(int i = 2; i > 0; i--) {  
            System.out.println(t.getName() + ": " + i);  
            Thread.sleep(1000);  
        }  
    }  
    catch (InterruptedException e)  
    {System.out.println(name + "Interrupted");}  
    System.out.println(name + " exiting.");  
}  
}
```

Multithreaded Programming : Creating a Multiple Threads

```
class Test {  
    public static void main(String args[]) {  
        new NewThread("One");  
        new NewThread("Two");  
        new NewThread("Three");  
        try {  
            // wait for other threads to end  
            Thread.sleep(10000);  
            //main sleeps for 10 seconds, to ensure that the main thread finishes last.  
        }  
        catch (InterruptedException e) { System.out.println("Main thread Interrupted"); }  
        System.out.println("Main thread exiting.");  
    }  
}
```

Multithreaded Programming : Creating a Multiple Threads

Output:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

One: 2

Two: 2

New thread: Thread[Three,5,main]

Three: 2

Two: 1

Three: 1

One: 1

Two exiting.

One exiting.

Three exiting.

Main thread exiting.

Multithreaded Programming : `isAlive()` and `join()`

Making the `main()` thread to sleep for a longer duration of time, and allowing child processes to complete its execution is not a competent solution. (Asynchronous waiting)

Two ways exist to determine if a thread is alive or not.

1. `isAlive()`
2. `join()` (Synchronous waiting)

```
class Thread {  
    final boolean isAlive()  
    final void join() throws InterruptedException  
}
```

`isAlive()` method returns true if the thread upon which it is called is still running, otherwise false.

`join()` method, puts the thread from which it is called, on wait,
until the thread on which it has called finishes its execution.

Multithreaded Programming : isAlive() and join()

If thread (which has invoked join())is interrupted then it will throw InterruptedException.

Ex: using join() method

```
class NewThread extends Thread
{
    NewThread(String n)
    {
        this.setName(n);
        System.out.println("Child thread: " + this);
        start();
    }
}
```

Multithreaded Programming : isAlive() and join()

```
@Override
```

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

```
try {
```

```
for(int i = 3; i > 0; i--) {
```

```
    System.out.println( getName() + " Thread: " + i);
```

```
    Thread.sleep(500); }
```

```
}
```

```
catch (InterruptedException e) { System.out.println("Child interrupted."); }
```

```
System.out.println("Exiting "+getName()+" child thread.");
```

```
}
```

Multithreaded Programming : isAlive() and join()

```
class Test {  
    public static void main(String args[]) {  
        NewThread nt = new NewThread("First");  
        NewThread nt1 = new NewThread("Second");  
        try {  
            nt.join();  
            nt1.join();  
/* Since nt.join( ) is called from main thread, main thread suspends execution until thread  
named “First” completes its execution. Once nt thread completes its execution, main( ) thread  
resumes execution */  
        }  
        catch (InterruptedException e) {System.out.println("Main thread interrupted.");}  
        System.out.println("Main thread exiting.");  
    } }
```

Multithreaded Programming : isAlive() and join()

Output:

Child thread: Thread[First,5,main]

Child thread: Thread[Second,5,main]

First Thread: 3

Second Thread: 3

First Thread: 2

Second Thread: 2

First Thread: 1

Second Thread: 1

Exiting First child thread.

Exiting Second child thread.

Main thread exiting.

Multithreaded Programming : Thread Priorities

<https://www.informit.com/articles/article.aspx?p=26326&seqNum=5>

Priorities Determine which thread gets CPU allocated and gets executed first.

Thread priority values in java range from 1 to 10, 1 being the least priority and 10 being the highest.

Higher the thread priority, larger is the chance for a process of getting executed first.

Ex:

Two threads are ready to run.

First thread priority is 5 and begins execution.

Second thread priority, consider to be 10 is ready for execution,

Then, First thread may suspend its execution relieving the control to Second thread.

A thread's priority is also used to decide when to switch from one running thread to the next termed as "**context switch**".

Multithreaded Programming : Thread Priorities

`setPriority()` method, is used to set priority for a thread

```
class Thread {  
    final void setPriority(int level)    level specifies the priority  
}  
}
```

level value must be in the range MIN_PRIORITY and MAX_PRIORITY, these values are 1 and 10, respectively.

Default priority of a thread is NORM_PRIORITY, which is equal to 5.

MIN_PRIORITY, MAX_PRIORITY & NORM_PRIORITY's are defined as **static final variables** within Thread.

Current priority of a thread can be obtained by calling `getPriority()` method of Thread,

```
class Thread {  
    final int getPriority()    }  
}
```

Multithreaded Programming : Thread Priorities

Theoretically higher-priority threads get more CPU time than lower-priority threads.

Ex://Demonstrate thread priorities.

```
class clicker implements Runnable {  
    long click = 0;  
    Thread t;  
    private volatile boolean running = true;  
  
    public clicker(int p) {  
        t = new Thread(this);  
        t.setPriority(p);  
    }  
  
    public void run() {  
        while (running)  
            click++;  
    }  
}
```

Multithreaded Programming : Thread Priorities

```
public void stop()    { running = false; }

public void start()   { t.start(); }

class test {
    public static void main(String args[])
    {
        clicker hi = new clicker(Thread.NORM_PRIORITY + 4);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

        hi.start();
        lo.start();
    }
}
```

Multithreaded Programming : Thread Priorities

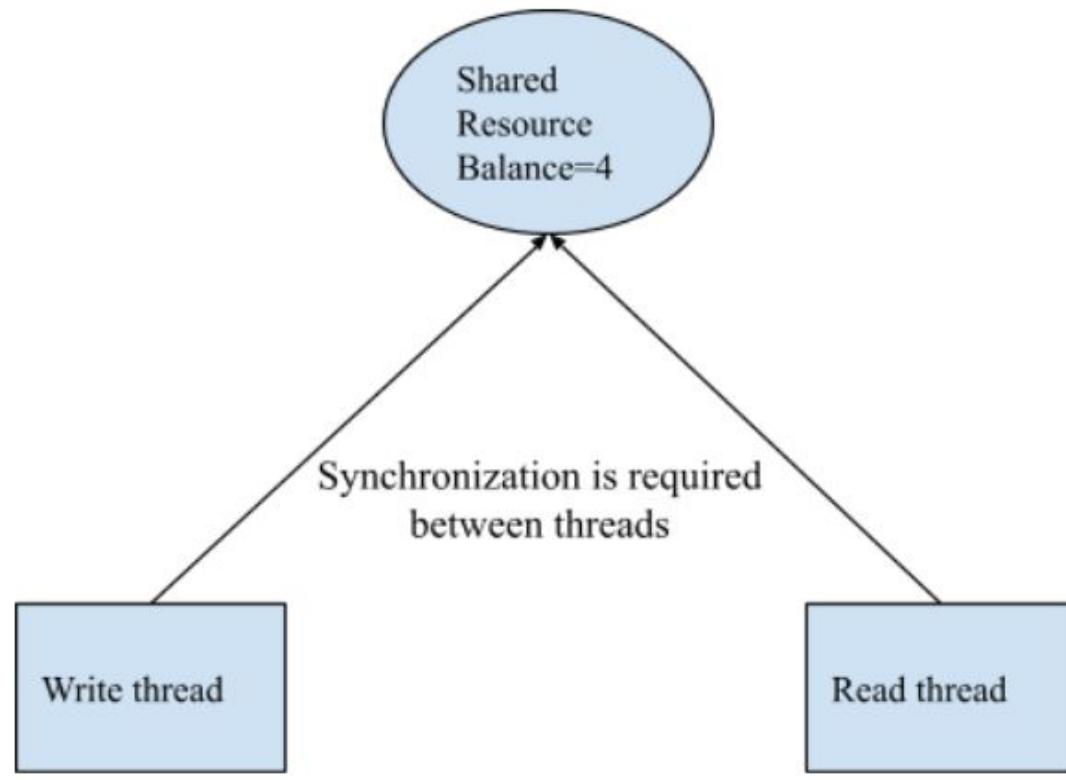
```
try { Thread.sleep(1); }
catch (InterruptedException e)
{ System.out.println("Main thread interrupted."); }
```

```
lo.stop();
hi.stop();
try {
    hi.t.join();    lo.t.join();
}
catch (InterruptedException e)
{ System.out.println("InterruptedException caught"); }
```

```
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);    }    }
```

Multithreaded Programming : Synchronization

Two threads can access a shared resource, if two threads access a resource **simultaneously** error may ensue.



Multithreaded Programming : Synchronization

Synchronization is a mechanism to achieve exclusive access to shared resources by more than one thread. Key to synchronization is the concept of monitor.

/*

Monitors

Abstract Data Type for handling/defining shared resources

Comprises:

Shared Private Data

The resource Cannot be accessed from outside

Procedures that operate on the data

Gateway to the resource

Can only act on data local to the monitor

Synchronization primitives

Among threads that access the procedures

Multithreaded Programming : Synchronization

Monitors guarantee mutual exclusion

*Only one thread can execute a monitor procedure at any time.
“in the monitor”*

*If second thread invokes a monitor procedure at that time
It will be blocked and put into wait for entry to the monitor
In Need of a wait queue*

Multithreaded Programming : Synchronization

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . .) {
        . . .
    }

    procedure P2(. . .) {
        . . .
    }

    :
    procedure PN(. . .) {
        . . .
    }

    initialization_code(. . .) {
        . . .
    }
}
```

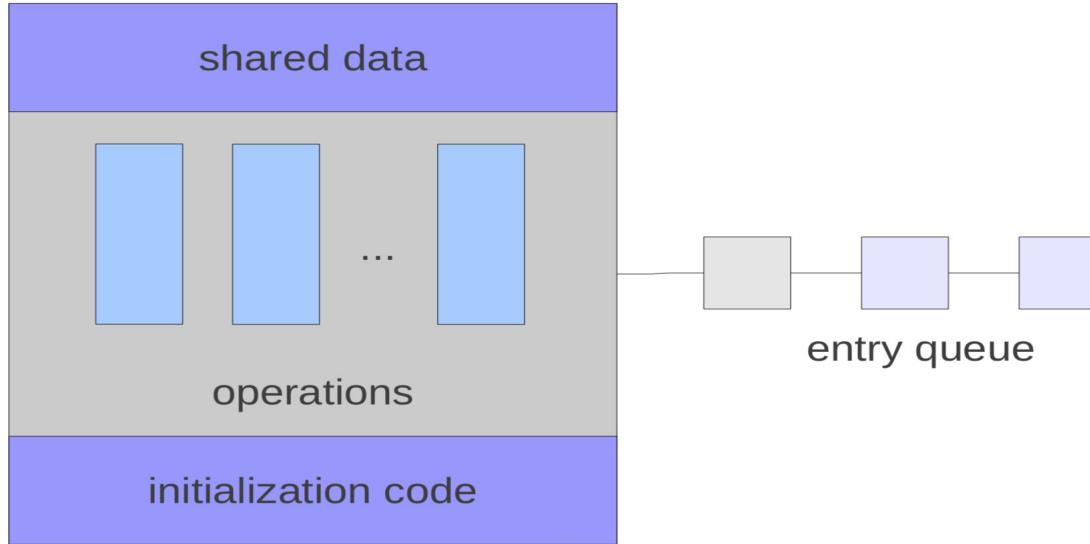
For example:

```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . .
    }

    any_t * pop() {
        . . .
    }

    initialization_code() {
        . . .
    }
}
```

Multithreaded Programming : Synchronization

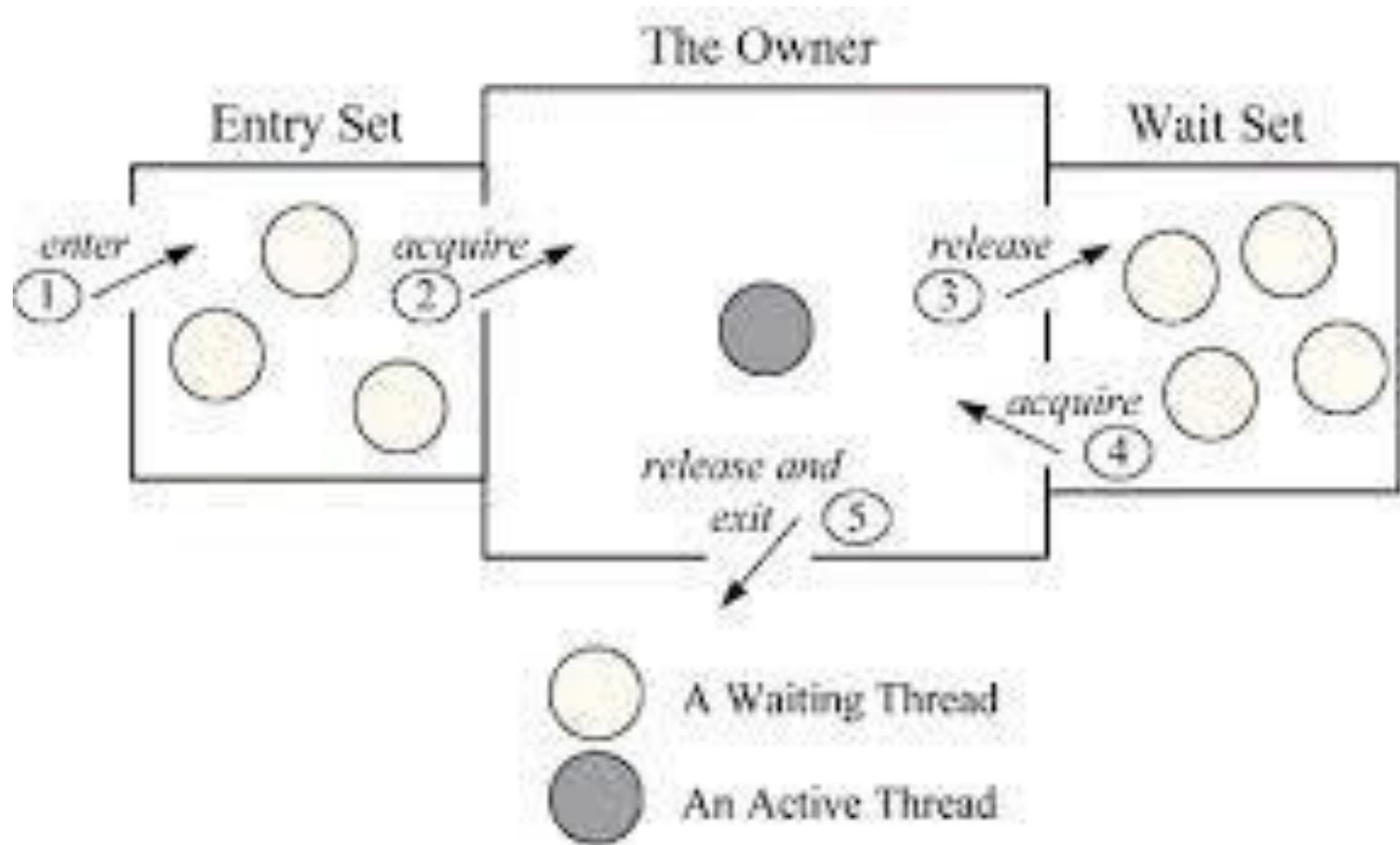


A monitor is an object that is used as a mutually exclusive lock, or mutex.

Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

*All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. */*

Multithreaded Programming : Synchronization



Multithreaded Programming : Synchronization

Ex: Expected output of the below program is

[Hello]

[Synchronized]

[World]

Using 3 different threads, **without any synchronization** between them.

Pgm Name : NotSynch.java

```
class Callme {  
  
    void call(String msg) {  
        System.out.print("[ " + msg);  
        try { Thread.sleep(1000); }  
        catch(InterruptedException e)  
        {System.out.println("Interrupted");}  
  
        System.out.println("]");  
    }    }  

```

Multithreaded Programming : Synchronization

```
class Caller implements Runnable {
```

```
    String msg;      // required to hold on to messages
```

```
    Callme target; /* target is the common object for three threads
```

Method in this will display the expected output.

*Statements in this method must be executed in a synchronized manner,
or non-overlapping manner. */*

```
    Thread t;      // object of Thread class required to call start() and run() method.
```

```
public Caller(Callme targ, String s) {
```

```
    target = targ;
```

```
    msg = s;
```

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

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

```
}
```

Multithreaded Programming : Synchronization

```
public void run() {  
    target.call(msg);  
}  
}
```

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

```
        Callme target = new Callme(); // ***** COMMON DATA SOURCE
```

```
        Caller ob1 = new Caller(target, "Hello");
```

```
        Caller ob2 = new Caller(target, "Synchronized");
```

```
        Caller ob3 = new Caller(target, "World");
```

```
/* all threads are simultaneously trying to print the message onto screen which creates race  
condition*/
```

Multithreaded Programming : Synchronization

// making the main thread wait for other threads to end its processes.

```
try
{
    ob1.t.join();        ob2.t.join();        ob3.t.join();
}
catch(InterruptedException e)
{System.out.println("Interrupted");}
}
```

Output: *need not be the same output, one of the ways how output will be obtained.*

```
[Hello[Syncronized[World]
]
]
```

Multithreaded Programming : Synchronization

by calling sleep(), the call() method allows execution to switch to another thread, which results in the mixed-up output of the three message strings.

In this program, all three threads are calling **the same method, on the same object**, at the same time, termed as **race condition**, because the three threads are competing with each other to complete the process.

Using Synchronized Methods

Instances in Java have their own implicit monitor associated with them, making it easy to achieve synchronization.

(Instances are the data source, which hold on to information. Synchronization or mutual exclusive access, is required on data sources from methods.)

“**synchronized**” is the keyword used in java to achieve synchronization between several processes accessing the same data source.

Multithreaded Programming : Synchronization

When **synchronized** block is used, internally Java uses a monitor also known as intrinsic lock, to provide synchronization.

These monitors are bound to an object, thus all synchronized blocks of an object can have only one thread executing/accessing them at any point of time.

While a thread is inside a synchronized block, making an attempt from other threads to access **the same instance** will be put into the wait state.

To exit the monitor and to give up control of the object to the next waiting thread, control returns from the synchronized method.

In the program “NotSynch.java”, access to call() method must be **serialized or synchronized**, by restricting only one thread at a time. To do this, precede call()’s definition with the keyword **synchronized**

Multithreaded Programming : Synchronization

```
class Callme
{
    synchronized void call(String msg) // synchronized methods
    {
        ...
    }
}
```

Only one thread will be allowed to execute/access the synchronized method.

If a thread is executing a synchronized method, then no other thread can enter it until it stops executing that method.

(Even if the 1st thread that has begun execution and has entered into sleep mode, pre-emption of it is not possible, until the thread itself will relinquish the access)

Multithreaded Programming : The synchronized Statement

Synchronized blocks/methods will not work in all scenarios.

Considering the method belongs to some other class, which is not editable or there is no access to source code, then attaching **synchronized** keyword for the method is not possible.

Solution to this is a synchronized block.

G.F of synchronized statement:

```
synchronized(object) {
    // statements to be synchronized
}
```

Here, object is a reference to the object being synchronized.

A synchronized block ensures that a call to a method (which is a member of) object occurs only after the current thread has successfully entered the object's monitor.

Multithreaded Programming : The synchronized Statement

Ex: Using Synchronized statement

class Callme

{

void call(String msg)

{

 System.**out**.print("[" + msg);

try

 { Thread.sleep(1000); }

catch (InterruptedException e)

 { System.**out**.println("Interrupted"); }

 System.**out**.println("]");

}

}

Multithreaded Programming : The synchronized Statement

class Caller **implements** Runnable

{

 String msg;

 Callme target;

 Thread t;

public Caller(Callme targ, String s)

{

 target = targ;

 msg = s;

 t = **new** Thread(this);

 t.start();

}

Multithreaded Programming : The synchronized Statement

```
public void run()
{
    synchronized(target) // synchronized block
    {
        target.call(msg);
    }
}
```

Multithreaded Programming : The synchronized Statement

```
class Test {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        }  
        catch(InterruptedException e)  
        {System.out.println("Interrupted");}  
    } }
```

Multithreaded Programming : Interthread/Interprocess Communication

Another way to achieve synchronization in java is by using interprocess communication.

Ex: Consider queuing problem, where one thread is producing some data and another is consuming it.

Word document can be considered as a producer and printer can be considered as consumer, where in the document will be printed.

Further, assume that the document is ranging in terms of MB size and many pages have to be printed, but the buffer associated with the printer is capable of holding on to only a small amount of information (KB).

The producer has to wait until the consumer is finished consuming before it generates(copies) more data(Considering the buffer which is used between them is full).

Multithreaded Programming : Interthread/Interprocess Communication

In a polling system, the consumer would waste many CPU cycles while it was waiting for the producer to produce. Once the producer has finished, it would start polling, just to check whether the consumer has consumed the data, wasting more CPU cycles, and so on. Clearly, this situation is time consuming.

To avoid polling, Java has an interprocess communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final methods in Object**.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

These methods are declared within Object, as shown here:

```
class Object {  
    final void wait( ) throws InterruptedException  
        //wait( ) method is overloaded  
    final void notify()  
    final void notifyAll()  
}
```

Multithreaded Programming : Interthread/Interprocess Communication

All three methods can be called only from within a synchronized context.

wait()

Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object. Current thread must own this object's monitor.

notify()

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.

notifyAll() wakes up all the threads that called **wait()** on the same object.

Alone **wait()** and **notify()** will not achieve exact synchronization in IPC.

Multithreaded Programming : Interthread/Interprocess Communication

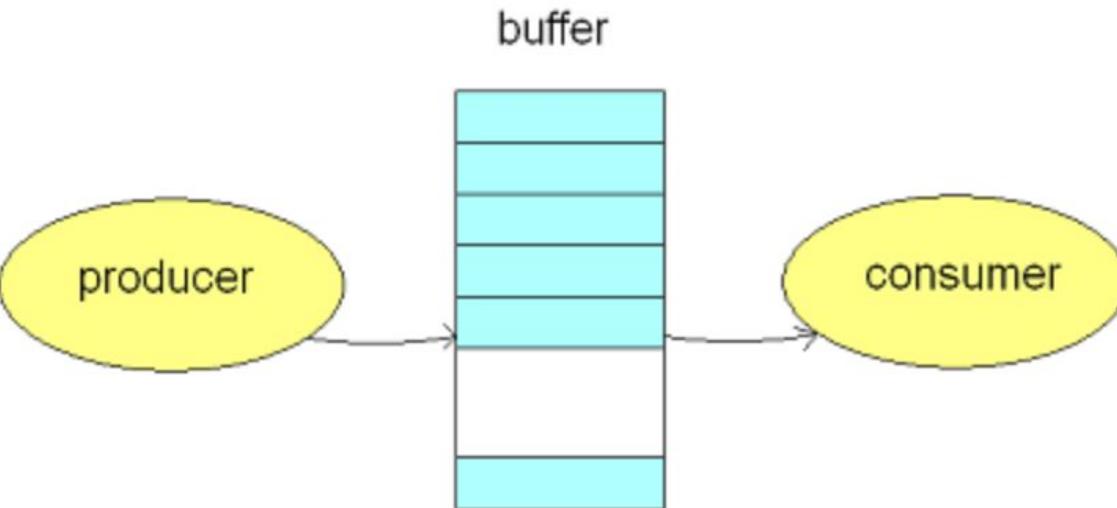
Before calling `wait()` or `notify()` a condition has to be checked in a loop to achieve exact synchronization in IPC. The condition will be based on the logical necessity of the application to be solved.

Ex: Simulating a producer and a consumer thread. Assumed that there is only one buffer between these two to hold on to only 1 information.

It is assumed that based on the capacity of the buffer, the producer can produce only 1 information and it has to wait until the consumer consumes this information before producing the next information.

Similarly, consumer can consume only one information, even if it has the capacity of consuming or using more than 1 information, because buffer size is limited to 1.

Multithreaded Programming : Interthread/Interprocess Communication



Multithreaded Programming : Interthread/Interprocess Communication

Ex: Following program consists of four classes:

Q, the queue that is being synchronized;

Producer, the threaded object that is producing queue entries;

Consumer, the threaded object that is consuming queue entries; and

PC, class that creates the single **Q**, **Producer**, and **Consumer**.

// An incorrect implementation of a producer and consumer without proper IPC.

```
class Q {  
    int n;  
/*  "n" is a common variable between producer and consumer thread.  
    "n" must be perceived as a buffer.  
    Producer thread will fill a value to n.  
    Consumer process will pick a value from n.
```

get() method will be called by consumer thread to get the value stored in n.
put() method will be called by producer thread to put a value into n. */

Multithreaded Programming : Interthread/Interprocess Communication

```
synchronized void get() // used by the consumer process
{
    System.out.println("Got: " + n);
}
```

```
synchronized void put(int n) //used by the producer process
{
    this.n = n;
    System.out.println("Put: " + n);
}
```

Multithreaded Programming : Interthread/Interprocess Communication

//Producer thread

```
class Producer implements Runnable {
```

```
    Q q;
```

```
    Producer(Q q) {
```

```
        this.q = q;
```

```
        new Thread(this, "Producer").start();
```

```
}
```

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

```
        int i = 0;
```

```
        while(true)
```

```
            q.put(i++);
```

```
} }
```

Multithreaded Programming : Interthread/Interprocess Communication

//Consumer thread

```
class Consumer implements Runnable {
```

```
    Q q;
```

```
    Consumer(Q q) {
```

```
        this.q = q;
```

```
        new Thread(this, "Consumer").start();
```

```
}
```

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

```
        while(true)
```

```
            q.get();
```

```
}
```

```
}
```

Multithreaded Programming : Interthread/Interprocess Communication

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

Output:

Put:1
Got:1
Got:1
Got:1
Got:1

Multithreaded Programming : Interthread/Interprocess Communication

Just achieving synchronization between threads is not sufficient for producer-consumer problem, in addition to this, two conditions have to be checked for p-c problem to work in a competent manner.

1. First condition is w.r.t to consumer, whether buffer is filled before consumer consumes from it and
2. Second condition with respect to the producer will be that before the producer fills the buffer with information it is necessary that the buffer is empty.

Multithreaded Programming : Interthread/Interprocess Communication

// A correct implementation of a producer and consumer.

```
class Q
{
    int n;
    volatile boolean valueSet = false;
```

/*“valueSet” is a common variable used between producer and consumer process to know whether the buffer is full or not.

```
*/
```

Multithreaded Programming : Interthread/Interprocess Communication

```
synchronized void get()
{
    while(!valueSet)
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
            System.out.println("InterruptedException caught");
        }

    System.out.println("Got: " + n);
    valueSet = false;
    notify();
}
```

Multithreaded Programming : Interthread/Interprocess Communication

```
synchronized void put(int n)
```

```
{
```

```
    while(valueSet)
```

```
        try {
```

```
            wait();
```

```
}
```

```
    catch(InterruptedException e)
```

```
    { System.out.println("InterruptedException caught"); }
```

```
    this.n = n;
```

```
    valueSet = true;
```

```
    System.out.println("Put: " + n);
```

```
    notify();
```

```
}
```

Multithreaded Programming : Interthread/Interprocess Communication

```
class producer implements Runnable  
{  
    .....
```

Bounded buffer problem is nothing but producer-consumer problem

MODULE 5

The Delegation Event Model

Events are handled in Java using a *delegation event model*.

Definition : EVENT

An *event* in Java is an object that is created, when changes are experienced in a graphical user interface.

Changes such as

- a user clicks on a button,
- clicks on a combo box, or
- types characters into a text field, etc.,

then an event triggers, creating the relevant event object.

This behavior is part of Java's Event Handling mechanism

(other properties of Java are

- Java is capable of handling Exceptions
- Java is capable of handling Garbage Collection
- etc.,)

Event handling is integral to the creation of swings and AWT type of GUI-based programs.

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

Words to ponder in DELEGATION-EVENT MODEL: EVENT, EVENT SOURCE, EVENT LISTENER

Delegation Event Model:

"An event source (GUI components) generates an event and sends it to one or more event listeners.

Listener waits until it receives an event.

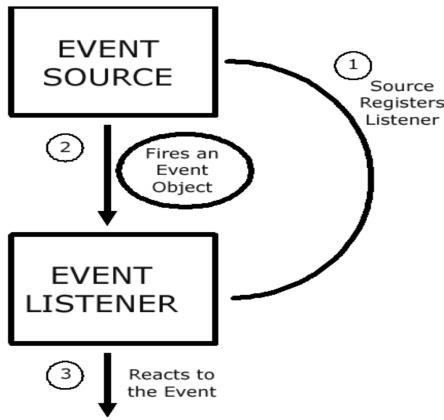
Once an event is received, the listener processes the event and then returns the result."

Advantage of this “DELEGATION-EVENT MODEL” is that the “application logic” that “processes events” is clearly separated from the “user interface logic” that “generates those events”.

A user interface element (GUI) is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, **listeners must register with a source** in order to receive an **event notification**.

Event notifications are sent only to listeners that have registered with the event source.
(callback interfaces)



Events

In the delegation model, an *event* is represented in the form of an object that describes a state change in a source (specifically a GUI application).

Ex: After typing google.com in a web browser, text box of google web page will be empty. (Event is yet to be generated).

As soon as the user types the keywords and press enter, an object containing all the necessary information, regarding the keyword search, including the keywords will be generated and passed to the nearest server.

Event is generated as a consequence of an user interacting with the elements in a GU interface.

Events may also occur that are **not directly caused by interactions with a user interface**.

Ex: an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Event Sources

Event source in a programming perspective is an object that holds all the necessary information about an event. This occurs when the internal state of that object changes in some way.

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:

```
public void add<Type>Listener(<Type>Listener el)
```

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

Ex: The method that registers a keyboard event listener is called **addKeyListener()**.

The method that registers a mouse motion listener is called **addMouseMotionListener()**.

When an event occurs, all registered listeners are notified and receive a copy of the “event object”. This is known as **multicasting** the event.

Notifications are sent only to listeners that register to receive them.

Some sources will allow only one listener to register.

GF:

```
public void add<Type>Listener(<Type>Listener el) throws  
    java.util.TooManyListenersException
```

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

When such an event occurs, the registered listener is notified. This is known as ***unicasting*** the event.

/*

A source will also provide a method that allows a listener to unregister an interest in a specific type of event.

GF: public void remove<Type>Listener(<Type>Listener el)

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

For example, to remove a keyboard listener, you would call ***removeKeyListener()***.

*/

Methods that add or remove listeners are provided by the source that generates events.

Below table lists some of the **user interface components/graphical user interface elements that can generate the 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; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-2 Event Source Examples

Event Classes

When an event occurs, all its information will be encapsulated within an instance of Event class type.

Classes that represent events are at the core of Java's event handling mechanism.

Most widely used events are defined by the AWT(Abstract Window Toolkit) and Swing.

```
/*
```

*At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events.*

EventObject class contains these methods:

```
Object getSource(); //returns the source of the event.  
String toString(); //returns the string equivalent of the event.  
int getID(); //returns event type
```

Class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.

AWTEvent is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

```
*/
```

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

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-1 Main Event Classes in **java.awt.event**

An instance of above classes will be generated by event source automatically and passed to event listeners.

Programmers need not worry about generating instances of these classes, it will be done automatically by JVM.

The Programmer's job will be to register a method as a listener and provide the necessary background logical actions for the event.

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**, and **SHIFT_MASK**.

In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

Constructors of ActionEvent class // **used only by event source**

```
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)
```

src : is a reference to the object that generated this event.

type : type of the event is specified by type.

modifier : indicates which modifier keys such as ALT, CTRL, META and/or SHIFT were pressed when the event was generated.

when: parameter specifies when the event occurred.

String getActionCommand() //used by event listeners

returns Command name for the invoking ActionEvent object

Ex: when a button is pressed, an action event is generated that has a command name equal to the label on that button.

int getModifiers() //used by event listeners

returns a value that indicates which modifier keys (ALT, CTRL,META, and/or SHIFT) were pressed when the event was generated.

long getWhen() //used by event listeners

returns the time at which the event took place. This is called the event's *timestamp*.

The AdjustmentEvent Class

The ContainerEvent Class

The InputEvent Class

The KeyEvent Class

The MouseWheelEvent Class

The WindowEvent Class

The ComponentEvent Class

The FocusEvent Class

The ItemEvent Class

The MouseEvent Class

The TextEvent Class

Event Listeners

/*

Definition : EVENT LISTENER

An *event listener* is a **procedure** or **function** in a computer program that waits for an *event* to occur. The listener is programmed to react to an *input* or *signal* by calling the event's handler.



*/

An *Event listener* is an object that is notified when an event occurs.

(Remember even *Event source* is an object, which generates events)

It has two major requirements.

First, it must have been registered with one or more sources to receive notifications about specific types of events.

Second, it must implement methods to process these notifications/requests, which will logically modify the received data from the event source.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. Ex: **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Listener Interfaces

Delegation event model has two parts: **source and listener**.

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Table lists commonly used listener interfaces and provides a brief description of the methods that they define.

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

TABLE 22-3 Commonly Used Event Listener Interfaces

ActionListener Interface

defines the **actionPerformed()** method that is invoked when an action event occurs.

```
void actionPerformed (ActionEvent ae)
```

AdjustmentListener Interface

defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs.

```
void adjustmentValueChanged (AdjustmentEvent ae)
```

.....

KeyListener Interface

defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively.

The **keyTyped()** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released.

If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

```
void keyPressed (KeyEvent ke)
void keyReleased (KeyEvent ke)
void keyTyped (KeyEvent ke)
```

MouseListener Interface

defines five methods.

If the mouse is pressed and released at the same point, `mouseClicked()` is invoked.
When the mouse enters a component, the `mouseEntered()` method is called.
When it leaves, `mouseExited()` is called.
The `mousePressed()` and `mouseReleased()` methods are invoked when the mouse is pressed and released, respectively.

```
void mouseClicked (MouseEvent me)
void mouseEntered (MouseEvent me)
void mouseExited (MouseEvent me)
void mousePressed (MouseEvent me)
void mouseReleased (MouseEvent me)
```

MouseListener Interface

This interface defines two methods.

```
void mouseDragged (MouseEvent me)
void mouseMoved (MouseEvent me)
```

The `mouseDragged()` method is called multiple times as the mouse is dragged.

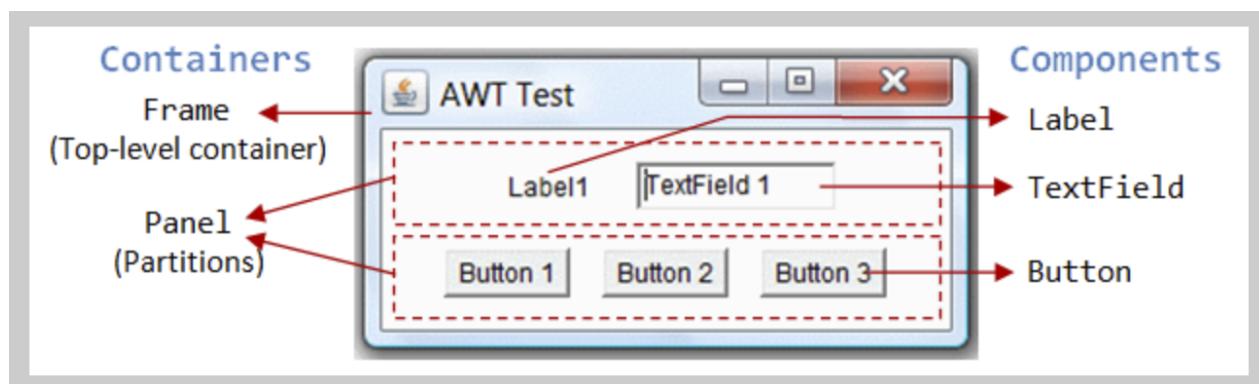
The `mouseMoved()` method is called multiple times as the mouse is moved.

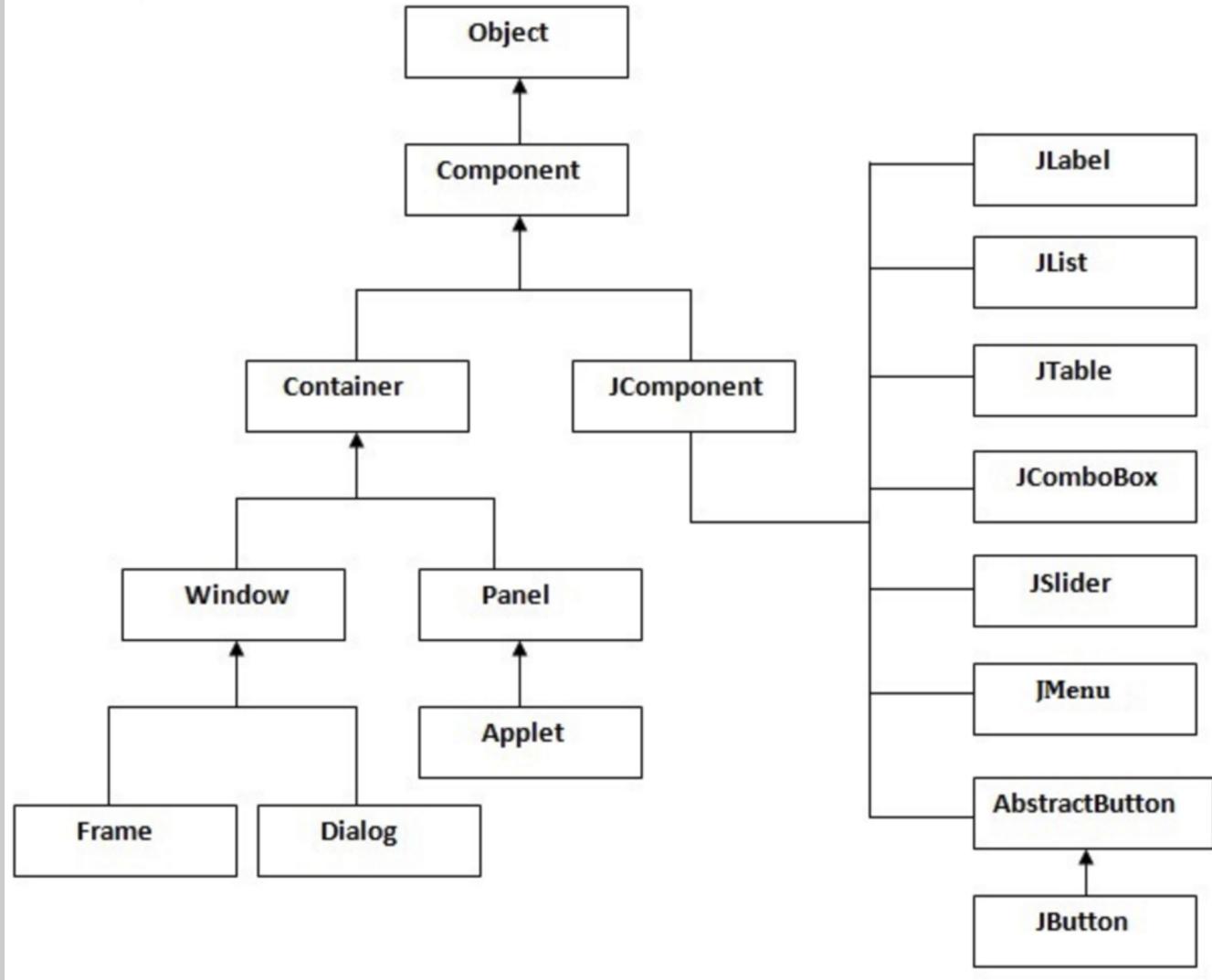
/* POINTS TO PONDER

A single GUI component like button, or text box or label will be part of a container like Frame

GUI components like buttons, textbox or labels are Event sources.

But since, these components are stored or added to a container **Frame acts as both an event source and an event listener.*/**





Ex: HANDLING MOUSE EVENTS

```
import java.awt.*; //1
import java.awt.event.*; //2
//GUI is both event source and event listener
public class GUI extends Frame implements MouseListener, MouseMotionListener
{
    Label l; //3
    Label l1,l2;

    int mx=0,my=0;

    GUI(){
        addMouseListener(this); //4
        addMouseMotionListener(this); //4

        l=new Label();
        l.setBounds(10,10,100,150); //5
        add(l); //6
    }

    l1 = new Label();
    l1.setBounds(10,90,100,150);
    add(l1);

    l2 = new Label();
    l2.setBounds(10,180,100,150);
    add(l2);

    setSize(400,400); //7
    setLayout(null); //8
    setVisible(true); //9
}
```

```

// All these methods are called implicitly by the event source.

public void mouseClicked(MouseEvent e)
{
    l.setText("Mouse Clicked");
    System.out.println(e.getX() + " " + e.getY());
}

public void mouseEntered(MouseEvent e) {
    l.setText("Mouse Entered");
}

public void mouseExited(MouseEvent e) {
    l.setText("Mouse Exited");
}

public void mousePressed(MouseEvent e) {
    l.setText("Mouse Pressed");
}

public void mouseReleased(MouseEvent e) {
    l2.setText("Mouse Released");
}

public void mouseDragged(MouseEvent e){
    l1.setText("Mouse Dragged");
}

public void mouseMoved(MouseEvent e) {
    l1.setText("Mouse Moved");
}

public static void main(String[] args) {
    new GUI();
}
*/

```

1. adds only those classes that are present in awt sub-directory.

2. event is a sub-package and all the classes in event is added to the program

3. The object of Label class, which is a GUI 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 an application but a user cannot edit it directly.

(IH of Label)

```

java.lang.Object
    java.awt.Component
        java.awt.Label //Label is derived class of Component

```

4. Frame is both the source and the listener for these events.

the Frame registers itself as a listener for mouse events.

This is done by using addMouseListener() and addMouseMotionListener(), which are members of Component.

```
void addMouseListener (MouseListener ml)  
void addMouseMotionListener (MouseMotionListener mml)
```

5. public void setBounds(int x, int y, int width, int height)

Moves and resizes this component. The new location from

top-left corner is specified by x and y, and
the new size is specified by width and height.

6. public Component add(Component comp)

Appends the specified component to the end of this container.

Parameters:

comp - the component to be added

Returns:

the component argument

Throws:

NullPointerException - if comp is null

Class Frame

INHERITANCE HIERARCHY

java.lang.Object

java.awt.Component // add(Component comp) method belongs to this class

java.awt.Container

java.awt.Window

java.awt.Frame

7. public void setSize(int w, int h) belongs to Component class

Resizes this component so that it has width w and height h.

Parameters:

w - the new width of this component in pixels

h - the new height of this component in pixels

User defined class GUI does not have setSize method but its base class Frame has it, which is inherited by Component class.

8. java.awt.Container contains setLayout method

The Layout managers enable programmer to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers.

There are different built-in classes for layout managers in java.

9. public void setVisible(boolean b) java.awt.Window class contains this method

Shows or hides this Window depending on the value of parameter b.

*/

Ex: HANDLING KEYBOARD EVENTS

```
import java.awt.*;
```

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

```
public class GUI extends Frame implements KeyListener
```

```
{
```

```
    Label l,l1;
```

```
    String msg ="";
```

```
GUI(){
```

```
    addKeyListener(this);
```

```
    l=new Label();
```

```
    l.setBounds(10,10,100,150);
```

```
    add(l);
```

```
    l1 = new Label();
```

```
    l1.setBounds(10,90,200,400);
```

```
    add(l1);
```

```
    setSize(400,400);
```

```
    setLayout(null);
```

```
    setVisible(true);
```

```
}
```

```
public void keyPressed(KeyEvent ke) {
```

```
    l.setText("Key Down");
```

```
    int key = ke.getKeyCode();
```

```
    switch(key) {
```

```
        case KeyEvent.VK_F1:
```

//Virtual key codes present in KeyEvent class Pg no. 645 text book

```

msg += "<F1>";
break;
case KeyEvent.VK_F2:
    msg += "<F2>";
    break;
case KeyEvent.VK_F3:
    msg += "<F3>";
    break;
case KeyEvent.VK_PAGE_DOWN:
    msg += "<PgDn>";
    break;
case KeyEvent.VK_PAGE_UP:
    msg += "<PgUp>";
    break;
case KeyEvent.VK_LEFT:
    msg += "<Left Arrow>";
    break;
case KeyEvent.VK_RIGHT:
    msg += "<Right Arrow>";
break; }

l1.setText(msg);
}

public void keyReleased(KeyEvent ke) {
    l1.setText("KeyReleased");
}

public void keyTyped(KeyEvent ke) {
    msg = msg + ke.getKeyChar();
    l1.setText(msg);
}

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

```

Adapter Classes

Adapter classes are used to simplify the creation task of event handlers.

Ex: GUI class is an Event handler it has registered for KeyListener. Since, it has registered for KeyListener, it has to implement all the methods that are declared in the KeyListener interface. After implementing KeyListener, if GUI class wants to implement only KeyTyped method it is not

allowed.

An adapter class provides **an empty implementation** for all the methods in an event listener interface.

Adapter classes are useful only when some of the events are supposed to be received and processed by a particular event listener interface.

Adapter classes can be inherited by the event handler classes (like GUI) and only those methods that are required can be implemented.

Ex:

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

class frame extends Frame //Event Source
{
    protected Label l; // Label is G-component. Frame is a container

frame(){
    l=new Label();
    l.setBounds(10,10,100,150);
    add(l);

    //Event register method
    addMouseMotionListener(new GUI(this)); //1

    setSize(400,400);
    setLayout(null);
    setVisible(true);
}

//GUI is considered as listener class
public class GUI extends MouseMotionAdapter // Event Listener
{
    frame fr;
    GUI(frame f){
        fr = f;
    }

    public void mouseDragged(MouseEvent me) {
        fr.l.setText("Mouse Dragged");
    }
}
```

```

    }

public static void main(String[] args) {
    new frame();
}
}
/*

```

1. Since, java allows only one class to extend, both Frame and MouseMotionAdapter cannot be extended to a single class.

"frame" class extends Frame and "GUI" extends MouseMotionAdapter class.

Frame is the component which generates an event and also the component which responds to the event (Event listener).

Registering for the event listener has to be done in frame class only.

GUI class implements mouseDragged method hence addMouseMotionListener is passed as an instance of GUI.

addMouseMotionListener cannot be called in GUI class because it is present in Component class whose properties are inherited by Frame class and MouseMotionAdapter has not inherited properties of the Component class.

```

public abstract class MouseMotionAdapter extends Object
                                implements MouseMotionListener

```

*/

TABLE 22-4
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

All adapter classes are by default abstract classes, in which the methods of respective interfaces are null defined. The methods in this class are empty.

```

public abstract class MouseMotionAdapter extends Object

```

implements MouseMotionListener

In other words definitions of compulsory/mandatory methods of implemented interface will not have any statements within it.

Inner Classes

“inner class” is a class defined within another class, or even within an expression.

Use of inner class will simplify the code using event adapter classes.

Ex:

```
import java.awt.*;
import java.awt.event.*;
class frame extends Frame //Event source
{
    protected Label l;

    frame(){
        l=new Label();
        l.setBounds(10,10,100,150);
        add(l);

        //Event register method
        addMouseMotionListener(new gui());

        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
    class gui extends MouseMotionAdapter // INNER CLASS event listener
    {
        public void mouseDragged(MouseEvent me)
        {
            l.setText("Mouse Dragged");
        }
    }// end of class gui
} // end of class frame
public class GUI
```

```

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

```

Swings (GUI components in swings are not thread safe)

Swing is a set of classes that provides more powerful and flexible GUI components than AWT.

AWT graphical components are considered as *heavy weight components*.

```

/*
AWT Graphical components look and feel is fixed. Look and feel of a component is defined by the
platform.
*/

```

Swing eliminates a number of the limitations inherent in the AWT.

Swing is built on the foundation of the AWT.

Swing also uses the same event handling mechanism as the AWT.

Two Key Swing Features

Lightweight components and

Pluggable look and feel.

“Look” refers to the appearance of GUI widgets and **“feel”** refers to the way the widgets behave.

“Pluggable” the look and feel that can be changed during runtime.

Swing Components Are Lightweight

Swing components are written entirely in Java and do not map directly to platform-specific peers.

Because lightweight components are rendered using graphics primitives (like line drawing, circle drawing and different graphical drawing algorithms), they can be transparent, which enables non rectangular shapes.

“Non-rectangular shapes” : Normally Dialog Boxes are rectangular in shape. Various methods can be adopted to make them non-rectangular in shape.

Hence, lightweight components are more efficient and more flexible.

Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.

This means that each component will work in a consistent manner across all platforms.

/*

Swing Supports a Pluggable Look and Feel (PLAF)

Since, each Swing component is rendered by Java code rather than by native code, the look and feel of a component is **under the control of Swing**.

Swing separates the look and feel of a component from the logic of the component.

Since, swing separates the look and feel of a component, it is possible to change the way that a component is rendered without affecting any of its other aspects.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms.

Conversely, it is possible to create a look and feel that acts wrt a specific platform.

Ex: if an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.

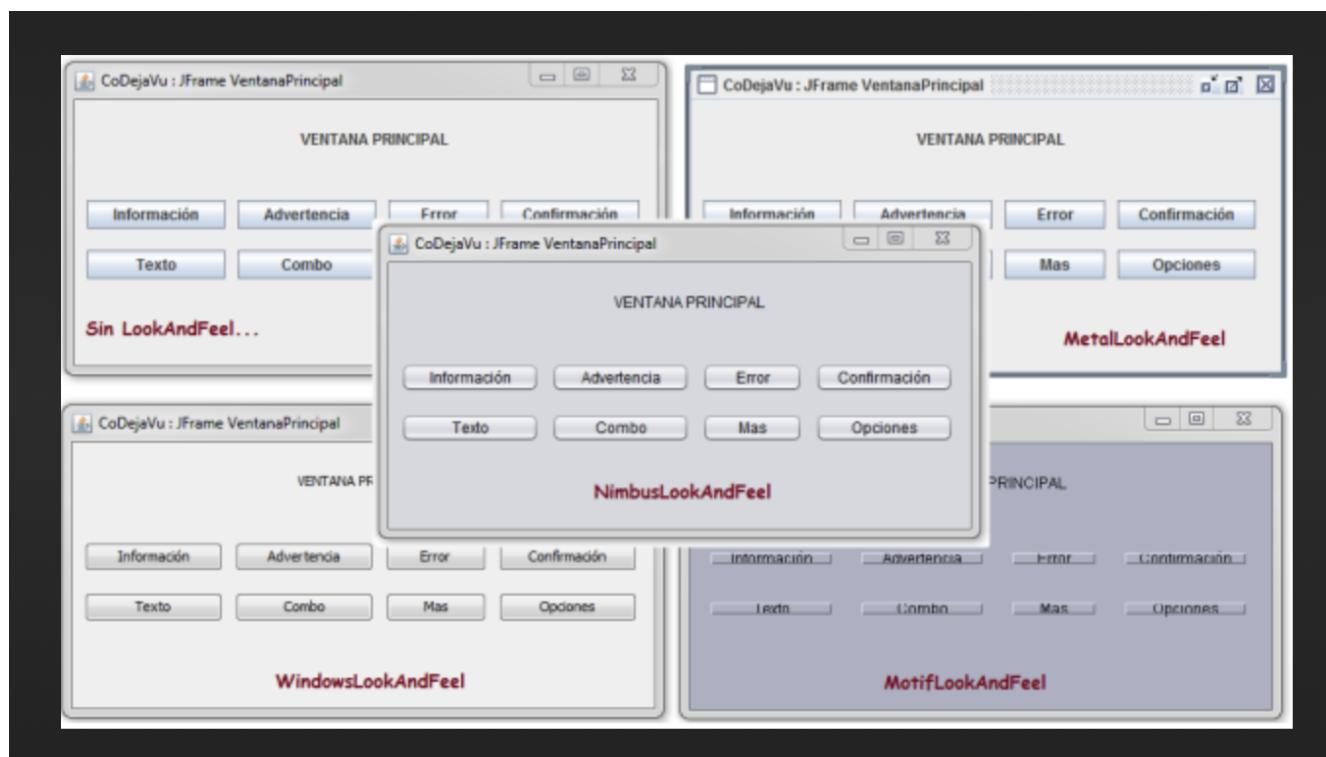
*/

The look and feel can be changed dynamically at run time.

Java SE 6 provides look-and-feels, such as **metal** and **Motif**, that are available to all Swing users.

The metal look and feel is also called the *Java look and feel*.

It is platform-independent and available in all Java execution environments. It is also the default look and feel.



For the Swing program metal look-and-feel will be considered since it is platform independent.

The MVC Connection

In general, a visual/Graphical component is a composite of three distinct aspects:

The way that the **component looks** when rendered on the screen

The way that the **component reacts** to the user

The **state information** associated with the component

MVC means the “**logic that interacts with data**” and “**user interface that represent data**” must be separate entities.

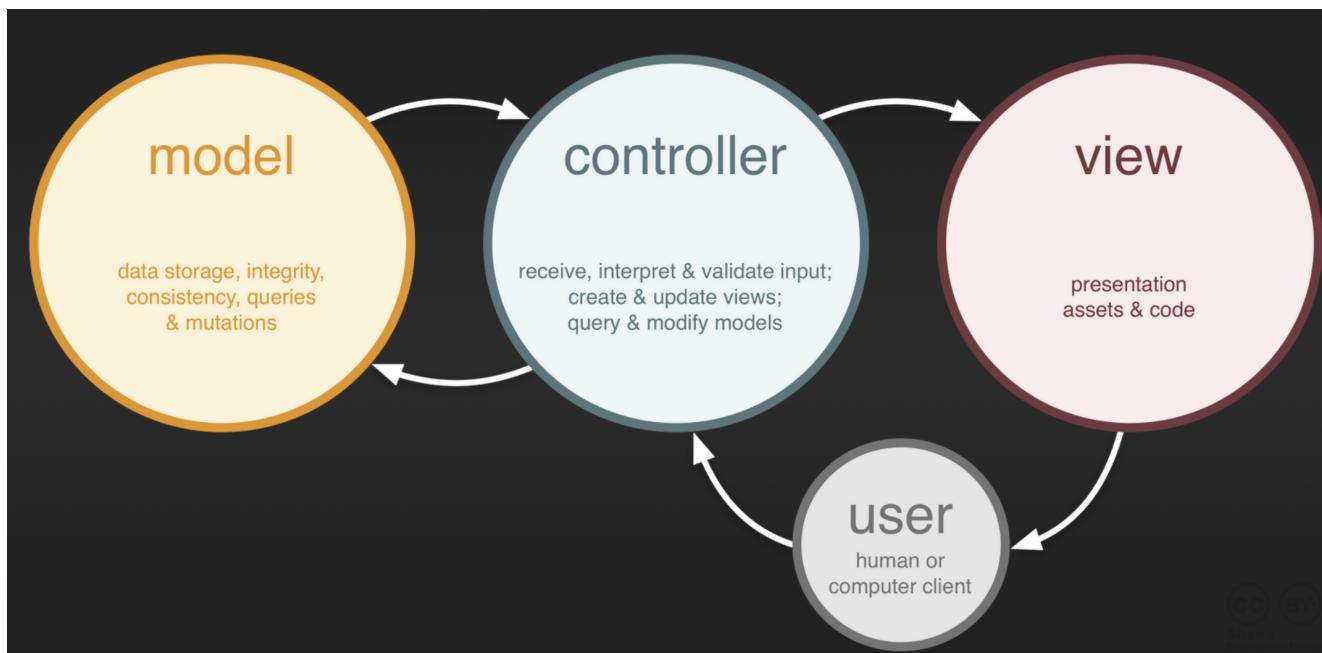
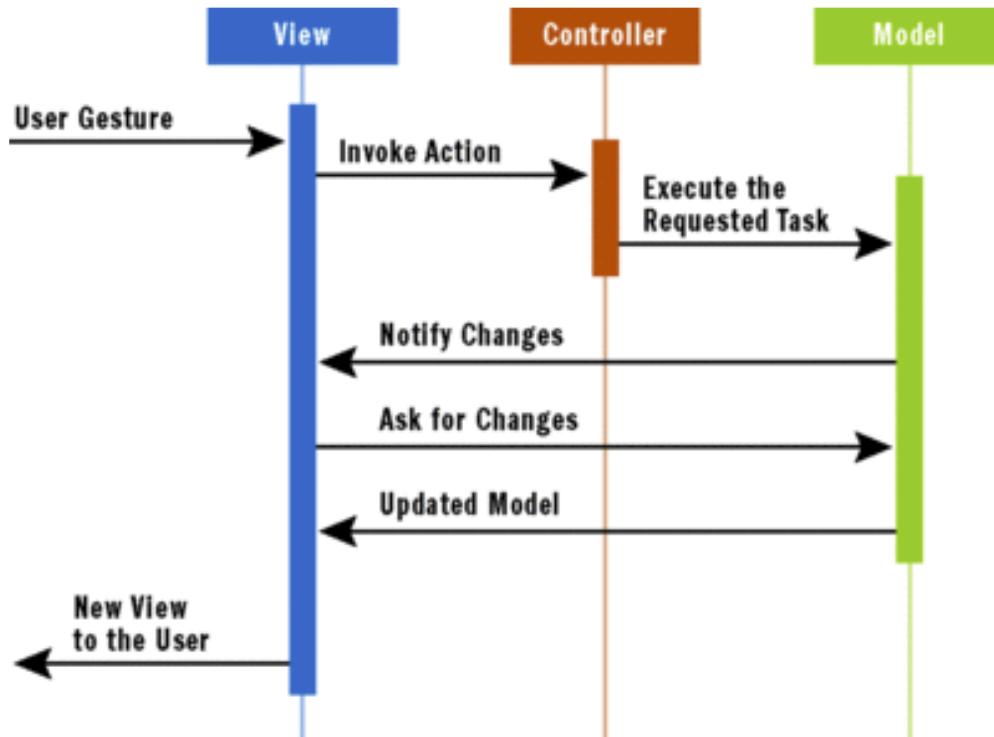
In MVC terminology, the **model** corresponds to the state information associated with the component.

Ex: in case of a check box, the model contains a field/data member that indicates if the box is checked or unchecked.

View determines how the component is displayed on the screen.

Controller determines how the component reacts to the user.

Ex: when the user clicks a checkbox, the controller reacts by changing the model to reflect the user’s choice (checked or unchecked). Which then results in the view being updated.



Ex:

In restaurant example, customer is the view, waiter is the controller and the chef is the model.

Data in this case is equal to the ingredients of the food.

Chef or the cook has to prepare them, just like the model retrieves and processes data from persistent storage.

Waiter is not going to prepare the food like the controller does not communicate with the database (also both the controller and the view), in restaurant example the waiter and customer doesn't know anything about the process of the cook.

Swing uses a modified version of MVC that **combines** the **view** and the **controller** into a **single logical entity** called the ***UI delegate***.

Hence, Swings are termed as **MODEL-DELEGATE / SEPARABLE MODEL** architecture.

To support the Model-Delegate architecture, most **Swing components contain two objects**.

First represents the model.

Second represents UI delegate.

Models are defined by interfaces. (Models are similar to listeners)

Ex: model for a button is defined by the **ButtonModel** interface.

UI delegates are classes that inherit **ComponentUI**.

Ex: UI delegate for a button is **ButtonUI**.

Difference between AWT and Swing

Java AWT	Java Swing
AWT components are platform-dependent	Swing components are platform-independent
AWT components are heavy weight	Swing components are lightweight
AWT does not provide pluggable look and feel	Swing supports pluggable look and feel
AWT provides less GUI components than swing	Swing provides more number of GUI components
AWT does not follow MVC (Model View Controller) Where model represents data View represents presentation Controller acts as an interface between model and view	Swing follows MVC

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*.

A *component* is an independent visual control, such as a push button or slider.

A container is a special type of component that is designed to hold other components.

Further, 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.

Containers are also considered as components, a container can also hold other containers.

Swing defines a *containment hierarchy*, at the top of which must be a *top-level container*.
Ex: JFrame, JApplet, JDialog etc.,

Components

Swing components are derived from the **JComponent** class.

JComponent provides the functionality that is common to all components.

All of Swing's components are represented by classes defined within the package **javax.swing**.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JToggleButton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Table: Swing Components and Containers.

/*Containers

Swing defines two types of containers.

The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog** (heavy weight).
 These containers do not inherit **JComponent**.
 They inherit the AWT classes **Component** and **Container**.

As the name implies, a top-level container must be at the top of a containment hierarchy.

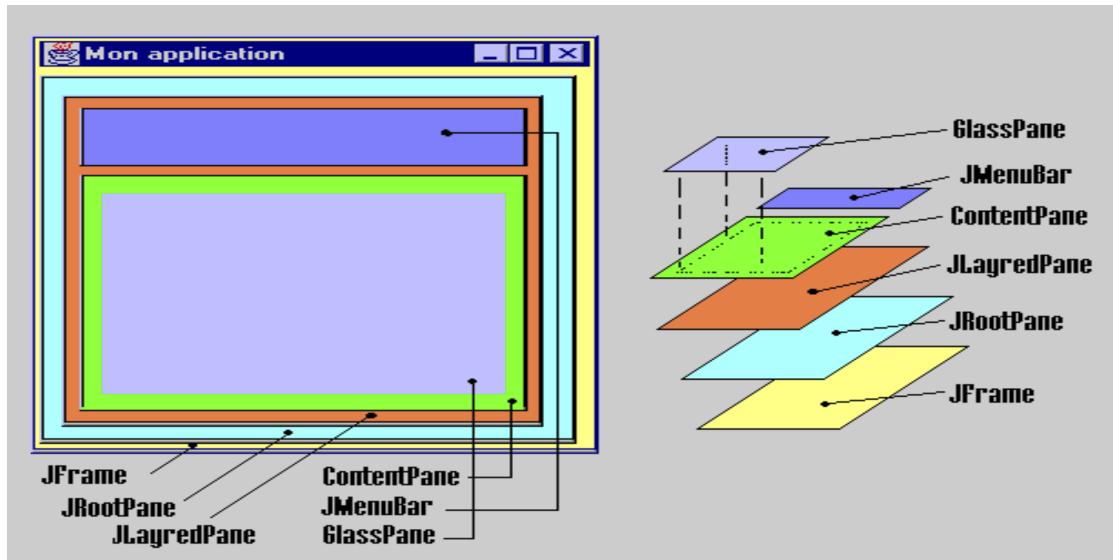
A top-level container is not contained within any other container.
 Ex: most commonly used container for applications is **JFrame**.

The second type of containers supported by Swing are lightweight containers.
 Lightweight 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 because a lightweight container can be contained within another container.

Thus, **JPanel** is used to create subgroups of related controls that are contained within an outer container.*/

The Top-Level Container Panes



The pane with which application will interact the most is the content pane, because this is the pane to which visual components will be added.

Ex: when a component, such as a button is added, to a top-level container, it will be added to the content pane. By default, the content pane is an instance of **JPanel**.

A SIMPLE SWING APPLICATIONS

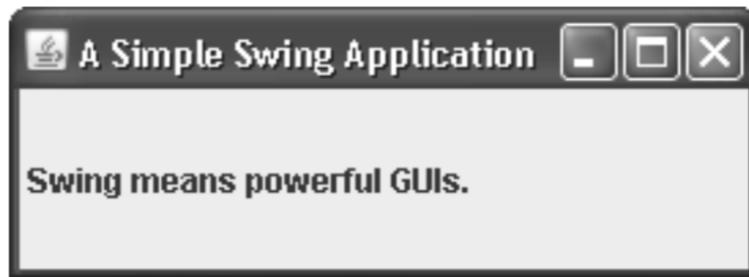
```
import javax.swing.*;  
class SwingDemo {  
    SwingDemo() {  
  
        JFrame jfrm = new JFrame("A Simple Swing Application");  
  
        jfrm.setSize(275, 100);  
  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        JLabel jlab = new JLabel(" Swing means powerful GUIs.");  
  
        jfrm.add(jlab);  
  
        jfrm.setVisible(true);  
    }  
}
```

```

public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run()
        {
            new SwingDemo();
        }
    } // end of anonymous class
); // end of creation of instance
}}

```

Output of the program



The program imports javax.swing, which contains components and models defined by Swing.
Ex: javax.swing defines classes that implement labels, buttons, text controls, and menus.

It will be included in all programs that use Swing.

First statement is constructor of “SwingDemo” user defined class is

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

This creates a **container** called jfrm that defines a rectangular window complete with a title bar;
close,
minimize,
maximize, and
restore buttons; and
a system menu.

It creates a standard, top-level window. **The title of the window is passed to the constructor.**

Next, the window is sized using this statement:

```
jfrm.setSize(275, 100); //void setSize(int width, int height)
```

`setSize()` method (which is inherited by `JFrame` from the AWT class `Component`) sets the dimensions of the window, which are specified in pixels.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated, which is the default behavior.

In some situations, when a top level window is closed the entire application has to terminate, which is done by calling `setDefaultCloseOperation()` as shown.

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Since, the above statement is used in the program closing the window causes the entire application to terminate.

GF: **void setDefaultCloseOperation(int what)**

The value passed in *what* determines what happens when the window is closed.

```
JFrame.EXIT_ON_CLOSE  
JFrame.DISPOSE_ON_CLOSE  
JFrame.HIDE_ON_CLOSE  
JFrame.DO NOTHING_ON_CLOSE
```

The next statement creates a Swing JLabel component:

```
JLabel jlab = new JLabel("Swing means powerful GUIs.");
```

JLabel displays information, which can consist of text, an icon, or a combination of the two.
The label created by the program contains only text, which is passed to its constructor.

Next statement is

```
jfrm.add(jlab); //Component add(Component comp)
```

All top-level containers have a content pane in which components are stored.

Adding a component to a frame's content pane is done by calling `add()` on **JFrame** reference i.e **jfrm**.

The last statement is: **jfrm.setVisible(true);**

If its argument is **true**, the window will be displayed. Otherwise, it will be hidden.

Anonymous inner class

<https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html>

Inside `main()`, a **SwingDemo** object is created, which causes the window and the label to be displayed.

```
SwingUtilities.invokeLater(new Runnable()  
{  
    public void run()  
    {  
        new SwingDemo();  
    }  
});
```

This sequence causes a **SwingDemo** object to be created on the ***event dispatching thread*** rather than on the ***main thread of the application***.

Swing programs are event-driven.

Ex: when a user interacts with a component, an event is generated.

An event is passed to the application by calling an event handler defined by the application.

Event handler is executed on the “**event dispatching thread (EDT)**” provided by Swing and not on the main thread of the application.

//Below URL’s answers why swing components must be executed in EDT

<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>

<https://javarevisited.blogspot.com/2013/08/why-swing-is-not-thread-safe-in-java-Swingworker-Event-thread.html>

Below lines are available in the above link.

“Since EDT thread is most important in Swing and responsible for listening event and updating GUI, you must not do any time consuming task on it, otherwise, you risk your application to become unresponsive or frozen. Another key thing to note, while using multi-threading in Swing development is that, not only GUI components but there model e.g. TableModel for JTable, must be updated in Event Dispatcher thread. One of the crucial [difference between AWT and Swing](#) is that AWT components are thread-safe.”

(***)To execute GUI code on the **event dispatching thread**, one of the two methods that are defined by the **SwingUtilities** class, are supposed to be called.

```
static void invokeLater(Runnable obj)
```

```
static void invokeAndWait(Runnable obj) throws InterruptedException, InvocationTargetException
```

obj is a **Runnable** object that will have its **run()** method called by the event dispatching thread.

The difference between the two methods is that **invokeLater()** returns immediately, but **invokeAndWait()** waits until **obj.run()** returns.

```
/*
invokeAndWait() method in swing is synchronous.
It blocks until the Runnable task is complete.
```

InvokeLater() method in swing is **asynchronous**.

It posts an action event to the event queue and returns immediately. It will not wait for the task to complete

Ex: Same program written using an intermediate class

```
import javax.swing.*;
class gui {
    gui() {

        JFrame jfrm = new JFrame("A Simple Swing Application");

        jfrm.setSize(275, 100);

        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel jlab = new JLabel(" Swing means powerful GUIs.");

        jfrm.add(jlab);

        jfrm.setVisible(true);
    }
}

class inter implements Runnable
{
    Thread t = new Thread(this);

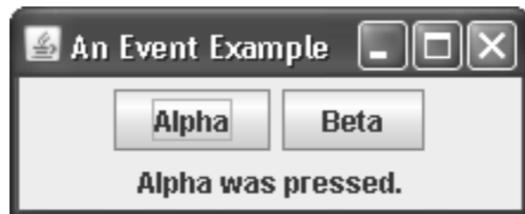
    public inter()
    {
        t.start();
    }

    public void run()
    {
        new gui();
    }
}

class test {
    public static void main(String args[])
    {
        new inter();
        OR
        SwingUtilities.invokeLater(new inter());
    }
}
```

Event Handling

The following program handles the event generated by a Swing push button.



```
//Handle an event in a Swing program.  
import java.awt.*; // because "FlowLayout" is present in this package  
import java.awt.event.*;  
import javax.swing.*;  
class EventDemo {  
JLabel jlab;  
EventDemo() {  
    // Create a new JFrame container.  
    JFrame jfrm = new JFrame("An Event Example");  
    // Specify FlowLayout for the layout manager.  
    jfrm.setLayout(new FlowLayout());  
    /*  
https://docs.oracle.com/javase/7/docs/api/java/awt/FlowLayout.html  
Reg why FlowLayout is necessary is explained in the above url  
*/  
    // Give the frame an initial size.  
    jfrm.setSize(220, 90);  
  
    // Terminate the program when the user closes the application.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    // Make two buttons.  
    JButton jbtnAlpha = new JButton("Alpha");  
    JButton jbtnBeta = new JButton("Beta");
```

```

// Add action listener for Alpha.
//https://docs.oracle.com/javase/7/docs/api/java.awt.event ActionListener.html
/*
Adds the specified action listener to receive action events from this button. Action events occur
when a user presses or releases the mouse over this button
*/
jbtnAlpha.addActionListener (new ActionListener()
{
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
} );

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});

// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);

// Create a text-based label.
jlab = new JLabel("Press a button.");

// Add the label to the content pane.
jfrm.add(jlab);

// Display the frame.
jfrm.setVisible(true);
}

public static void main(String args[]) {
// Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable()
{
    public void run()
    {
        new EventDemo();
    }
})
}

```

```
    }  
});  
} }
```

The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame.

The **java.awt.event** package is needed because it has an **ActionListener** interface and **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**.

Then, **EventDemo()** creates two push buttons, as shown here:

```
 JButton jbtnAlpha = new JButton("Alpha");  
 JButton jbtnBeta = new JButton("Beta");
```

Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

```
 JButton(String msg)
```

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent** instance.

JButton provides the **addActionListener()** method, which is used to add an action listener.

ActionListener interface defines only one method:

```
 void actionPerformed(ActionEvent ae)
```

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button's action events are added by the code shown here:

```
// Add action listener for Alpha.  
jbtnAlpha.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent ae)  
    {  
        jlab.setText("Alpha was pressed.");  
    }  
});  
  
// Add action listener for Beta.  
jbtnBeta.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent ae)  
    {  
        jlab.setText("Beta was pressed.");  
    }  
});
```

Anonymous inner classes are used to provide the event handlers for the two buttons.

Each time a button is pressed, the string displayed in the **jlab** is changed to reflect which button was pressed.

Next, the buttons are added to the content pane of **jfrm**:
`jfrm.add(jbtnAlpha);`

`jfrm.add(jbtnBeta);`

Finally, **jlab** is added to the content pane and window is made visible.

When the program is executed, each time a button is pressed, a message is displayed in the label that indicates which button was pressed.

IMP: All event handlers, such as actionPerformed(), are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If the application needs to do something time consuming as the result of an event, it must use a separate thread.

Create a Swing Applet

Applet definition

<https://docs.oracle.com/javase/tutorial/deployment/applet/index.html>

And

Life cycle of the applet

<https://docs.oracle.com/javase/tutorial/deployment/applet/lifeCycle.html>

Swing-based applets are similar to AWT-based applets, but with an important difference:

A Swing applet extends **JApplet** rather than **Applet**.

JApplet is derived from **Applet**.

Swing applets use four lifecycle methods: **init()**, **start()**, **stop()**, and **destroy()**.

Only the required methods will be overridden in the application program.



Ex:

```
//A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/* THIS HTML CODE IS NEEDED IF THIS PROGRAM IS EXECUTED IN TERMINAL MODE.
<object code="MySwingApplet" width=220 height=90>
</object>
*/
public class Test extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;
    JLabel jlab;
    // Initialize the applet.
    @Override
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable ()
            {
                public void run()
                {
                    makeGUI(); // initialize the GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
        }
    }
}
```

```

// This applet does not need to override start(), stop(), or destroy().
// Set up and initialize the GUI.
private void makeGUI() {
    // Set the applet to use flow layout.
    setLayout(new FlowLayout());
    // Make two buttons.
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent le)
        {
            jlab.setText("Alpha was pressed.");
        }
    });

    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent le)
        {
            jlab.setText("Beta was pressed.");
        }
    });
}

// Add the buttons to the content pane.
add(jbtnAlpha);
add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
add(jlab);
}
}

```

Test extends **JApplet**. All Swing-based applets extend **JApplet** rather than **Applet**.

init() method initializes the Swing components on the event dispatching thread by setting up a call to **makeGUI()**. This is accomplished through the use of **invokeAndWait()** rather than **invokeLater()**.

Applets must use **invokeAndWait()** because the **init()** method must not return until the entire initialization process has been completed.

Inside **makeGUI()**, the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane.

Exploring Swings

Two types of programs can be created using Swings in java

1. Swing application
2. Swing Applet

Swing Component classes are as follows.

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, and they are all derived from **JComponent**.

The Swing components are used in Swing applets rather than swing applications.

JLabel and ImageIcon

JLabel creates a label.

JLabel can be used to display text and/or an icon.

It is a passive component in that it does not respond to user input.

JLabel defines several constructors.

JLabel(Icon icon)

JLabel(String str)

JLabel(String str, Icon icon, int align)

str and *icon* are the text and icon used for the label.

align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label.

align value must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING.

Icons are specified by objects of type Icon, which is an interface defined by Swing.

ImageIcon class can be used to obtain an icon. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.

There are several ways to provide the image, including reading it from a file or downloading it from a URL. ImageIcon constructor is as follows

ImageIcon(String filename)

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

Icon getIcon()

String getText()

The icon and text associated with a label can be set by these methods:

void setIcon(Icon icon)

void setText(String str)

Here, *icon* and *str* are the icon and text, respectively.

Therefore, using `setText()` it is possible to change the text inside a label during program execution.

Ex:

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

//JApplet is deprecated

```
public class Test extends JApplet {
```

```
public void init() {
```

```
    try {
```

```
        SwingUtilities.invokeAndWait( new Runnable()
```

```
        {
```

```
            public void run()
```

```
            {
```

```
                makeGUI();
```

```
            }
```

```
        }
```

```
    );
```

```
}
```

```
    catch (Exception exc)
```

```
    { System.out.println("Can't create because of " + exc);}
```

```
}
```

```
private void makeGUI() {
```

// Create an icon.

```
ImageIcon ii = new
```

```
    ImageIcon("/Users/hemanth//eclipse-workspace//hellot//src//hello//india.gif");
```

// Create a label.

```
JLabel jl = new JLabel("India", ii, JLabel.CENTER);
```

// Add the label to the content pane.

```
add(jl); }
```

```
}
```

JTextField

How to use Text fields in swings in java

<https://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html>

<https://docs.oracle.com/javase/8/docs/api/java/swing/JTextField.html>

“..... *VK_ENTER* results in the listeners getting an *ActionEvent*, and the *VK_ENTER* event is consumed. This is compatible with how AWT text fields handle *VK_ENTER* events.”

JTextField allows users to edit one line of text.

It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components.

Three of **JTextField**’s constructors are shown here:

```
JTextField(int cols)  
JTextField(String str, int cols)  
JTextField(String str)
```

str is the string to be initially presented *cols* is the number of columns in the text field.

If no string is specified, the text field is initially empty.

If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction.

ActionEvent is fired when the user presses ENTER.

CaretEvent is fired each time the caret (i.e., the cursor) changes position.

To obtain the text currently in the text field, call **getText()**.

Ex:

```
import java.awt.*;  
import javax.swing.*;  
  
//Demonstrate JTextField.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet {  
    JTextField jtf;  
    public void init() {  
        try {  
            SwingUtilities.invokeAndWait(new Runnable() {  
                public void run() {  
                    makeGUI();  
                }  
            }  
        }  
        catch (Exception exc) {System.out.println("Can't create because of " + exc);}  
    }  
}
```

```

private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add text field to content pane.
    jtf = new JTextField(15);

    add(jtf);

    jtf.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            // Show text when user presses ENTER.
            showStatus(jtf.getText());
        }
    });
}

```

<https://docs.oracle.com/javase/8/docs/api/javax/swing/event/CaretListener.html#caretUpdate-javax.swing.event.CaretEvent->

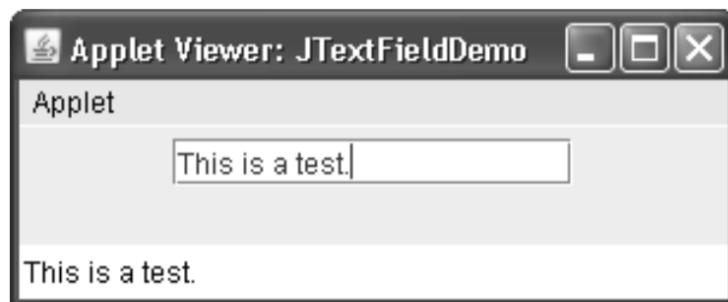
```

jtf.addCaretListener(new CaretListener() {
    public void caretUpdate(CaretEvent e)
    {
        System.out.println(e.getDot());
    }
});

}

```

Output:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**.

All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow programmers to control the behavior of buttons. For example, different icons can be displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon (Icon di)
void setPressedIcon (Icon pi)
void setSelectedIcon (Icon si)
void setRolloverIcon (Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```
String getText ()
void setText (String str)
```

str is the text to be associated with the button.

JButton

JButton class provides the functionality of a push button.

JButton allows an icon, a string, or both to be associated with the push button.

Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, *str* and *icon* are the string and icon used for the button.

str is the string which can be used to identify the button, which is also termed as “**Action Command String**”

When the button is pressed, an **ActionEvent** is generated.

Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, *action command* string associated with the button can be obtained.

By default, this is the string displayed inside the button.

Action command can be set by calling **setActionCommand()** on the button.

Action command can be obtained by calling **getActionCommand()** on the event object.

```
String getActionCommand()
```

Thus, when using two or more buttons within the same application, the action command gives an easy way to determine which button was pressed.

Ex:

```
//Demonstrate an icon-based JButton.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
public class Test extends JApplet implements ActionListener {  
    JLabel jlab;  
    public void init() {  
        try {  
            SwingUtilities.invokeAndWait( new Runnable()  
            {  
                public void run() {  
                    makeGUI();  
                }  
            }  
        );  
    } catch (Exception exc) {System.out.println("Can't create because of " + exc); }  
}  
  
private void makeGUI() {  
    // Change to flow layout.  
    setLayout(new FlowLayout());  
  
    // Add buttons to content pane.  
    ImageIcon india = new  
        ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/india.gif");  
    JButton jb = new JButton("India",india);  
    //jb.setActionCommand("India");  
/*  
The string "India" passed to the constructor of JButton itself will act as Action Command String.  
The same action command string can also be set by invoking setActionCommand( ) method of  
JButton.
```

The string passed during the constructor call will be displayed as a part of the button.

But, the string passed by setActionCommand will not be displayed as a part of the button.

*/

```
    jb.addActionListener(this);
```

/* Above function call has "this" in its parameter list because the class Test is implementing
ActionListener as part of the code, which in turn makes the class Test as both event source and
event listener. */

```

add(jb);

ImageIcon usa = new
    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/usa.jpeg");
jb = new JButton(usa);
jb.setActionCommand("USA");
jb.addActionListener(this);
add(jb);

ImageIcon aus = new
    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/aus.jpeg");
jb = new JButton(aus);
jb.setActionCommand("Australia");
jb.addActionListener(this);
add(jb);

ImageIcon japan = new
    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/japan.jpeg");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
add(jb);

// Create and add the label to content pane.
jlab = new JLabel("Choose a Flag");
add(jlab);

}

// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
}

```

Check Boxes

The **JCheckBox** class provides the functionality of a check box.

JCheckBox defines several constructors.

JCheckBox (String str)

It creates a check box that has the text specified by *str* as a label.

Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a checkbox, an **ItemEvent** is generated.

A reference to the **JCheckBox** that generated the event can be obtained by calling **getItem()** on the **ItemEvent** passed to the **itemStateChanged()** method defined by **ItemListener**.

```
interface ItemListener
{
    void itemStateChanged(ItemEvent e);
/*Invoked when an item has been selected or deselected by the user. The code written for this method
performs the operations that need to occur when an item is selected (or deselected).*/
}
```

The easiest way to determine the selected state of a check box is to call **isSelected()** on the **JCheckBox** instance.

In addition to supporting the normal check box operation, **JCheckBox** allows programmers to specify the icons that indicate when a checkbox is selected, cleared, and rolled-over.

When user clicks a checkbox, an **ItemEvent** is generated. Inside the **itemStateChanged()** method, **getItem()** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected()** determines if the box was selected or cleared. The **getText()** method gets the text for that check box and uses it to set the text inside the label.



Ex:

```
//Demonstrate JCheckbox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet implements ItemListener {  
JLabel jlab;  
public void init() {  
try {  
SwingUtilities.invokeAndWait( new Runnable()  
{  
public void run()  
{  
makeGUI();  
}  
});  
}  
catch (Exception exc) {System.out.println("Can't create because of " + exc);}  
}  
private void makeGUI() {  
// Change to flow layout.  
setLayout(new FlowLayout());  
  
// Add check boxes to the content pane.  
JCheckBox cb = new JCheckBox("C");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("C++");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("Java");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("Perl");  
cb.addItemListener(this);  
add(cb);  
  
// Create the label and add it to the content pane.
```

```

jlab = new JLabel("Select languages");
add(jlab);
}
// Handle item events for the check boxes.
public void itemStateChanged(ItemEvent ie) {
/*
Object getItem( ) method belongs to ItemEvent class which is a common base class for many GUI
components in swings, hence return type is generic in nature.
*/
JCheckBox cb = (JCheckBox)ie.getItem();
if(cb.isSelected())
    jlab.setText(cb.getText() + " is selected");
else
    jlab.setText(cb.getText() + " is cleared");
}
}

```

JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront.

JTabbedPane defines three constructors. Default constructor is considered, which creates an empty control with the tabs positioned across the top of the pane.

JTabbedPane uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab()**. Here is one of its forms:

void addTab(String name, Component comp)

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

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

1. Create an instance of **JTabbedPane**.
2. Add each tab by calling **addTab()**.
3. Add the tabbed pane to the content pane.



Ex:

//Demonstrate JTabbedPane.

```
import javax.swing.*;
```

```
public class Test extends JApplet {
public void init() {
try {
SwingUtilities.invokeAndWait( new Runnable()
{
public void run()
{ makeGUI(); }
} );
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
add(jtp);
}
}//End of class Test
```

```
//Make the panels that will be added to the tabbed pane.  
class CitiesPanel extends JPanel {  
    public CitiesPanel() {  
        JButton b1 = new JButton("New York");  
        add(b1);  
        JButton b2 = new JButton("London");  
        add(b2);  
        JButton b3 = new JButton("Hong Kong");  
        add(b3);  
        JButton b4 = new JButton("Tokyo");  
        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 FlavorsPanel extends JPanel {  
    public FlavorsPanel() {  
        JComboBox jcb = new JComboBox();  
        jcb.addItem("Vanilla");  
        jcb.addItem("Chocolate");  
        jcb.addItem("Strawberry");  
        add(jcb);  
    } }
```

JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component.

The component being scrolled can either be an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.

If the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed.

In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed.

For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors.

JScrollPane(Component comp)

The component to be scrolled is specified by *comp*.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.



Ex://Demonstrate JScrollPane.

```

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

public class Test extends JApplet {
public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run() {
            makeGUI();
        }
    });
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
// Add 400 buttons to a panel.
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++)
    for(int j = 0; j < 20; j++)
    {
        jp.add(new JButton("Button " + b));
        ++b;
    }
// Create the scroll pane.
JScrollPane jsp = new JScrollPane(jp);
// Add the scroll pane to the content pane.
add(jsp);
}
}//End of class Test

```

JList

JList supports the selection of one or more items from a list.

Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed.

JList provides several constructors.

JList(Object[] *items*)

This creates a **JList** that contains the items in the array specified by *items*.

Although a JList will work properly by itself, most of the time it will be wrapped inside JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**, which

specifies only one method, called **valueChanged()**, which is shown here:

void valueChanged(ListSelectionEvent *le*)

Here, *le* is a reference to the object that generated the event.

```
interface ListSelectionListener
```

```
{  
    void valueChanged(ListSelectionEvent le);  
}
```

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode()**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)

Here, *mode* specifies the selection mode. It must be one of these values defined by

ListSelectionModel:

SINGLE_SELECTION

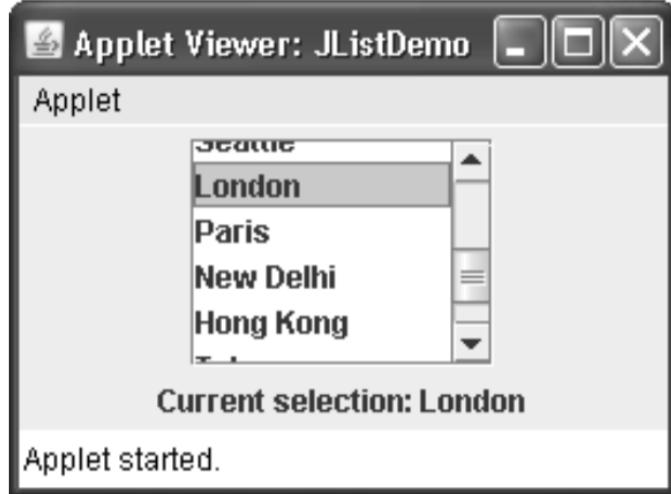
SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, allows the user to select multiple ranges of items within a list.

With single-interval selection, the user can select one range of items.

With single selection, the user can select only a single item.



Ex:

```
//Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
    JList jlst;
    JLabel jlab;
    JScrollPane jscrlp;

    // Create an array of cities.
    String Cities[] = { "New York", "Chicago", "Houston",
        "Denver", "Los Angeles", "Seattle",
        "London", "Paris", "New Delhi",
        "Hong Kong", "Tokyo", "Sydney" };

    public void init() {
        try {
            SwingUtilities.invokeAndWait( new Runnable()
            {
                public void run() {
                    makeGUI();
                }
            });
        } catch (Exception exc) {System.out.println("Can't create because of " + exc);}
    }

    void makeGUI() {
        jscrlp = new JScrollPane(jlst = new JList(Cities));
        jscrlp.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);
        jscrlp.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        jlab = new JLabel("Current selection: " + jlst.getSelectedItem());
        jscrlp.add(jlab);
        add(jscrlp);
    }
}
```

```

private void makeGUI() {
    // Change to flow layout.
    setLayout(new FlowLayout());

    //Create a JList.
    jlst = new JList(Cities);

    // Set the list selection mode to single selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add the list to a scroll pane.
    jscrlp = new JScrollPane(jlst);

    // Make a label that displays the selection.
    jlab = new JLabel("Choose a City");

    // Add selection listener for the list.
    jlst.addListSelectionListener(new ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent le)
        {
            // Get the index of the changed item.
            int idx = jlst.getSelectedIndex();
            // Display selection, if item was selected.
            if(idx != -1)
                jlab.setText("Current selection: " + Cities[idx]);
            else // Otherwise, reprompt.
                jlab.setText("Choose a City");
        }
    });
    // Add the list and label to the content pane.
    add(jscrlp);
    add(jlab);  } }

```

JComboBox

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

A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry.

A combo box can also be created that lets the user enter a selection into the text field.

JComboBox constructor used by the example is shown here:

JComboBox(Object[] items)

Here, *items* is an array that initializes the combo box.

JComboBox uses the ComboBoxModel.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem()** method,

void addItem (Object obj)

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

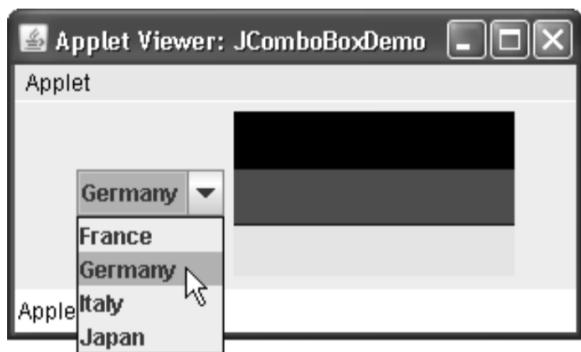
JComboBox generates an **action event** when the user selects an item from the list.

JComboBox also generates an **item event** when the state of selection changes, which occurs when an item is selected or deselected.

To obtain the item selected in the list is to call **getSelectedItem()** on the combo box.

Object getSelectedItem()

There is a need to cast the returned value into the type of object stored in the list.



Ex:

```
//Demonstrate JComboBox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet {  
    JLabel jlab;  
    ImageIcon france, germany, italy, japan;  
    JComboBox jcb;  
    String flags[] = { "france", "usa", "aus", "Japan" };  
public void init()  
{  
    try {  
        SwingUtilities.invokeAndWait( new Runnable()  
        {  
            public void run()  
            { makeGUI(); }  
        }  
    );  
    }  
    catch (Exception exc)  
    {System.out.println("Can't create because of " + exc);}  
}  
  
private void makeGUI()  
{  
    // Change to flow layout.  
    setLayout(new FlowLayout());  
  
    // Instantiate a combo box and add it to the content pane.  
    jcb = new JComboBox(flags);  
    add(jcb);
```

```

// Handle selections.
jcb.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        String s= "/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/";
        s = s + (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".jpeg"));
    }
}
);

// Create a label and add it to the content pane.
jlab = new JLabel(new ImageIcon("france.gif"));
add(jlab);
}
}

```

JTable

JTable is a component that displays rows and columns of data.

Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.

JTable has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

JTable is a component that consists of one or more columns of information.

At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table.

JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors.

JTable (Object data[][], Object colHeads[])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models.

The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format.

The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**.

The third model determines how items are selected, and it is specified by the **ListSelectionModel**.

A **JTable** can generate several different events.

The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**.

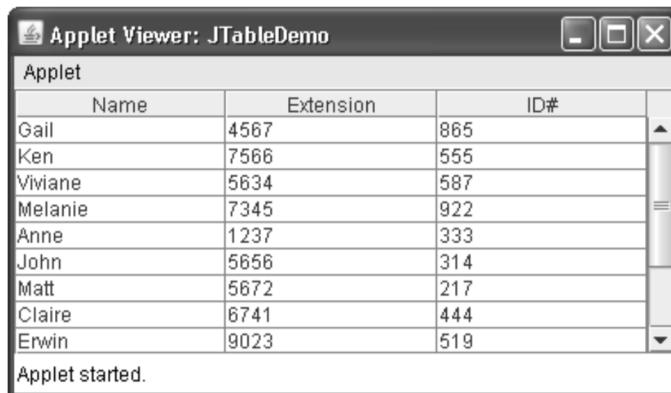
A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected.

A **TableModelEvent** is fired when that table's data changes in some way.

If **JTable** is used only to display data then it is not necessary to handle events.

Steps required to set up a simple **JTable** which can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.



Ex:

```
//Demonstrate JTable.  
import java.awt.*;  
import javax.swing.*;
```

```

public class Test extends JApplet {
public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run() {
            makeGUI();
        }
    }
);
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
// Initialize column headings.
String[] colHeads = { "Name", "Extension", "ID#" };

// Initialize data.
Object[][] data = {
{ "Gail", "4567", "865" },
{ "Ken", "7566", "555" },
{ "Viviane", "5634", "587" },
{ "Melanie", "7345", "922" },
{ "Anne", "1237", "333" },
{ "John", "5656", "314" },
{ "Matt", "5672", "217" },
{ "Claire", "6741", "444" },
{ "Erwin", "9023", "519" },
{ "Ellen", "1134", "532" },
{ "Jennifer", "5689", "112" },
{ "Ed", "9030", "133" },
{ "Helen", "6751", "145" }
};

// Create the table.
JTable table = new JTable(data, colHeads);

// Add the table to a scroll pane.
JScrollPane jsp = new JScrollPane(table);

// Add the scroll pane to the content pane.
add(jsp);
}
}

```

Module 4

Interfaces

2 forms of creating abstract types in java

1. Abstract classes
2. Interface

Interface is a mechanism to achieve abstraction (specifying what a class must do but not how it does it).

Interface implements abstraction in its basic form, by not defining any method within it.

Abstract classes will not implement abstraction to its fullest, because some methods can be defined within it.

There can be only abstract methods in interface.

Instance cannot be created for an interface, but a reference can be created.

Interfaces are used to achieve multiple inheritance in Java.

Java Interface also represents the “IS-A” relationship.

A single class can inherit multiple interfaces.

A single class **cannot** inherit multiple abstract classes.

“implements” is the keyword used to inherit the interface.

G.F:

```
<access-specifier> interface <interface-name>
{
    return-type method-name1(parameter-list); //method declaration
    return-type method-name2(parameter-list);
    .....
    type final varname1 = value; // final and static variables which must be initialized
    type final varname2 = value;
}
```

Ex:

```
interface test
{ }

class <calss-name> [extends classname] [implements interface [,interface...]]
{
....//class body
}
```

If a class **inherits or implements** an interface “test”, then all the methods in the test must be compulsorily

defined by the inherited class.

All methods and variables are implicitly public.

NOTE: An interface can extend (inherit) another interface.

Ex:

```
interface disp
{
    public void display();
    int i=90; // by default i is public final static
}
```

```
class cmp implements disp
{
    public void cmp_method()
    { System.out.println("method of cmp"); }
```

```
    @Override
    public void display()
    { System.out.println("in cmp display"); }
}
```

```
class student implements disp
{
    public void student_method()
    { System.out.println("method of student"); }
```

```
    @Override
    public void display()
    { System.out.println("in student display"); }
}
```

```
class test
{
    public static void main(String[] args)
    {
        disp d = new student();
        d.display();
        //d.student_method(); // CTE
```

```
        d = new cmp();
        d.display();
        //d.cmp_method(); // CTE
```

```
    }  
}
```

Ex:

```
import java.lang.*;  
interface stack  
{  
    void push(Object );  
    Object pop();  
    final int size=3;  
}  
  
class intstack implements stack  
{  
    private int top,a[];  
  
    public intstack()  
    { top=-1; a = new int[size]; }  
  
    public void push(Object obj)  
    {  
        if (top==size-1)  
            {System.out.println("Integer stack full"); return;}  
  
        top=top+1;  
        a[top] = ((Integer) obj).intValue();  
  
/* cmp i = new cmp(9,0); // i is a reference of type Integer  
   Object obj = i; // generates java.lang.ClassCastException  
   int a = ((Integer) i).intValue(); // Exception.  
*/  
    }  
  
    public Object pop()  
    {  
        if (top==-1)  
            {System.out.println("Integer stack empty"); return null; }  
  
        Object obj = new Integer(a[top]);  
        top--;  
        return obj;  
    }  
}
```

```

class fstack implements stack
{
    int top;
    float a[];

    public fstack()
    {   top=-1; a = new float[size]; }

    public void push(Object obj)
    {
        if (top==size-1)
            {System.out.println("Integer stack full"); return; }

        top=top+1;
        a[top] = ((Float)obj).floatValue();
    }

    public Object pop()
    {
        if (top== -1)
            {System.out.println("Integer stack empty"); return null; }

        Object obj = new Float(a[top]);
        top--;
        return obj;
    }
}

class test
{
    public static void main(String args[])
    {
        intstack i = new intstack();
        i.push(new Integer(10));
        System.out.println( ((Integer)i.pop()).intValue() );

        fstack j = new fstack();
        j.push(new Float(10));
        System.out.println( ((Float)j.pop()).floatValue() );

        System.out.println("One interface Multiple method invocation");
        stack s = i;// base class reference pointing to its derived class

        s.push(new Integer(20));
        System.out.println( ((Integer)s.pop()).intValue() );
    }
}

```

```

s = j;
s.push(new Float(30));
System.out.println( ((Float)s.pop()).intValue() );
}
}

```

IMPLEMENTING MULTIPLE INTERFACE & INTERFACE EXTENDING ANOTHER INTERFACE

```
interface accept_value
{ void accept(); }
```

```
interface display_value
{ void display(); }
```

```
/*
interface accept_value
{ void accept(); }
```

//INTERFACE INHERITING PROPERTIES OF ANOTHER INHERITANCE

```
interface display_value extends accept_value
{
    void display();
}*/
```

```
//class test implements display_value
class test implements accept_value, display_value
{
    public void accept()
    {
        System.out.println("In accept");
    }
    public void display()
    {
        System.out.println("In display");
    }

    public static void main (String[] args)
    {
        test a = new test();
        a.accept();
        a.display();
    }
}
```

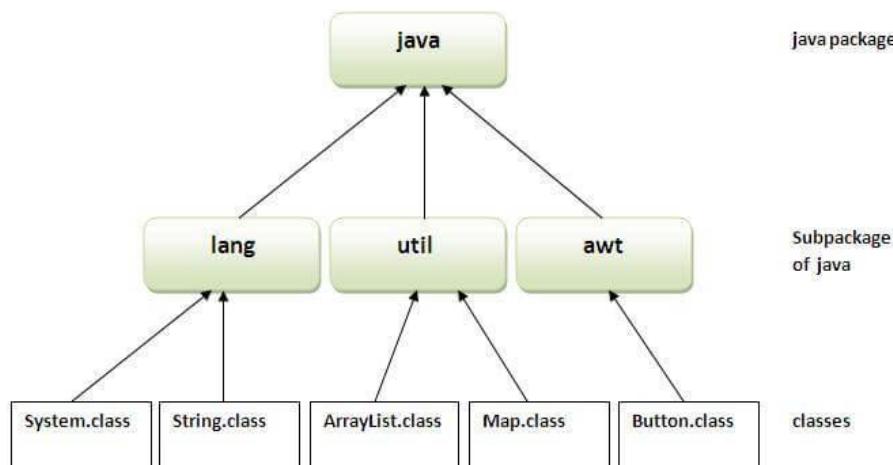
```
    }  
}
```

Packages

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two forms, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



Package provides access protection.

Package removes naming collision.

Package is used to categorize the classes and interfaces so that they can be easily maintained.

*Creating a package actually creates a directory with the same name as the package name.
(in Eclipse a default package is created for each project, if a suitable name is provided for the package, then a directory of the same name will be created to hold on to the classes, which are inside the package)*

In console based implementation of packages, directory name which matches package name has to be created manually and respective classes must be kept in that directory.

Ex:

```
package rns.engineering.cse
```

rns is the main directory and main package name

engineering is the sub directory of rns and intermediate package name

cse is the subdirectory of engineering and final package name

rns ---> engineering ----> cse

Any class to be added to this package must be stored in the “cse” directory.

Creating packages

package rns.engineering;

public class CSE

//public specifier for class, to access CSE class outside the package //rns.engineering.cse

{

 public static void print()

 { System.out.println("In rns.engineering package class CSE"); }

}

ONLY ONE PUBLIC CLASS IS ALLOWED IN ONE PACKAGE

ANY OTHER CLASS IN THE SAME PACKAGE MUST NOT BE PUBLIC.

class ISE

{

 public static void print()

 { System.out.println("In rns.engineering package class ISE"); }

}

package rns.pucollege;

public class PUC

{

 public static void print()

 { System.out.println("In rns.pucollege package class PUC"); }

}

//Test.java

import rns.engineering.CSE;

import rns.pucollege.*;

class Test

{

 public static void main(String args[])

{

 CSE.print();

 PUC.print();

}

}

Access protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table: Class member access

Any member declared as public can be accessed from anywhere.

Any member declared as private cannot be accessed outside its class.

Any member declared as protected can be accessed outside the current package, but only inside the subclasses . Non-sub-classes cannot access protected members.

When a member default access specifier, it is visible to subclasses as well as to other classes in the same package.

A class in a package can have only **public, final or default access specifiers**.

When a class is **declared as public**, it is accessible anywhere in the project. (Project is a combination of several packages).

When a class is public, it must be the **only public class** declared in the file, and the **file must have the same name as the class**.

If a class is **final**, it cannot be inherited by another class.

If a class has **default access**, then it can only be accessed by other classes within the same package.

Ex:

File path: rns\engineering\CSE.java

package rns.engineering;

public class CSE

{

private int ipri;

```
protected int ipro;
public int ipub;
int inomod;
}
```

File path: rns\engineering\NonSubClassSamePackage.java

```
package rns.engineering;
public class NonSubClassSamePackage {

    public void access()
    {
        System.out.println("In Same Package Non-Subclass");
        CSE obj = new CSE();
        //obj.ipri = 90; // CTE
        obj.ipro = 900;
        obj.ipub = 9000;
        obj.inomod = 89; } }
```

File path: rns\engineering\SubClassSamePackage.java

```
package rns.engineering;
public class SubClassSamePackage extends CSE
{
    public void access()
    {
        System.out.println("In Same Package Sub class");
        //ipri =90; // CTE
        ipro = 900;
        ipub = 9000;
        inomod = 89; } }
```

File path: rns\pucollege\NonSubClassDifferentPackage.java

```
package rns.pucollege;
import rns.engineering.CSE;
public class NonSubClassDifferentPackage
{
    public void access()
    {
        System.out.println("In Different Package non-sub-class");
```

```

CSE cse = new CSE();
//cse.ipri=90; //CTE
//cse.ipro = 90; //CTE
//cse.inomod = 90; // CTE
cse.ipub = 90;
}
}

```

File path: rns\pucollege\SubClassDifferentPackage.java

```

package rns.pucollege;
import rms.engineering.CSE;
public class SubClassDifferentPackage extends CSE
{
    public void acess()
    {
        System.out.println("In Different Package Sub class");
        //ipri=90; // CTE : not visible
        ipro = 900;
        ipub = 9000;
        //inomod = 89; // CTE : not visible } }

```

File path: src\Test..java

```

package Test;
import rms.engineering.*;
import rns.pucollege.*;
class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SubClassSamePackage p = new SubClassSamePackage();

        NonSubClassSamePackage q = new NonSubClassSamePackage();

        SubClassDifferentPackage r = new SubClassDifferentPackage();

        NonSubClassDifferentPackage s = new NonSubClassDifferentPackage();

        p.access();           q.access();

        r.acess();           s.access(); } }

```

Ex:

File: p1\Protection.java

```
package p1;
public class Protection
{
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("Protection constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

File: p1\Derived.java

```
package p1;
class Derived extends Protection {
    Derived() {
        System.out.println("Derived constructor");
        System.out.println("n = " + n);

        //class only
        //System.out.println("n_pri = "4 + n_pri);

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

File: p1\SamePackage.java

```
package p1;

class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("Same package non-derived constructor");
```

```

System.out.println("n = " + p.n);

//class only
//System.out.println("n_pri = " + p.n_pri);

System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}

}//End of class SampePackage

```

File: p2\Protection2.java

```

package p2;
import p1.*;

class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package constructor");
        // class or package only
        //System.out.println("n = " + n);

        // class only
        //System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

File: p2\ OtherPackage.java

```

package p2;
import p1.*;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");

        // class or package only
        // System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}

```

```
// Demo package p1.  
package p1;  
package p2;  
  
// Instantiate the various classes in p1.  
public class Demo {  
    public static void main(String args[]) {  
        Protection ob1 = new Protection();  
        Derived ob2 = new Derived();  
        SamePackage ob3 = new SamePackage();  
  
        Protection2 ob1 = new Protection2();  
        OtherPackage ob2 = new OtherPackage();  
    }  
}
```

Importing Packages

```
*****
```

Packages are a way to categorize different classes from each other.

All built-in java classes are stored in packages, there is not a single builtin java class that is present in the unnamed-default package.

All standard classes are stored in some named package.

It is a tedious task to type long dot-separated package path names for each and every class present in the package to use it.

Ex:

```
class test {  
    public static void main(String args[]) {  
  
        java.util.Scanner ip = new java.util.Scanner(java.lang.System.in);  
  
        java.lang.System.out.println("Afd");  
    }  
}
```

Java includes import statements to bring certain classes, or entire packages, into visibility.

Once imported, a class can be referred to directly, using only its name. The import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

G.F:

import pkg1[.pkg2].(classname|*);

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package or all the classes in the package.

Ex:

```
import java.util.Date;  
import java.io.*;
```

Multi-Threaded Programming

All modern operating systems support multitasking. There are two different types of multitasking.

1. Process-based
2. Thread-based.

Process-based multitasking

A process is a program under execution.

If a system allows more than one process to execute simultaneously, it is termed as *process-based multitasking*.

Ex: A process-based multitasking facilitates to run the Java compiler at the same time that you are using a text editor.

Processes are heavyweight tasks that require their own separate **address spaces**.

*/*Address space is the amount of memory allocated for all possible addresses for a computational entity, such as a device, a file, a server, or a networked computer.*

Address space may refer to a range of either physical or virtual addresses accessible to a processor or reserved for a process.

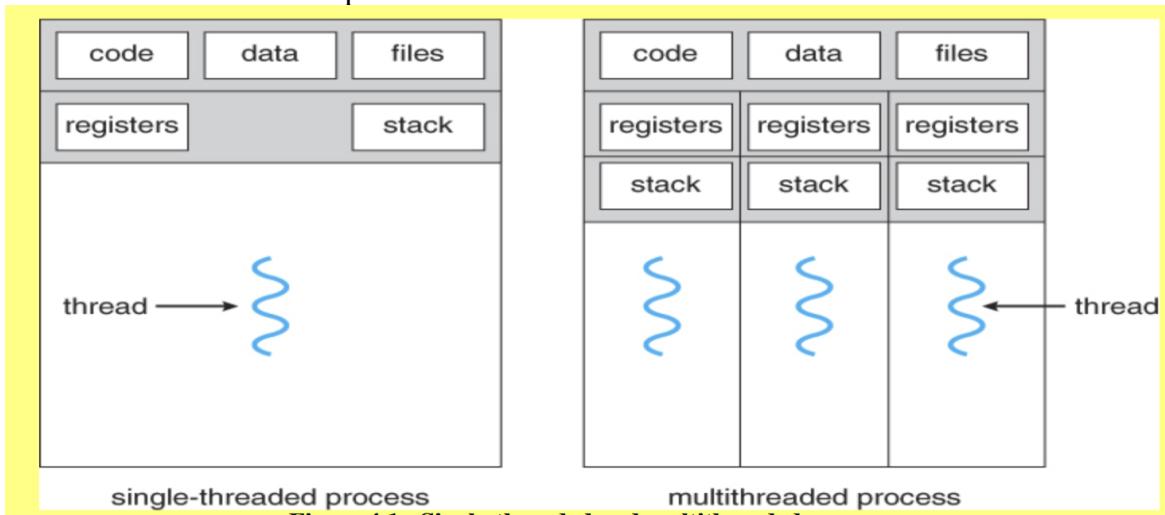
On a computer, each computer device and process is allocated address space, which is some portion of the processor's address space/*

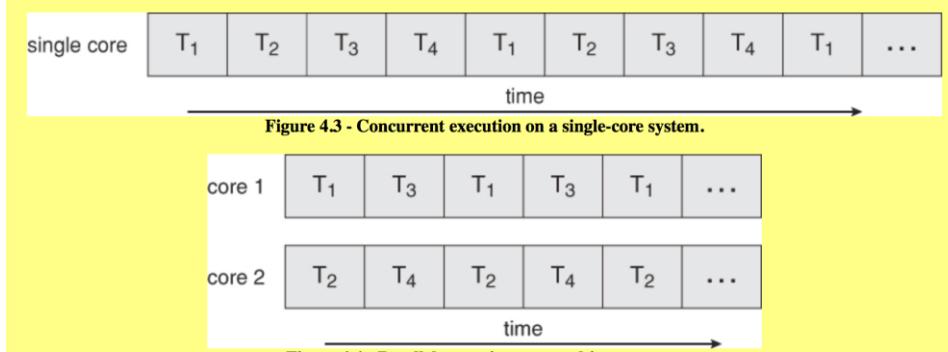
Interprocess communication is expensive and limited.

Context switching from one process to another is also costly. (Costly in terms of CPU time wastage)

Thread-based multitasking

Thread is the smallest unit of dispatchable/executable code.





https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

A single program can perform two or more tasks simultaneously.

Ex: A text editor can format text at the same time that it is printing. Printing and formatting will be done by two separate threads which belong to a single process.

“Multitasking threads” require less overhead than “multitasking processes”.

Threads are a lightweight process.

They share the same address space and cooperatively share the same heavyweight process.

Inter Thread communication is inexpensive.

Context switching from one thread to the next is low cost.

Java provides built-in support for *multithreaded programming*.

A multithreaded program contains two or more parts of the program that can run **concurrently**. Each part of such a program is called a *thread*, and each thread defines a separate path of execution.

Multithreading is a specialized form of multitasking.

While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.

Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

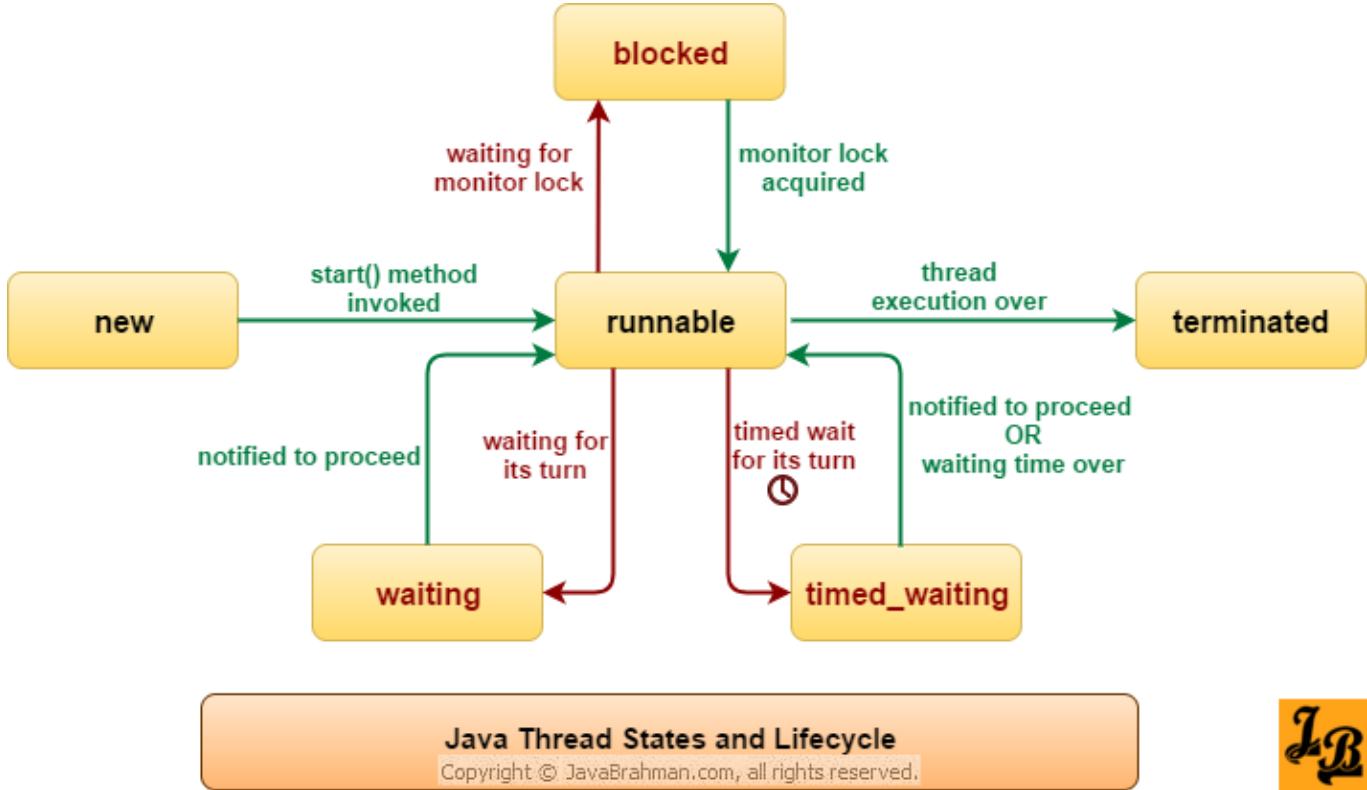
Ex: Multithreading is especially important for the interactive, networked environment in which Java operates, because idle time is common.

The transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, user input is much slower than the computer.

In a single-threaded environment, a program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.

Multithreading lets you gain access to this idle time and put it to good use.

Thread states



“Thread” Class and “Runnable” Interface

Multithreading in java is built on the **Thread class**, its methods, and its companion **Runnable interface**.

Thread class, encapsulates a thread under execution.

To create a new thread, the program will either **extend Thread** or **implement the Runnable interface**.

The Thread class defines several methods that help manage threads.

getName Obtain a thread's name.

getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>

All java programs by default have a **single thread termed as main**.

The Main Thread

When a Java program begins execution, one thread begins running immediately i.e main function.

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

```
class Thread //built-in class
{
    public static Thread currentThread( );
    public final void setName(String threadName);
    public final String getName( );
    public static void sleep(long milliseconds) throws InterruptedException
}
```

currentThread() returns a reference to the thread in which it is called.

setName() is used to change the internal name of the thread, **threadName** specifies the name of the thread.

Ex:

```
main( ) thread can be controlled through a Thread object.
A reference for main( ) has to be obtained using currentThread( ).
// Controlling the main Thread.
class CurrentThreadDemo
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);

        t.setName("My Thread"); // changing the name of the thread

        System.out.println("After name change: " + t);

        try // becz, Thread.sleep( ) is anticipated to generate InterruptedException
        {
            for(int n = 5; n > 0; n--)
            {
                System.out.println(n);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        { System.out.println("Main thread interrupted"); }
    }
}
```

```
}
```

Output

```
*****
```

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Numerical values are displayed from 5 to 0 with a sleep interval of 1 second by calling sleep() method.

Exceptions will be generated automatically by Thread.sleep(), if another thread tries to interrupt a sleeping thread.

Thread information is displayed in the following manner.

the name of the thread, (which is main)

its priority, (default priority is 5)

and the name of its group. (main is the name of the group)

Creating a Thread

```
*****
```

A thread is generated by **instantiating an object of the builtin class Thread**.

Java has two ways of creating a thread:

- implement the Runnable interface.
- extend the Thread class.

Implementing Runnable

```
*****
```

Generating a thread in java, implementing Runnable interface is a two step process.

1) A way to generate a thread is to create a class that implements the Runnable interface.

```
interface Runnable
{
    void run();
}
```

run() is the entry point for a thread in the program. Statements that are supposed to be executed in thread must be coded in run() method. Even function calls can also be made within the run () method.

2) An object of type Thread must be instantiated from within the class which implements Runnable interface.

Thread defines several constructors, one of them is

```
class Thread
{
    Thread( Runnable threadOb,      String threadName);
}
```

threadOb: is an instance of a class that implements the Runnable interface.

ThreadName: name of new thread.

After a new thread object is created, it will not start executing until the start() method is called, which is declared within the Thread class.

In essence, start() executes a call to run().

```
class Thread  {

    public void start() throws IllegalThreadStateException //if the thread was already started.
}
```

Thread process must start its execution only once and not many times.

Ex:

```
class NewThread implements Runnable
{
    Thread t;
```

//Since, start() and run() is present only in Thread class, an instance of Thread must be created here

```
NewThread()
{
    // Thread object is created, which is used to call start( ) method.
    t = new Thread(this, "Demo Thread");
/*
    "this" is a reference of type NewThread which implements Runnable interface.
    Hence, it is passed as the first parameter.
*/
    System.out.println("Child thread: " + t);
```

```
    // Create a new, second thread. Becz main is considered as the 1st thread
    t.start(); // Begins thread execution
}
```

// This is the entry point for thread.

```
@Override
public void run()
{
```

```

try
{
    for(int i = 5; i > 0; i--)
    {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
    }
}
catch (InterruptedException e)
{ System.out.println("Child interrupted."); }

System.out.println("Exiting child thread.");
}

}//End of class NewThread

class ThreadDemo
{
public static void main(String args[])
{
    new NewThread(); // create a new thread

    try
    {
        for(int i = 5; i > 0; i--)
        {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
        }
    }
    catch (InterruptedException e)
    {System.out.println("Main thread interrupted.");}

    System.out.println("Main thread exiting.");
}
}

}//End of class ThreadDemo

```

a new Thread object is created by the following statement inside constructor

t = new Thread(this, "Demo Thread");

Passing “**this**” as the first argument indicates that the new thread has to be called on this object.

Next, start() is called, which starts the thread execution beginning from the run() method.

After calling start(), NewThread’s constructor returns to main(), then main thread resumes, it enters its for loop.

Both threads continue running, sharing the CPU, until their loops finish.

Output may vary based on processor speed and task load

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

In a multithreaded program, often the main thread must be the last thread to finish running.

(automatically the main thread will not conclude the execution at last, but forcibly it will be made to conclude the execution after all the child threads completes its execution.)

In case if any resources that are acquired by the main thread and in turn they are used by the child threads, then the main thread has to wait until the child thread concludes its execution.

Ex: Scanner instance can be instantiated in main thread and in turn it can be used by child thread.

Only after the usage of child thread is over, the scanner instance can be closed in the main thread.

Closing of scanner instance in child thread, will avoid further usage of the same in main thread.

Habit, of acquiring resources will be done in main because it stays for a longer period of time, since the execution starting point is main and the usual exit of control is from main.)

The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Extending Thread

Another way to create a thread is to create a new class that extends the “Thread” class, and then to create an instance of it.

Extending class must override the run() method, which is the entry point for the new thread.

It must also call start() to begin execution of the new thread.

Ex:

```
class NewThread extends Thread {  
    NewThread() {  
        // Calling Thread class constructor  
        super("Demo Thread");  
        System.out.println("Child thread: " + this);  
        start(); // Start the thread, which will in turn call run() method.  
    }  
}
```

// This is the entry point for the second thread. Overriding run method, which is present in “Thread” class.

```
@Override  
public void run() {  
    try {  
        for(int i = 5; i > 0; i--) {  
            System.out.println("Child Thread: " + i);  
            Thread.sleep(500);  
        }  
    }  
    catch (InterruptedException e) {  
        System.out.println("Child interrupted.");  
        System.out.println("Exiting child thread.");  
    }  
}//End of class NewThread
```

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

```

new NewThread(); // create a new thread, unreferenced instance
try {
    for(int i = 5; i > 0; i--)
    {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{
    System.out.println("Main thread interrupted.");
    System.out.println("Main thread exiting.");
}
} //End of class ExtendThread

```

Output will be the same as before.

Whether to use *implements* or *extends* to create a thread

Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run() to create a thread. This is, of course, the same method required by Runnable.

Thread class must be extended only when they are being enhanced or modified in some way.

If no other thread methods are overridden other than run() method, then it is best suggested to implement Runnable.

Creating Multiple Threads

```

//Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
}

```

// This is the entry point for thread.

```

@Override
public void run() {
/*!!!

```

```

super.run();
Is a valid statement at this point. Generic run() in Thread class does not do any specific task and is
negligible.
*/
try {
    for(int i = 2; i > 0; i--) {
        System.out.println(t.getName() + ": " + i);
        Thread.sleep(1000);
    }
}
catch (InterruptedException e)
{System.out.println(name + "Interrupted");}
System.out.println(name + " exiting.");
} }

```

```

class Test {
public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
try {
    // wait for other threads to end
    Thread.sleep(10000); //main sleeps for 10 seconds, to ensure that the main thread finishes last.
}
catch (InterruptedException e)
{ System.out.println("Main thread Interrupted"); }

System.out.println("Main thread exiting.");
} }

```

Output:

```

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 2
Two: 2
New thread: Thread[Three,5,main]

```

Three: 2
Two: 1
Three: 1
One: 1
Two exiting.
One exiting.
Three exiting.
Main thread exiting.

isAlive() and join()

Making main() thread to sleep for a long duration of time, allowing child processes to complete its execution is not a competent solution. (Asynchronous waiting)

Two ways exist to determine if a thread is alive or not.

1. **isAlive()**
2. **join()** (Synchronous waiting)

```
class Thread {  
    final boolean isAlive()  
    final void join() throws InterruptedException  
}
```

isAlive() method returns true if the thread upon which it is called is still running, otherwise false.

Join() method, puts current thread on wait until the thread on which it has called finishes its execution.

If thread (which has invoked join())is interrupted then it will throw InterruptedException.

Ex: using join() method

```
class NewThread extends Thread {  
  
    NewThread(String n) {  
        this.setName(n);  
        System.out.println("Child thread: " + this);  
        start(); // Start the thread  
    }  
  
    @Override  
    public void run() {  
        try {  
            for(int i = 3; i > 0; i--) {  
                System.out.println( getName() + " Thread: " + i);  
                Thread.sleep(500);  
            }  
        }  
        catch (InterruptedException e) { System.out.println("Child interrupted."); }  
  
        System.out.println("Exiting "+getName()+" child thread.");  
    }  
}//End of class NewThread  
  
class Test {  
    public static void main(String args[]) {  
        NewThread nt = new NewThread("First"); // create a new thread  
        NewThread nt1 = new NewThread("Second");  
  
        try {  
            nt.join();  
            nt1.join();  
        }  
        /* Since nt.join( ) is called on main thread, main thread suspends execution until thread named "First"  
         completes its execution.  
         Once nt thread completes its execution, main( ) thread resumes execution and subsequent call to  
         nt.join( ) will not execute, because already nt thread has finished its execution.  
        */  
        catch (InterruptedException e) {System.out.println("Main thread interrupted.");}  
        System.out.println("Main thread exiting.");  
    } } //End of class ThreadDemo
```

Output:

Child thread: Thread[First,5,main]

Child thread: Thread[Second,5,main]

First Thread: 3

Second Thread: 3

First Thread: 2

Second Thread: 2

First Thread: 1

Second Thread: 1

Exiting First child thread.

Exiting Second child thread.

Main thread exiting.

Ex: using isAlive() method

```
class NewThread extends Thread {  
  
    NewThread(String n)  
    {  
        this.setName(n);  
        System.out.println("Child thread: " + this);  
        start(); // Start the thread  
    }  
  
    public void run() {  
        try {  
            for(int i = 3; i > 0; i--)  
            {  
                System.out.println( getName() +" Thread: " + i);  
                Thread.sleep(500);  
            }  
        }  
        catch (InterruptedException e)  
        { System.out.println("Child interrupted."); }  
  
        System.out.println("Exiting "+getName()+" child thread.");  
    } //End of class NewThread
```

```
class Test {  
    public static void main(String args[]) {  
        NewThread nt = new NewThread("First"); // create a new thread  
        NewThread nt1 = new NewThread("Second");  
  
        while(nt.isAlive());  
  
        while(nt1.isAlive());  
  
        System.out.println("Main thread exiting.");  
    } } //End of class ThreadDemo
```

Output:

Child thread: Thread[First,5,main]
Child thread: Thread[Second,5,main]
First Thread: 3
Second Thread: 3
First Thread: 2
Second Thread: 2
First Thread: 1
Second Thread: 1
Exiting First child thread.
Exiting Second child thread.
Main thread exiting.

Thread Priorities

<https://www.informit.com/articles/article.aspx?p=26326&seqNum=5>

Priorities Determine which thread gets CPU allocated and gets executed first.

Thread priority values in java range from 1 to 10, 1 being the least priority and 10 being the highest.

Higher the thread priority, larger is the chance for a process of getting executed first.

Ex:

Two threads are ready to run.

First thread priority is 5 and begins execution.

Second thread priority if assumed to be 10 comes in for execution,

then First thread may suspend its execution relieving the control to Second thread.

A thread's priority is also used to decide when to switch from one running thread to the next termed as "context switch".

`setPriority()` method, is used to set priority for a thread

```
class Thread {  
    final void setPriority(int level)    level specifies the priority  
}
```

level value must be in the range MIN_PRIORITY and MAX_PRIORITY, these values are 1 and 10, respectively.

Default priority of a thread is NORM_PRIORITY, which is equal to 5.

MIN_PRIORITY, MAX_PRIORITY & NORM_PRIORITY's are defined as **static final variables** within Thread.

Current priority of a thread can be obtained by calling `getPriority()` method of Thread,

```
class Thread {  
    final int getPriority()  
}
```

Theoretically higher-priority threads get more CPU time than lower-priority threads.

Ex:

```
//Demonstrate thread priorities.  
class clicker implements Runnable  
{  
    long click = 0;  
    Thread t;  
    private volatile boolean running = true;  
  
    public clicker(int p) {  
        t = new Thread(this);  
        t.setPriority(p);  
    }  
  
    public void run() {  
        while (running)  
            click++;  
    }  
  
    public void stop() { running = false; }  
  
    public void start() { t.start(); }  
}  
  
class test {  
    public static void main(String args[]) {  
        (Thread.currentThread()).setPriority(Thread.MAX_PRIORITY);  
  
        clicker hi = new clicker(Thread.NORM_PRIORITY + 4);  
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);  
  
        hi.start();  
        lo.start();  
        try {  
            Thread.sleep(1);  
        }  
        catch (InterruptedException e)  
        { System.out.println("Main thread interrupted."); }  
  
        lo.stop();  
        hi.stop();  
  
        // Wait for child threads to terminate.
```

```

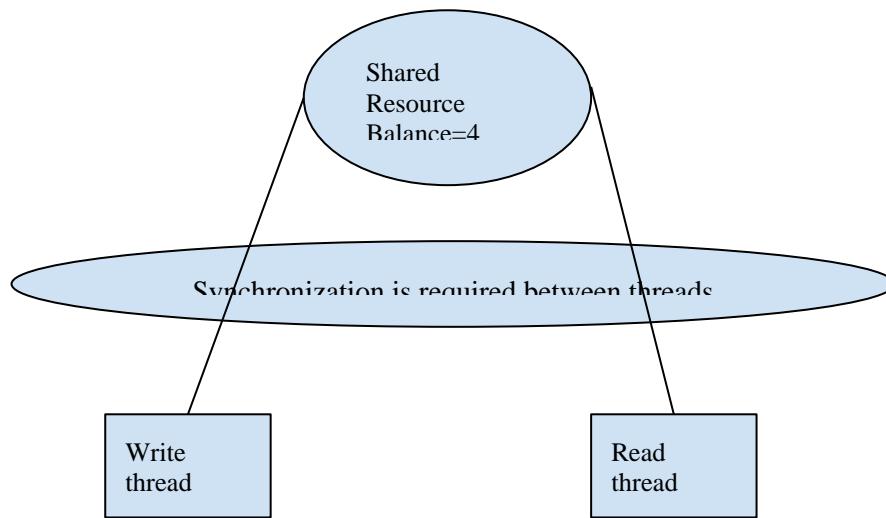
try {
    hi.t.join();
    lo.t.join();
}
catch (InterruptedException e)
{ System.out.println("InterruptedException caught"); }

System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}
}

```

Synchronization

Two threads can access a shared resource, if two threads access a resource **simultaneously** error may ensue.



Synchronization is a mechanism to achieve exclusive access to shared resources by more than one thread.

Key to synchronization is the concept of monitor (also called a semaphore).

/*

Monitors

Abstract Data Type for handling/defining shared resources

Comprises:

Shared Private Data

The resource Cannot be accessed from outside

Procedures that operate on the data

Gateway to the resource

Can only act on data local to the monitor

Synchronization primitives

Among threads that access the procedures

Monitors guarantee mutual exclusion

Only one thread can execute a monitor procedure at any time.
“in the monitor”

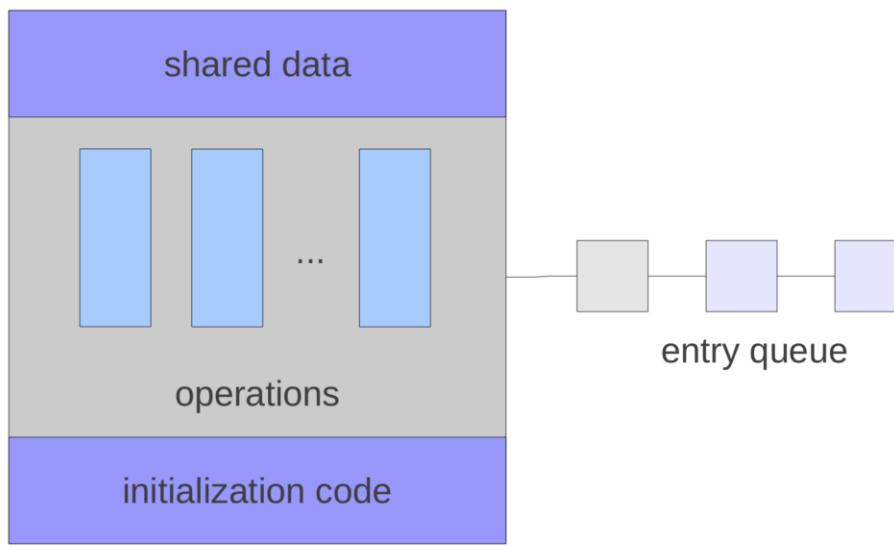
If second thread invokes a monitor procedure at that time
It will block and wait for entry to the monitor
Need for a wait queue

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations
    procedure P1(. . .) {
        . . .
    }
    procedure P2(. . .) {
        . . .
    }
    :
    procedure PN(. . .) {
        . . .
    }
    initialization_code(. . .)
    . . .
}
```

For example:

```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . .
    }
    any_t * pop() {
        . . .
    }
    initialization_code() {
        . . .
    }
}
```

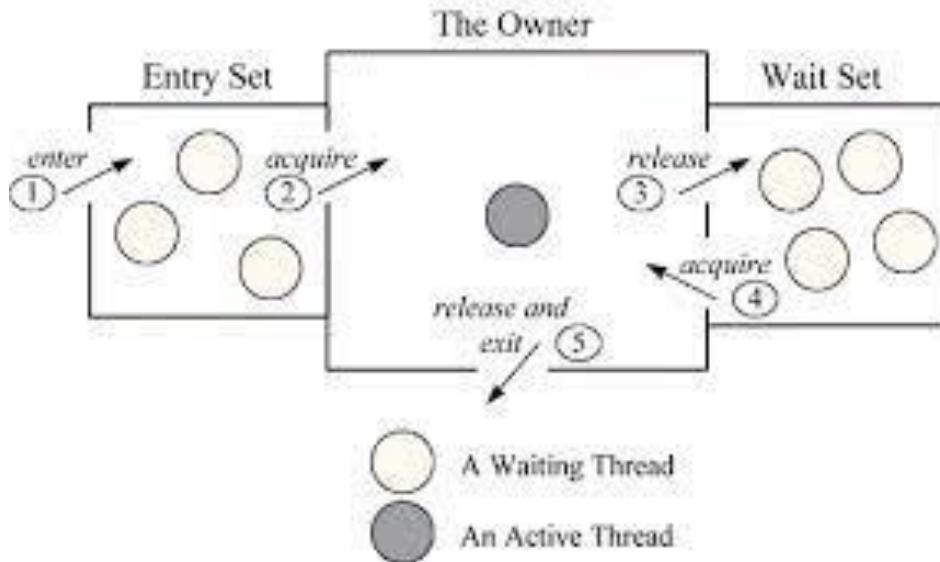


*/

A monitor is an object that is used as a mutually exclusive lock, or mutex.

Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.



Ex: Expected output of the below program is

[Hello]
[Synchronized]
[World]

By using 3 different threads, **without any synchronization** between them.

PGM NAME: NotSynch

```
class Callme
{
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {System.out.println("Interrupted");}
        System.out.println(" ]");
    }
}
```

```
class Caller implements Runnable {
    String msg;      // required to hold on to messages
    Callme target;  // target is the common object for three threads
                    // Method in this will display the expected output.
                    // Statements in this method must be executed in a synchronized manner,
```

```

        // or non-overlapping manner.
Thread t;      // object of Thread class required to call start() and run() method.

public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
}

public void run() {
    target.call(msg);
}
}

class NotSynch {
    public static void main(String args[]) {

        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

/* all threads are simultaneously trying to print the message onto screen which creates race condition*/
        // making the main thread wait for other threads to end its processes.

        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e)
        {System.out.println("Interrupted");}
    }
}

```

Output: *need not be the same output, one of the ways how output will be obtained.*

```

[Hello[Synchronized[World]
]
]
```

by calling sleep(), the call() method allows execution to switch to another thread, which results in the mixed-up output of the three message strings.

In this program, all three threads are calling **the same method, on the same object**, at the same time, termed as **race condition**, because the three threads are competing with each other to complete the process.

Using Synchronized Methods

Instances in Java have their own implicit monitor associated with them, hence it is easy to achieve synchronization in Java.

(Instances are the data source, which hold on to information. Synchronization or mutual exclusive access is required on data sources from methods.)

“**synchronized**” is the keyword used in java to achieve synchronization between several processes accessing the same data source.

When synchronized block is used, internally Java uses a monitor also known as monitor lock or intrinsic lock, to provide synchronization. These monitors are bound to an object, thus all synchronized blocks of the same object can have only one thread executing them at any point of time.

While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) **on the same instance** have to wait.

To exit the monitor and to give up control of the object to the next waiting thread, the owner of the monitor (which is a method which belongs to a thread) simply returns from the synchronized method.

In the program “NotSynch.java”, access to call() method must be **serialized or synchronized**, by restricting only one thread at a time. To do this, precede call()’s definition with the keyword **synchronized**

```
class Callme
{
    synchronized void call(String msg) // synchronized methods
    {
        ...
    }
}
```

Only one thread will be allowed to execute in synchronized method.

If a thread is executing a synchronized method, then no other thread can enter it until it stops executing that method.

(Even if the 1st thread that has begun execution and has entered into sleep mode pre-emption of it is not possible, until the thread itself will relinquish the access)

The synchronized Statement

Synchronized blocks/methods will not work in all scenarios.

Considering the method belongs to some other class, which is not editable or there is no access to source code, then attaching **synchronized** keyword for the method is not possible.

Solution to this is a synchronized block.

G.F of synchronized statement:

```
synchronized(object) {
    // statements to be synchronized
}
```

Here, object is a reference to the object being synchronized.

A synchronized block ensures that a call to a method (which is a member of) object occurs only after the current thread has successfully entered the object's monitor.

Ex: Using Synchronized statement

```
class Callme {  
    void call(String msg) {  
        System.out.print("[ " + msg);  
        try  
        { Thread.sleep(1000); }  
        catch (InterruptedException e)  
        {System.out.println("Interrupted");}  
  
        System.out.println("]");  
    } }  
  
class Caller implements Runnable {  
    String msg; Callme target; Thread t;  
  
    public Caller(Callme targ, String s) {  
        target = targ;  
        msg = s;  
        t = new Thread(this);  
        t.start();  
    }  
  
    // synchronize calls to call()  
    public void run() {  
        synchronized(target)  
        { // synchronized block  
            target.call(msg);  
        }  
    } }  
  
class Test {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
  
        // wait for threads to end  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } }
```

```

catch(InterruptedException e)
{System.out.println("Interrupted");}
}
}

```

Interthread/Interprocess Communication

Another way to achieve synchronization in java is by using interprocess communication.

Ex: Consider queuing problem, where one thread is producing some data and another is consuming it.

Word document can be considered as a producer and printer can be considered as consumer, where in the document will be printed.

Further, assume that document is ranging in terms of Mb size and pages together contents have to be printed, but the buffer associated with the printer is capable of holding on to only kb amount of information.

The producer has to wait until the consumer is finished consuming before it generates(copies) more data(Considering the buffer which is used between them is full).

In a polling system, the consumer would waste many CPU cycles while it was waiting for the producer to produce. Once the producer has finished, it would start polling, just to check whether the consumer has consumed the data, wasting more CPU cycles, and so on. Clearly, this situation is time consuming.

To avoid polling, Java has an interprocess communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final methods in Object**.

<https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

All three methods can be called only from within a synchronized context.

wait() tells the calling thread to give up the monitor and go to sleep until,
some other thread enters the same monitor and calls **notify()** method.

notify() wakes up a thread that called **wait()** on the same object.

notifyAll() wakes up all the threads that called **wait()** on the same object.
One of the threads will be granted access.

These methods are declared within Object, as shown here:

```
class Object {  
    final void wait() throws InterruptedException  
        wait( ) method is overloaded  
    final void notify()  
    final void notifyAll()  
}
```

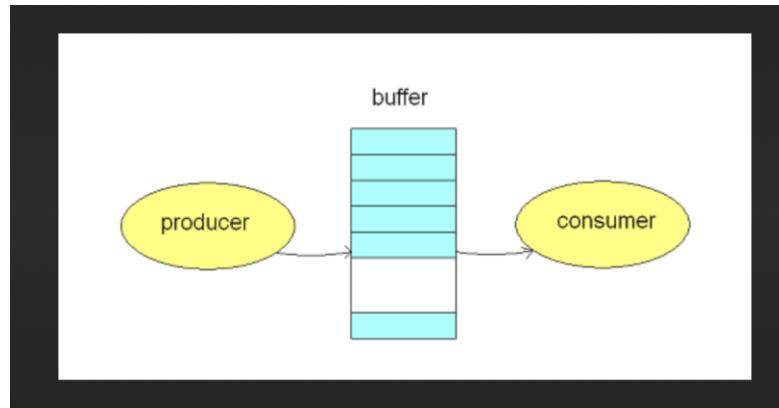
Alone wait() and notify() will not achieve exact synchronization in IPC.

Before calling wait() or notify() a condition has to be checked in a loop to achieve exact synchronization in IPC.

Ex: Simulating a producer and a consumer thread. Assumed that there is only one buffer between these two to hold on to only 1 information.

It is assumed that based on the capacity of the buffer, the producer can produce only 1 information and it has to wait until the consumer consumes this information before producing the next information.

Similarly, consumer can consume only one information, even if it has the capacity of consuming or using more than 1 information, because buffer size is limited to 1.



Ex:

Following program consists of four classes:

Q, the queue that is being synchronized;

Producer, the threaded object that is producing queue entries;

Consumer, the threaded object that is consuming queue entries; and

PC, class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer without proper IPC.
```

```
class Q {
```

```
    int n;
```

```
/*
```

“n” is a common variable between producer and consumer thread.

“n” must be perceived as a buffer.

Producer thread will fill a value to n.

Consumer process will pick a value from n.

get() method will be called by consumer thread to get the value stored in n.

put() method will be called by producer thread to put a value into n.

```
*/
```

```
synchronized void get() // used by the consumer process
```

```
{
```

```
    System.out.println("Got: " + n);
```

```
}
```

```
synchronized void put(int n) //used by the producer process
```

```
{
```

```
    this.n = n;
```

```
    System.out.println("Put: " + n);
```

```
}
```

```
}
```

```
//Producer thread
```

```
class Producer implements Runnable {
```

```
    Q q;
```

```
Producer(Q q)
```

```
{
```

```
    this.q = q;
```

```
    new Thread(this, "Producer").start();
```

```
}
```

```
public void run()
```

```
{
```

```
    int i = 0;
```

```
    while(true)
```

```
        q.put(i++);
```

```
    } }
```

```
//Consumer thread
```

```
class Consumer implements Runnable {
```

```
    Q q;
```

```
Consumer(Q q)
{
    this.q = q;
    new Thread(this, "Consumer").start();
}
```

```
public void run()
{
    while(true)
        q.get();
}
```

```
class Test {
    public static void main(String args[])
    {
        Q q = new Q();
        new Consumer(q);
        new Producer(q);

        System.out.println("Press Control-C to stop.");
    }
}
```

Output:

```
Put:1
Got:1
Got:1
Got:1
Got:1
```

```
/*
Just achieving synchronization between threads is not sufficient for producer-consumer problem, in addition to this, two conditions have to be checked for p-c problem to work in a competent manner.
```

1. First condition is w.r.t to consumer, whether buffer is filled before consumer consumes from it and
2. Second condition with respect to the producer will be that before the producer fills the buffer with information it is necessary that the buffer must be empty.

```
*/
```

```

// A correct implementation of a producer and consumer.
class Q {
    int n;
    volatile boolean valueSet = false;
/*
“valueSet” is a common variable used between producer and consumer process to know whether the
buffer is full or not.
*/
    synchronized void get()
    {
        while(!valueSet)
            try {
                wait();
            }
            catch(InterruptedException e)
            { System.out.println("InterruptedException caught"); }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
    }

    synchronized void put(int n)
    {
        while(valueSet)
            try {
                wait();
            }
            catch(InterruptedException e)
            { System.out.println("InterruptedException caught"); }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}//end of class Q

```

```

class producer implements Runnable
{
.....

```

MODULE 5

The Delegation Event Model

Events are handled in Java using a *delegation event model*.

Definition : EVENT

An *event* in Java is an object that is created, when changes are experienced in a graphical user interface.

Changes such as

- a user clicks on a button,
- clicks on a combo box, or
- types characters into a text field, etc.,

then an event triggers, creating the relevant event object.

This behavior is part of Java's Event Handling mechanism

(other properties of Java are

- Java is capable of handling Exceptions
- Java is capable of handling Garbage Collection
- etc.,)

Event handling is integral to the creation of swings and AWT type of GUI-based programs.

Events are supported by a number of packages, including java.util, java.awt and java.awt.event.

Words to ponder in DELEGATION-EVENT MODEL: EVENT, EVENT SOURCE, EVENT LISTENER

Delegation Event Model:

"An event source (GUI components) generates an event and sends it to one or more event listeners.

Listener waits until it receives an event.

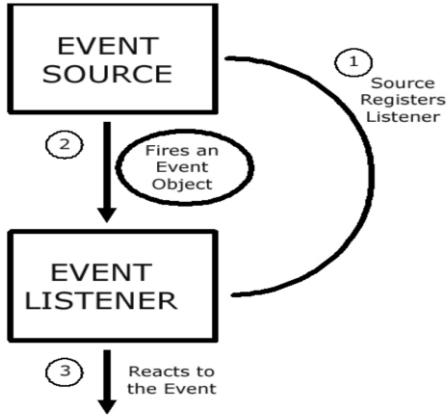
Once an event is received, the listener processes the event and then returns the result."

Advantage of this “DELEGATION-EVENT MODEL” is that the “application logic” that “processes events” is clearly separated from the “user interface logic” that “generates those events”.

A user interface element (GUI) is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, **listeners must register with a source** in order to receive an **event notification**.

Event notifications are sent only to listeners that have registered with the event source.
(callback interfaces)



Events

In the delegation model, an *event* is represented in the form of an object that describes a state change in a source (specifically a GUI application).

Ex: After typing google.com in a web browser, text box of google web page will be empty. (Event is yet to be generated).

As soon as the user types the keywords and press enter, an object containing all the necessary information, regarding the keyword search, including the keywords will be generated and passed to the nearest server.

Event is generated as a consequence of an user interacting with the elements in a GU interface.

Events may also occur that are **not directly caused by interactions with a user interface**.

Ex: an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Event Sources

Event source in a programming perspective is an object that holds all the necessary information about an event. This occurs when the internal state of that object changes in some way.

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:

```
public void add<Type>Listener(<Type>Listener el)
```

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

Ex: The method that registers a keyboard event listener is called **addKeyListener()**.

The method that registers a mouse motion listener is called **addMouseMotionListener()**.

When an event occurs, all registered listeners are notified and receive a copy of the “event object”. This is known as **multicasting** the event.

Notifications are sent only to listeners that register to receive them.

Some sources will allow only one listener to register.

GF:

```
public void add<Type>Listener(<Type>Listener el) throws  
    java.util.TooManyListenersException
```

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

When such an event occurs, the registered listener is notified. This is known as ***unicasting*** the event.

/*

A source will also provide a method that allows a listener to unregister an interest in a specific type of event.

GF: `public void remove<Type>Listener(<Type>Listener el)`

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

For example, to remove a keyboard listener, you would call `removeKeyListener()`.

*/

Methods that add or remove listeners are provided by the source that generates events.

Below table lists some of the **user interface components/graphical user interface elements that can generate the 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; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-2 Event Source Examples

Event Classes

When an event occurs, all its information will be encapsulated within an instance of Event class type.

Classes that represent events are at the core of Java's event handling mechanism.

Most widely used events are defined by the AWT(Abstract Window Toolkit) and Swing.

/*

*At the root of the Java event class hierarchy is **EventObject**, which is in `java.util`. It is the superclass for all events.*

EventObject class contains these methods:

```
Object getSource(); //returns the source of the event.  
String toString(); //returns the string equivalent of the event.  
int getID(); //returns event type
```

Class **AWTEvent**, defined within the `java.awt` package, is a subclass of **EventObject**.

AWTEvent is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

*/

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

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-1 Main Event Classes in `java.awt.event`

An instance of above classes will be generated by event source automatically and passed to event listeners.

Programmers need not worry about generating instances of these classes, it will be done automatically by JVM.

The Programmer's job will be to register a method as a listener and provide the necessary background logical actions for the event.

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**, and **SHIFT_MASK**.

In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

Constructors of ActionEvent class // used only by event source

```
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)
```

src : is a reference to the object that generated this event.

type : type of the event is specified by type.

modifier : indicates which modifier keys such as ALT, CTRL, META and/or SHIFT were pressed when the event was generated.

when: parameter specifies when the event occurred.

String getActionCommand() //used by event listeners

returns Command name for the invoking ActionEvent object

Ex: when a button is pressed, an action event is generated that has a command name equal to the label on that button.

int getModifiers() //used by event listeners

returns a value that indicates which modifier keys (ALT, CTRL,META, and/or SHIFT) were pressed when the event was generated.

long getWhen() //used by event listeners

returns the time at which the event took place. This is called the event's *timestamp*.

The AdjustmentEvent Class

The ContainerEvent Class

The InputEvent Class

The KeyEvent Class

The MouseWheelEvent Class

The WindowEvent Class

The ComponentEvent Class

The FocusEvent Class

The ItemEvent Class

The MouseEvent Class

The TextEvent Class

Event Listeners

/*

Definition : EVENT LISTENER

An *event listener* is a **procedure** or **function** in a computer program that waits for an *event* to occur. The listener is programmed to react to an *input* or signal by calling the event's handler.



*/

An *Event listener* is an object that is notified when an event occurs.
(Remember even *Event source* is an object, which generates events)

It has two major requirements.

First, it must have been registered with one or more sources to receive notifications about specific types of events.

Second, it must implement methods to process these notifications/requests, which will logically modify the received data from the event source.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.

Ex: **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Listener Interfaces

Delegation event model has two parts: **source and listener**.

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Table lists commonly used listener interfaces and provides a brief description of the methods that they define.

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

TABLE 22-3 Commonly Used Event Listener Interfaces

ActionListener Interface

defines the **actionPerformed()** method that is invoked when an action event occurs.

```
void actionPerformed (ActionEvent ae)
```

AdjustmentListener Interface

defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs.

```
void adjustmentValueChanged (AdjustmentEvent ae)
```

.....

KeyListener Interface

defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively.

The **keyTyped()** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released.

If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

```
void keyPressed (KeyEvent ke)
void keyReleased (KeyEvent ke)
void keyTyped (KeyEvent ke)
```

MouseListener Interface

defines five methods.

If the mouse is pressed and released at the same point, **mouseClicked()** is invoked.

When the mouse enters a component, the `mouseEntered()` method is called.

When it leaves, `mouseExited()` is called.

The `mousePressed()` and `mouseReleased()` methods are invoked when the mouse is pressed and released, respectively.

```
void mouseClicked (MouseEvent me)
void mouseEntered (MouseEvent me)
void mouseExited (MouseEvent me)
void mousePressed (MouseEvent me)
void mouseReleased (MouseEvent me)
```

MouseListener Interface

This interface defines two methods.

```
void mouseDragged (MouseEvent me)
void mouseMoved (MouseEvent me)
```

The `mouseDragged()` method is called multiple times as the mouse is dragged.

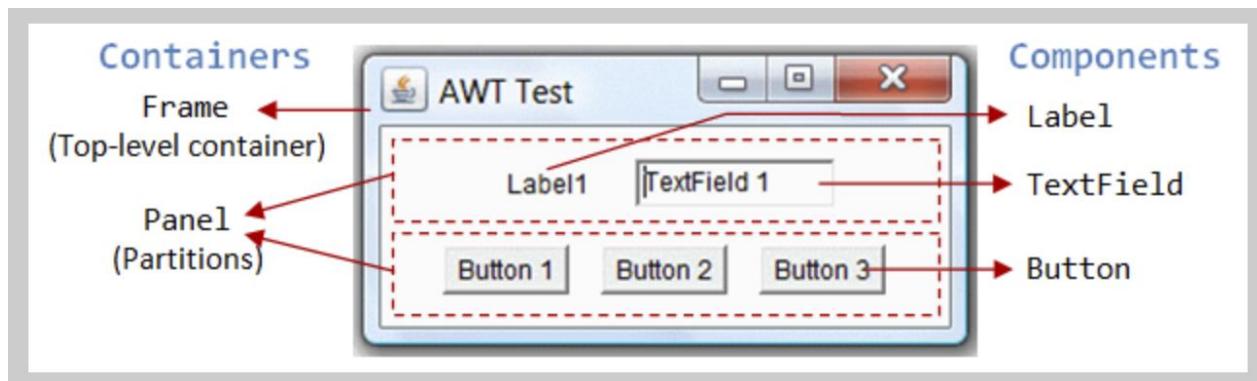
The `mouseMoved()` method is called multiple times as the mouse is moved.

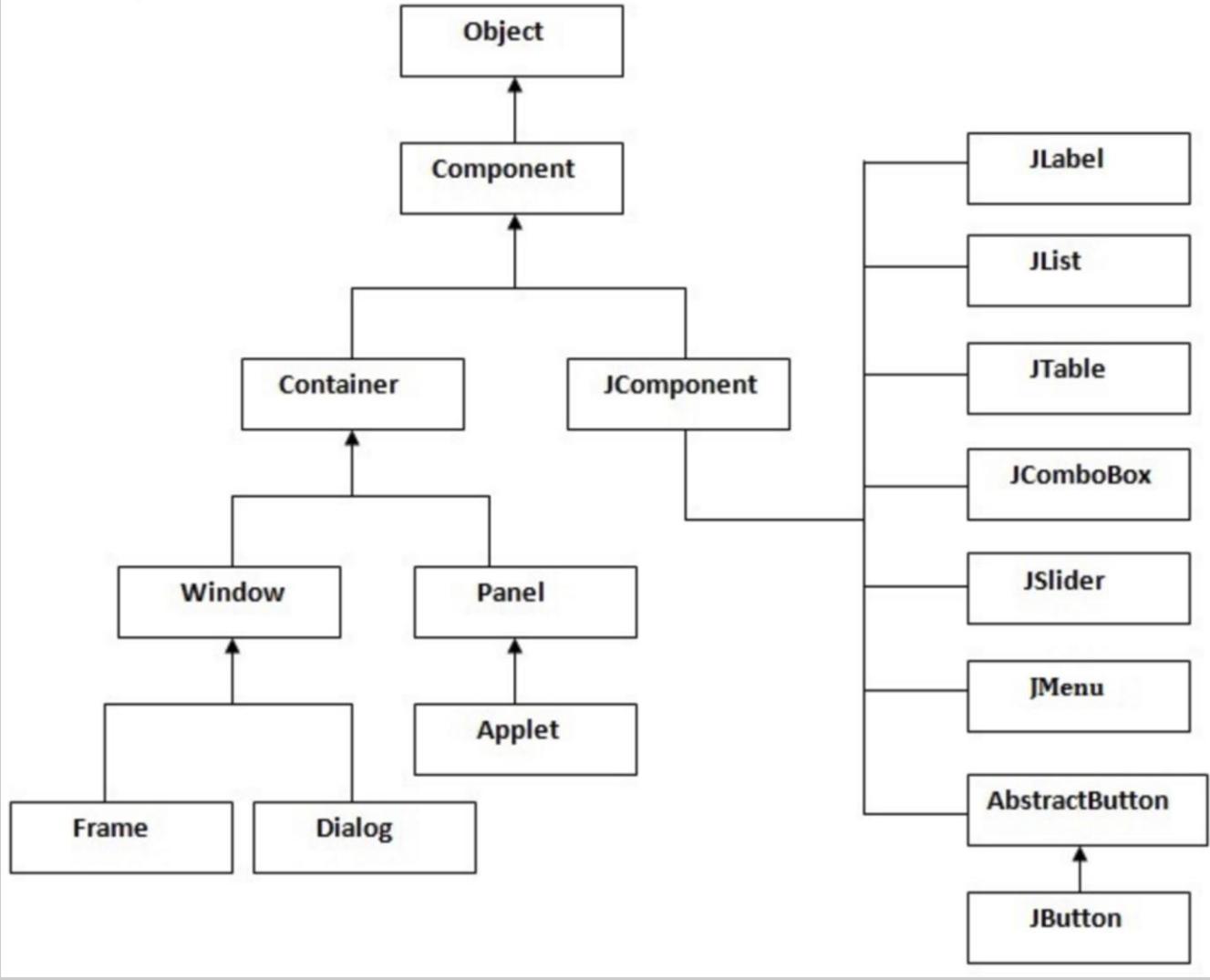
/* POINTS TO PONDER

A single GUI component like button, or text box or label will be part of a container like Frame

GUI components like buttons, textbox or labels are Event sources.

But since, these components are stored or added to a container **Frame acts as both an event source and an event listener.***





Ex: HANDLING MOUSE EVENTS

```
import java.awt.*; //1
import java.awt.event.*; //2
//GUI is both event source and event listener
public class GUI extends Frame implements MouseListener, MouseMotionListener
{
    Label l; //3
    Label l1,l2;

    int mx=0,my=0;

    GUI(){
        addMouseListener(this); //4
        addMouseMotionListener(this); //4

        l=new Label();
        l.setBounds(10,10,100,150); //5
        add(l); //6

        l1 = new Label();
        l1.setBounds(10,90,100,150);
        add(l1);

        l2 = new Label();
        l2.setBounds(10,180,100,150);
        add(l2);

        setSize(400,400); //7
        setLayout(null); //8
        setVisible(true); //9
    }
}
```

```

// All these methods are called implicitly by the event source.

public void mouseClicked(MouseEvent e)
{
    l.setText("Mouse Clicked");
    System.out.println(e.getX() + " " + e.getY());
}

public void mouseEntered(MouseEvent e) {
    l.setText("Mouse Entered");
}

public void mouseExited(MouseEvent e) {
    l.setText("Mouse Exited");
}

public void mousePressed(MouseEvent e) {
    l.setText("Mouse Pressed");
}

public void mouseReleased(MouseEvent e) {
    l2.setText("Mouse Released");
}

public void mouseDragged(MouseEvent e){
    l1.setText("Mouse Dragged");
}

public void mouseMoved(MouseEvent e) {
    l1.setText("Mouse Moved");
}

public static void main(String[] args) {
    new GUI();
}
/*

```

1. adds only those classes that are present in awt sub-directory.

2. event is a sub-package and all the classes in event is added to the program

3. The object of Label class, which is a GUI 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 an application but a user cannot edit it directly.

```

(IH of Label)
java.lang.Object
    java.awt.Component
        java.awt.Label //Label is derived class of Component

```

4. Frame is both the source and the listener for these events.

the Frame registers itself as a listener for mouse events.

This is done by using `addMouseListener()` and `addMouseMotionListener()`, which are members of Component.

```
void addMouseListener (MouseListener ml)
void addMouseMotionListener (MouseMotionListener mml)
```

5. `public void setBounds(int x, int y, int width, int height)`

Moves and resizes this component. The new location from

top-left corner is specified by x and y, and
the new size is specified by width and height.

6. `public Component add(Component comp)`

Appends the specified component to the end of this container.

Parameters:

`comp` - the component to be added

Returns:

the component argument

Throws:

`NullPointerException` - if `comp` is null

Class Frame

INHERITANCE HIERARCHY

`java.lang.Object`

`java.awt.Component` // `add(Component comp)` method belongs to this class

`java.awt.Container`

`java.awt.Window`

`java.awt.Frame`

7. `public void setSize(int w, int h)` belongs to Component class

Resizes this component so that it has width w and height h.

Parameters:

`w` - the new width of this component in pixels

`h` - the new height of this component in pixels

User defined class GUI does not have `setSize` method but its base class Frame has it, which is inherited by Component class.

8. `java.awt.Container` contains `setLayout` method

The Layout managers enable programmer to control the way in which visual components

are arranged in the GUI forms by determining the size and position of components within the containers.

There are different built-in classes for layout managers in java.

9. public void setVisible(boolean b) java.awt.Window class contains this method
Shows or hides this Window depending on the value of parameter b.

*/

Ex: HANDLING KEYBOARD EVENTS

```
import java.awt.*;
import java.awt.event.*;
public class GUI extends Frame implements KeyListener
{
    Label l,l1;
    String msg ="";
```

```
GUI(){
    addKeyListener(this);
```

```
l=new Label();
l.setBounds(10,10,100,150);
add(l);
```

```
l1 = new Label();
l1.setBounds(10,90,200,400);
add(l1);
```

```
setSize(400,400);
setLayout(null);
setVisible(true);
```

```
}
```

```
public void keyPressed(KeyEvent ke) {
    l.setText("Key Down");
    int key = ke.getKeyCode();
    switch(key) {
        case KeyEvent.VK_F1:
```

//Virtual key codes present in KeyEvent class Pg no. 645 text book

```
        msg += "<F1>";
        break;
```

```
        case KeyEvent.VK_F2:
```

```
            msg += "<F2>";
            break;
```

```
        case KeyEvent.VK_F3:
```

```

msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;
case KeyEvent.VK_RIGHT:
msg += "<Right Arrow>";
break; }
l1.setText(msg);
}

public void keyReleased(KeyEvent ke) {
l1.setText("KeyReleased");
}

public void keyTyped(KeyEvent ke) {
msg = msg + ke.getKeyChar();
l1.setText(msg);
}

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

```

Adapter Classes

Adapter classes are used to simplify the creation task of event handlers.

Ex: GUI class is an Event handler it has registered for KeyListener. Since, it has registered for KeyListener, it has to implement all the methods that are declared in the KeyListener interface. After implementing KeyListener, if GUI class wants to implement only KeyTyped method it is not allowed.

An adapter class provides **an empty implementation** for all the methods in an event listener interface.

Adapter classes are useful only when some of the events are supposed to be received and processed by a particular event listener interface.

Adapter classes can be inherited by the event handler classes (like GUI) and only those methods that are required can be implemented.

Ex:

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

class frame extends Frame //Event Source
{
    protected Label l; // Label is G-component. Frame is a container

frame(){
    l=new Label();
    l.setBounds(10,10,100,150);
    add(l);

    //Event register method
    addMouseMotionListener(new GUI(this)); //1

    setSize(400,400);
    setLayout(null);
    setVisible(true);
}

}

//GUI is considered as listener class
public class GUI extends MouseMotionAdapter // Event Listener
{
    frame fr;
    GUI(frame f){
        fr = f;
    }

    public void mouseDragged(MouseEvent me) {
        fr.l.setText("Mouse Dragged");
    }
}

public static void main(String[] args) {
    new frame();
}
/*

```

1. Since, java allows only one class to extend, both Frame and MouseMotionAdapter cannot be extended to a single class.

"frame" class extends Frame and "GUI" extends MouseMotionAdapter class.

Frame is the component which generates an event and also the component which responds to the event (Event listener).

Registering for the event listener has to be done in frame class only.

GUI class implements mouseDragged method hence addMouseMotionListener is passed as an instance of GUI.

addMouseMotionListener cannot be called in GUI class because it is present in Component class whose properties are inherited by Frame class and MouseMotionAdapter has not inherited properties of the Component class.

**public abstract class MouseMotionAdapter extends Object
implements MouseMotionListener**

*/

TABLE 22-4
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

All adapter classes are by default abstract classes, in which the methods of respective interfaces are null defined. The methods in this class are empty.

**public abstract class MouseMotionAdapter extends Object
implements MouseMotionListener**

In other words definitions of compulsory/mandatory methods of implemented interface will not have any statements within it.

Inner Classes

“inner class” is a class defined within another class, or even within an expression.

Use of inner class will simplify the code using event adapter classes.

Ex:

```
import java.awt.*;
import java.awt.event.*;
class frame extends Frame //Event source
{
    protected Label l;

    frame(){
        l=new Label();
        l.setBounds(10,10,100,150);
        add(l);

        //Event register method
        addMouseMotionListener(new gui());

        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
    class gui extends MouseMotionAdapter // INNER CLASS event listener
    {
        public void mouseDragged(MouseEvent me)
        {
            l.setText("Mouse Dragged");
        }
    }// end of class gui
} // end of class frame
public class GUI
{
    public static void main(String[] args) {
        new frame();
    }
}
```

Swings (GUI components in swings are not thread safe)

Swing is a set of classes that provides more powerful and flexible GUI components than AWT.

AWT graphical components are considered as *heavy weight components*.

```
/*
AWT Graphical components look and feel is fixed. Look and feel of a component is defined by the
platform.
*/
```

Swing eliminates a number of the limitations inherent in the AWT.

Swing is built on the foundation of the AWT.

Swing also uses the same event handling mechanism as the AWT.

Two Key Swing Features

Lightweight components and

Pluggable look and feel.

“Look” refers to the appearance of GUI widgets and **“feel”** refers to the way the widgets behave.
“Pluggable” the look and feel that can be changed during runtime.

Swing Components Are Lightweight

Swing components are written entirely in Java and do not map directly to platform-specific peers.

Because lightweight components are rendered using graphics primitives (like line drawing, circle drawing and different graphical drawing algorithms), they can be transparent, which enables non rectangular shapes.

“Non-rectangular shapes” : Normally Dialog Boxes are rectangular in shape. Various methods can be adopted to make them non-rectangular in shape.

Hence, lightweight components are more efficient and more flexible.

Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.

This means that each component will work in a consistent manner across all platforms.

```
/*
```

Swing Supports a Pluggable Look and Feel (PLAF)

Since, each Swing component is rendered by Java code rather than by native code, the look and feel of a component is **under the control of Swing**.

Swing separates the look and feel of a component from the logic of the component.

Since, swing separates the look and feel of a component, it is possible to change the way that a component is rendered without affecting any of its other aspects.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms.

Conversely, it is possible to create a look and feel that acts wrt a specific platform.

Ex: if an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.

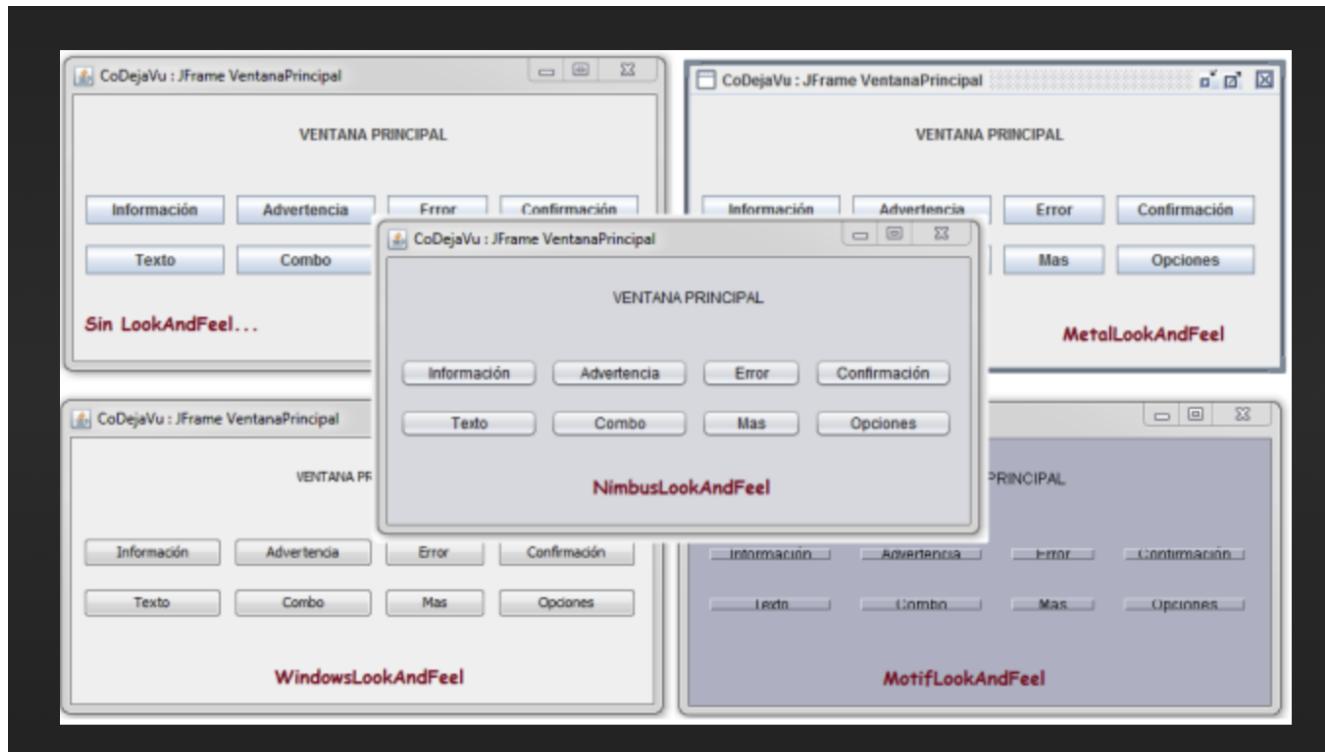
*/

The look and feel can be changed dynamically at run time.

Java SE 6 provides look-and-feels, such as **metal** and **Motif**, that are available to all Swing users.

The metal look and feel is also called the *Java look and feel*.

It is platform-independent and available in all Java execution environments. It is also the default look and feel.



For the Swing program metal look-and-feel will be considered since it is platform independent.

The MVC Connection

In general, a visual/Graphical component is a composite of three distinct aspects:

The way that the **component looks** when rendered on the screen

The way that the **component reacts** to the user

The **state information** associated with the component

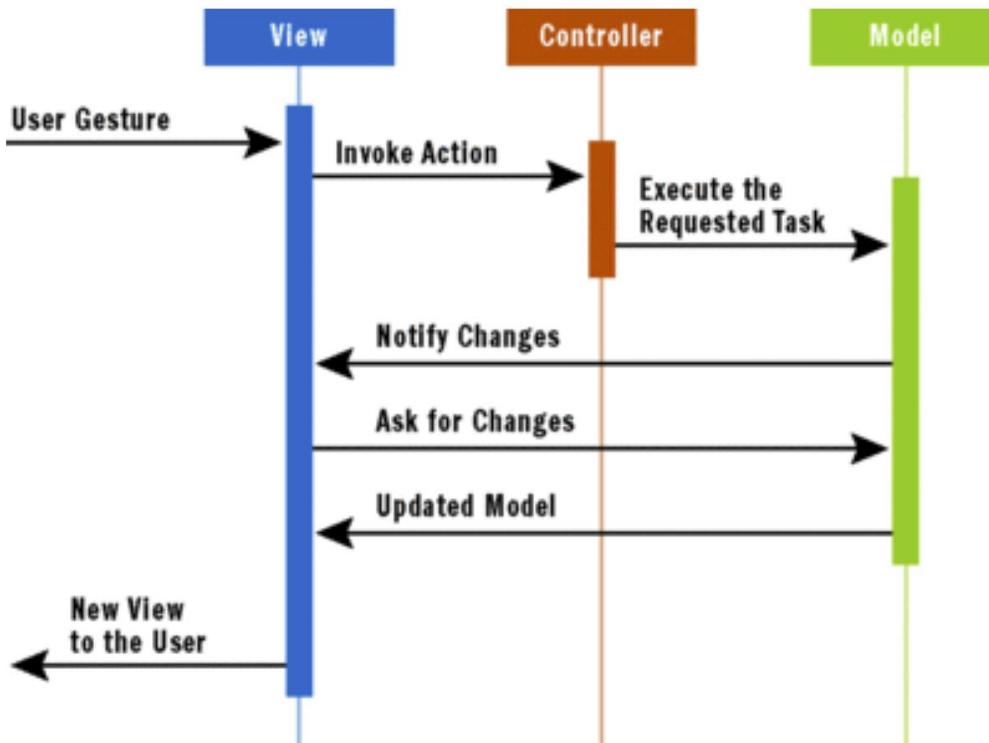
MVC means the “**logic that interacts with data**” and “**user interface that represent data**” must be separate entities.

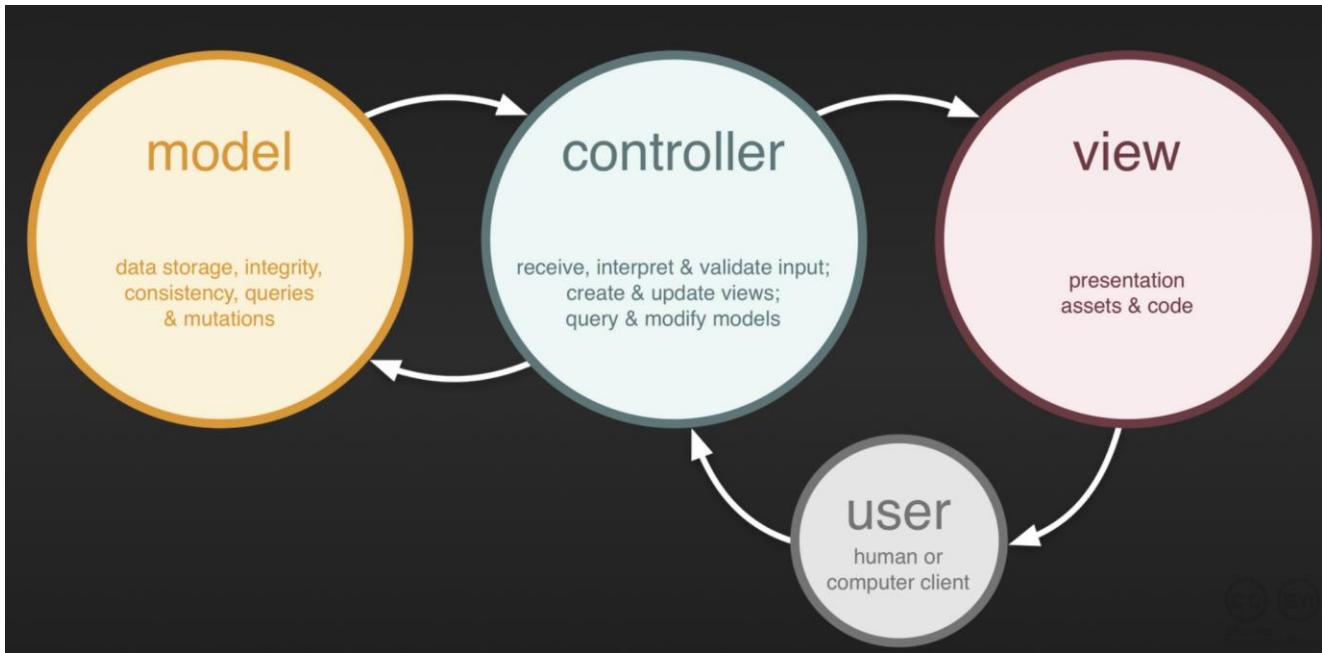
In MVC terminology, the **model** corresponds to the state information associated with the component.
Ex: in case of a check box, the model contains a field/data member that indicates if the box is checked or unchecked.

View determines how the component is displayed on the screen.

Controller determines how the component reacts to the user.

Ex: when the user clicks a checkbox, the controller reacts by changing the model to reflect the user’s choice (checked or unchecked). Which then results in the view being updated.





Ex:

In restaurant example, customer is the view, waiter is the controller and the chef is the model.

Data in this case is equal to the ingredients of the food.

Chef or the cook has to prepare them, just like the model retrieves and processes data from persistent storage.

Waiter is not going to prepare the food like the controller does not communicate with the database (also both the controller and the view), in restaurant example the waiter and customer doesn't know anything about the process of the cook.

Swing uses a modified version of MVC that **combines** the **view** and the **controller** into a single logical entity called the **UI delegate**.

Hence, Swings are termed as **MODEL-DELEGATE / SEPARABLE MODEL** architecture.

To support the Model-Delegate architecture, most **Swing components contain two objects**.

First represents the model.

Second represents UI delegate.

Models are defined by interfaces. (Models are similar to listeners)

Ex: model for a button is defined by the **ButtonModel** interface.

UI delegates are classes that inherit **ComponentUI**.

Ex: UI delegate for a button is **ButtonUI**.

Difference between AWT and Swing

Java AWT	Java Swing
AWT components are platform-dependent	Swing components are platform-independent
AWT components are heavy weight	Swing components are lightweight
AWT does not provide pluggable look and feel	Swing supports pluggable look and feel
AWT provides less GUI components than swing	Swing provides more number of GUI components
AWT does not follow MVC (Model View Controller) Where model represents data View represents presentation Controller acts as an interface between model and view	Swing follows MVC

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*.

A *component* is an independent visual control, such as a push button or slider.

A container is a special type of component that is designed to hold other components.

Further, 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.

Containers are also considered as components, a container can also hold other containers.

Swing defines a *containment hierarchy*, at the top of which must be a *top-level container*.

Ex: JFrame, JApplet, JDialo g etc.,

Components

Swing components are derived from the **JComponent** class.

JComponent provides the functionality that is common to all components.

All of Swing's components are represented by classes defined within the package **javax.swing**.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JToggleButton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Table: Swing Components and Containers.

/*Containers

Swing defines two types of containers.

The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog** (heavy weight).
These containers do not inherit JComponent.
They inherit the AWT classes Component and Container.

As the name implies, a top-level container must be at the top of a containment hierarchy.

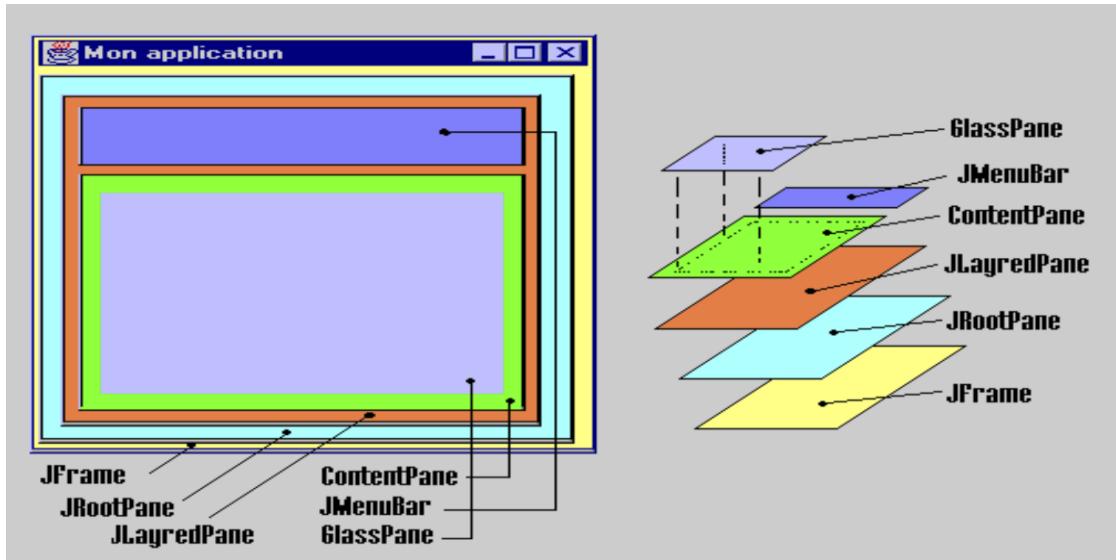
A top-level container is not contained within any other container.
Ex: most commonly used container for applications is **JFrame**.

The second type of containers supported by Swing are lightweight containers.
Lightweight 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 because a lightweight container can be contained within another container.

Thus, **JPanel** is used to create subgroups of related controls that are contained within an outer container.*/

The Top-Level Container Panes



The pane with which application will interact the most is the content pane, because this is the pane to which visual components will be added.

Ex: when a component, such as a button is added, to a top-level container, it will be added to the content pane. By default, the content pane is an **JPanel**.

A SIMPLE SWING APPLICATIONS

```
import javax.swing.*;  
class SwingDemo {  
    SwingDemo() {  
  
        JFrame jfrm = new JFrame("A Simple Swing Application");  
  
        jfrm.setSize(275, 100);  
  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        JLabel jlab = new JLabel(" Swing means powerful GUIs.");  
  
        jfrm.add(jlab);  
  
        jfrm.setVisible(true);  
    }  
}
```

```

public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
        {
            new SwingDemo();
        }
    } // end of anonymous class
}); // end of creation of instance
}

```

Output of the program



The program imports javax.swing, which contains components and models defined by Swing.
Ex: javax.swing defines classes that implement labels, buttons, text controls, and menus.

It will be included in all programs that use Swing.

First statement is constructor of “SwingDemo” user defined class is

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

This creates a **container** called jfrm that defines a rectangular window complete with a title bar;
close,
minimize,
maximize, and
restore buttons; and
a system menu.

It creates a standard, top-level window. **The title of the window is passed to the constructor.**

Next, the window is sized using this statement:

```
jfrm.setSize(275, 100); //void setSize(int width, int height)
```

`setSize()` method (which is inherited by `JFrame` from the AWT class `Component`) sets the dimensions of the window, which are specified in pixels.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated, which is the default behavior.

In some situations, when a top level window is closed the entire application has to terminate, which is done by calling `setDefaultCloseOperation()` as shown.

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Since, the above statement is used in the program closing the window causes the entire application to terminate.

GF: **void setDefaultCloseOperation(int what)**

The value passed in *what* determines what happens when the window is closed.

```
JFrame.EXIT_ON_CLOSE  
JFrame.DISPOSE_ON_CLOSE  
JFrame.HIDE_ON_CLOSE  
JFrame.DO NOTHING_ON_CLOSE
```

The next statement creates a Swing `JLabel` component:

```
JLabel jlab = new JLabel("Swing means powerful GUIs.");
```

`JLabel` displays information, which can consist of text, an icon, or a combination of the two.

The label created by the program contains only text, which is passed to its constructor.

Next statement is

```
jfrm.add(jlab); //Component add(Component comp)
```

All top-level containers have a content pane in which components are stored.

Adding a component to a frame's content pane is done by calling `add()` on `JFrame` reference i.e `jfrm`.

The last statement is: `jfrm.setVisible(true);`

If its argument is **true**, the window will be displayed. Otherwise, it will be hidden.

Anonymous inner class

<https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html>

Inside `main()`, a `SwingDemo` object is created, which causes the window and the label to be displayed.

```
SwingUtilities.invokeLater(new Runnable()  
{  
    public void run()  
    {  
        new SwingDemo();  
    }  
});
```

This sequence causes a `SwingDemo` object to be created on the *event dispatching thread* rather than on the *main thread of the application*.

Swing programs are event-driven.

Ex: when a user interacts with a component, an event is generated.

An event is passed to the application by calling an event handler defined by the application.

Event handler is executed on the “**event dispatching thread (EDT)**” provided by Swing and not on the main thread of the application.

//Below URL's answers why swing components must be executed in EDT
<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>

<https://javarevisited.blogspot.com/2013/08/why-swing-is-not-thread-safe-in-java-Swingworker-Event-thread.html>

Below lines are available in the above link.

“Since EDT thread is most important in Swing and responsible for listening event and updating GUI, you must not do any time consuming task on it, otherwise, you risk your application to become unresponsive or frozen. Another key thing to note, while using multi-threading in Swing development is that, not only GUI components but there model e.g. TableModel for JTable, must be updated in Event Dispatcher thread. One of the crucial [difference between AWT and Swing](#) is that AWT components are thread-safe.”

(***)To execute GUI code on the **event dispatching thread**, one of the two methods that are defined by the **SwingUtilities** class, are supposed to be called.

static void invokeLater(Runnable obj)

static void invokeAndWait(Runnable obj) throws InterruptedException, InvocationTargetException

obj is a **Runnable** object that will have its **run()** method called by the event dispatching thread.

The difference between the two methods is that **invokeLater()** returns immediately, but **invokeAndWait()** waits until **obj.run()** returns.

/*
invokeAndWait() method in swing is **synchronous**.
It blocks until the Runnable task is complete.

InvokeLater() method in swing is **asynchronous**.

It posts an action event to the event queue and returns immediately. It will not wait for the task to complete
*/

Ex: Same program written using an intermediate class

```
import javax.swing.*;
class gui {
    gui() {

        JFrame jfrm = new JFrame("A Simple Swing Application");

        jfrm.setSize(275, 100);

        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel jlab = new JLabel(" Swing means powerful GUIs.");

        jfrm.add(jlab);

        jfrm.setVisible(true);
    }
}

class inter implements Runnable
{
    Thread t = new Thread(this);

    public inter()
    {
        t.start();
    }

    public void run()
    {
        new gui();
    }
}

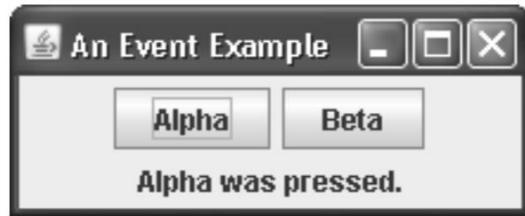
class test {
    public static void main(String args[])
    {
        new inter();
    }
}

OR

SwingUtilities.invokeLater(new inter());
}
```

Event Handling

The following program handles the event generated by a Swing push button.



```
//Handle an event in a Swing program.  
import java.awt.*; // because "FlowLayout" is present in this package  
import java.awt.event.*;  
import javax.swing.*;  
class EventDemo {  
JLabel jlab;  
EventDemo() {  
    // Create a new JFrame container.  
    JFrame jfrm = new JFrame("An Event Example");  
    // Specify FlowLayout for the layout manager.  
    jfrm.setLayout(new FlowLayout());  
    /*  
https://docs.oracle.com/javase/7/docs/api/java/awt/FlowLayout.html  
Reg why FlowLayout is necessary is explained in the above url  
*/  
    // Give the frame an initial size.  
    jfrm.setSize(220, 90);  
  
    // Terminate the program when the user closes the application.  
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    // Make two buttons.  
    JButton jbtnAlpha = new JButton("Alpha");  
    JButton jbtnBeta = new JButton("Beta");  
}
```

```

// Add action listener for Alpha.
/https://docs.oracle.com/javase/7/docs/api/java.awt.event ActionListener.html
/*
Adds the specified action listener to receive action events from this button. Action events occur
when a user presses or releases the mouse over this button
*/
jbtnAlpha.addActionListener (new ActionListener()
{
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});

// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);

// Create a text-based label.
jlab = new JLabel("Press a button.");

// Add the label to the content pane.
jfrm.add(jlab);

// Display the frame.
jfrm.setVisible(true);
}

public static void main(String args[]) {
// Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable()
{
    public void run()
    {
        new EventDemo();
    }
});
}

```

```
} }
```

The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame.

The **java.awt.event** package is needed because it has an **ActionListener** interface and **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**.

Then, **EventDemo()** creates two push buttons, as shown here:

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

```
JButton(String msg)
```

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent** instance.

JButton provides the **addActionListener()** method, which is used to add an action listener.

ActionListener interface defines only one method:

```
void actionPerformed(ActionEvent ae)
```

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button's action events are added by the code shown here:

```
// Add action listener for Alpha.  
jbtnAlpha.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent ae)  
    {  
        jlab.setText("Alpha was pressed.");  
  
    }  
});  
  
// Add action listener for Beta.  
jbtnBeta.addActionListener(new ActionListener() {  
  
    public void actionPerformed(ActionEvent ae)  
    {  
        jlab.setText("Beta was pressed.");  
  
    }  
});
```

Anonymous inner classes are used to provide the event handlers for the two buttons.

Each time a button is pressed, the string displayed in the jlab is changed to reflect which button was pressed.

Next, the buttons are added to the content pane of **jfrm**:
`jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);`

Finally, **jlab** is added to the content pane and window is made visible.

When the program is executed, each time a button is pressed, a message is displayed in the label that indicates which button was pressed.

IMP: All event handlers, such as actionPerformed(), are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If the application needs to do something time consuming as the result of an event, it must use a separate thread.

Create a Swing Applet

Applet definition

<https://docs.oracle.com/javase/tutorial/deployment/applet/index.html>

And

Life cycle of the applet

<https://docs.oracle.com/javase/tutorial/deployment/applet/lifeCycle.html>

Swing-based applets are similar to AWT-based applets, but with an important difference:
A Swing applet extends **JApplet** rather than **Applet**.

JApplet is derived from **Applet**.

Swing applets use four lifecycle methods: **init()**, **start()**, **stop()**, and **destroy()**.
Only the required methods will be overridden in the application program.



Ex:

```
//A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/* THIS HTML CODE IS NEEDED IF THIS PROGRAM IS EXECUTED IN TERMINAL MODE.
<object code="MySwingApplet" width=220 height=90>
</object>
*/
public class Test extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;
    JLabel jlab;
    // Initialize the applet.
    @Override
    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable ()
            {
                public void run()
                {
                    makeGUI(); // initialize the GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of "+ exc);
        }
    }
}
```

```

// This applet does not need to override start(), stop(), or destroy().
// Set up and initialize the GUI.
private void makeGUI() {
    // Set the applet to use flow layout.
    setLayout(new FlowLayout());
    // Make two buttons.
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent le)
        {
            jlab.setText("Alpha was pressed.");
        }
    });
}

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent le)
    {
        jlab.setText("Beta was pressed.");
    }
});

// Add the buttons to the content pane.
add(jbtnAlpha);
add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
add(jlab);
}
}

```

Test extends **JApplet**. All Swing-based applets extend **JApplet** rather than **Applet**.

init() method initializes the Swing components on the event dispatching thread by setting up a call to **makeGUI()**. This is accomplished through the use of **invokeAndWait()** rather than **invokeLater()**.

Applets must use **invokeAndWait()** because the **init()** method must not return until the entire initialization process has been completed.

Inside **makeGUI()**, the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane.

Exploring Swings

Two types of programs can be created using Swings in java

1. Swing application
2. Swing Applet

Swing Component classes are as follows.

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, and they are all derived from **JComponent**.

The Swing components are used in Swing applets rather than swing applications.

JLabel and ImageIcon

JLabel creates a label.

JLabel can be used to display text and/or an icon.

It is a passive component in that it does not respond to user input.

JLabel defines several constructors.

JLabel(Icon icon)

JLabel(String str)

JLabel(String str, Icon icon, int align)

str and *icon* are the text and icon used for the label.

align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label.

align value must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING.

Icons are specified by objects of type Icon, which is an interface defined by Swing.

ImageIcon class can be used to obtain an icon. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor.

There are several ways to provide the image, including reading it from a file or downloading it from a URL. ImageIcon constructor is as follows

ImageIcon(String filename)

It obtains the image in the file named *filename*.

The icon and text associated with the label can be obtained by the following methods:

Icon getIcon()
String getText()

The icon and text associated with a label can be set by these methods:

void setIcon(Icon icon)
void setText(String str)

Here, *icon* and *str* are the icon and text, respectively.

Therefore, using `setText()` it is possible to change the text inside a label during program execution.

Ex:

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

```
//JApplet is deprecated  
public class Test extends JApplet {
```

```
public void init() {  
    try {  
        SwingUtilities.invokeAndWait( new Runnable()  
        {  
            public void run()  
            {  
                makeGUI();  
            }  
        }  
    );  
    }  
    catch (Exception exc)  
    { System.out.println("Can't create because of " + exc);}  
}
```

```
private void makeGUI() {  
  
    // Create an icon.  
    ImageIcon ii = new  
        ImageIcon("/Users/hemanth//eclipse-workspace//hellot//src//hello//india.gif");  
  
    // Create a label.  
    JLabel jl = new JLabel("India", ii, JLabel.CENTER);  
  
    // Add the label to the content pane.  
    add(jl); }  
}
```

JTextField

How to use Text fields in swings in java

<https://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html>

<https://docs.oracle.com/javase/8/docs/api/javax/swing/JTextField.html>

“..... *VK_ENTER* results in the listeners getting an *ActionEvent*, and the *VK_ENTER* event is consumed. This is compatible with how AWT text fields handle *VK_ENTER* events.”

JTextField allows users to edit one line of text.

It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components.

Three of **JTextField**’s constructors are shown here:

```
JTextField(int cols)  
JTextField(String str, int cols)  
JTextField(String str)
```

str is the string to be initially presented *cols* is the number of columns in the text field.

If no string is specified, the text field is initially empty.

If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user interaction.

ActionEvent is fired when the user presses ENTER.

CaretEvent is fired each time the caret (i.e., the cursor) changes position.

To obtain the text currently in the text field, call **getText()**.

Ex:

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

```
//Demonstrate JTextField.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet {  
    JTextField jtf;  
    public void init() {  
        try {  
            SwingUtilities.invokeAndWait(new Runnable() {  
                public void run() {  
                    makeGUI();  
                }  
            }  
        );  
    }  
    catch (Exception exc) {System.out.println("Can't create because of " + exc);}  
}  
  
private void makeGUI() {  
    // Change to flow layout.  
    setLayout(new FlowLayout());  
  
    // Add text field to content pane.
```

```
jtf = new JTextField(15);  
add(jtf);  
  
jtf.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent ae) {  
        // Show text when user presses ENTER.  
        showStatus(jtf.getText());  
    } });
```

<https://docs.oracle.com/javase/8/docs/api/javax/swing/event/CaretListener.html#caretUpdate-javax.swing.event.CaretEvent->

```
jtf.addCaretListener(new CaretListener() {  
    public void caretUpdate(CaretEvent e)  
    {  
        System.out.println(e.getDot());  
    }  
});  
}  
}
```

Output:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**.

All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow programmers to control the behavior of buttons. For example, different icons can be displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon (Icon di)
void setPressedIcon (Icon pi)
void setSelectedIcon (Icon si)
void setRolloverIcon (Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```
String getText ()
void setText (String str)
```

str is the text to be associated with the button.

JButton

JButton class provides the functionality of a push button.

JButton allows an icon, a string, or both to be associated with the push button.

Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, *str* and *icon* are the string and icon used for the button.

str is the string which can be used to identify the button, which is also termed as “**Action Command String**”

When the button is pressed, an **ActionEvent** is generated.

Using the **ActionEvent** object passed to the **actionPerformed()** method of the registered **ActionListener**, *action command* string associated with the button can be obtained.

By default, this is the string displayed inside the button.

Action command can be set by calling **setActionCommand()** on the button.

Action command can be obtained by calling **getActionCommand()** on the event object.

```
String getActionCommand()
```

Thus, when using two or more buttons within the same application, the action command gives an easy way to determine which button was pressed.

Ex:

```
//Demonstrate an icon-based JButton.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
public class Test extends JApplet implements ActionListener {  
    JLabel jlab;  
    public void init() {  
        try {  
            SwingUtilities.invokeAndWait( new Runnable()  
            {  
                public void run() {  
                    makeGUI();  
                }  
            }  
            );  
        } catch (Exception exc) {System.out.println("Can't create because of " + exc); }  
    }  
  
private void makeGUI() {  
    // Change to flow layout.  
    setLayout(new FlowLayout());  
  
    // Add buttons to content pane.  
    ImageIcon india = new  
        ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/india.gif");  
    JButton jb = new JButton("India",india);  
    //jb.setActionCommand("India");  
/*  
The string "India" passed to the constructor of JButton itself will act as Action Command String.  
The same action command string can also be set by invoking setActionCommand( ) method of  
JButton.  
  
The string passed during the constructor call will be displayed as a part of the button.  
But, the string passed by setActionCommand will not be displayed as a part of the button.  
*/  
    jb.addActionListener(this);  
/* Above function call has "this" in its parameter list because the class Test is implementing  
ActionListener as part of the code, which in turn makes the class Test as both event source and  
event listener. */  
    add(jb);  
  
    ImageIcon usa = new
```

```

    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/usa.jpeg");
jb = new JButton(usa);
jb.setActionCommand("USA");
jb.addActionListener(this);
add(jb);

ImageIcon aus = new
    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/aus.jpeg");
jb = new JButton(aus);
jb.setActionCommand("Australia");
jb.addActionListener(this);
add(jb);

ImageIcon japan = new
    ImageIcon("/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/japan.jpeg");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
add(jb);

// Create and add the label to content pane.
jlab = new JLabel("Choose a Flag");
add(jlab);

}

// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
}

```

Check Boxes

The **JCheckBox** class provides the functionality of a check box.

JCheckBox defines several constructors.

JCheckBox (String str)

It creates a check box that has the text specified by *str* as a label.

Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a checkbox, an **ItemEvent** is generated.

A reference to the **JCheckBox** that generated the event can be obtained by calling **getItem()** on the **ItemEvent** passed to the **itemStateChanged()** method defined by **ItemListener**.

```
interface ItemListener
```

```
{  
    void itemStateChanged(ItemEvent e);  
    /*Invoked when an item has been selected or deselected by the user. The code written for this method  
    performs the operations that need to occur when an item is selected (or deselected).*/  
}
```

The easiest way to determine the selected state of a check box is to call **isSelected()** on the **JCheckBox** instance.

In addition to supporting the normal check box operation, **JCheckBox** allows programmers to specify the icons that indicate when a checkbox is selected, cleared, and rolled-over.

When user clicks a checkbox, an **ItemEvent** is generated. Inside the **itemStateChanged()** method, **getItem()** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected()** determines if the box was selected or cleared. The **getText()** method gets the text for that check box and uses it to set the text inside the label.



Ex:

```
//Demonstrate JCheckbox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet implements ItemListener {  
JLabel jlab;  
public void init() {  
try {  
SwingUtilities.invokeAndWait( new Runnable()  
{  
public void run()  
{  
makeGUI();  
}  
});  
}  
catch (Exception exc) {System.out.println("Can't create because of " + exc);}  
}  
private void makeGUI() {  
// Change to flow layout.  
setLayout(new FlowLayout());  
  
// Add check boxes to the content pane.  
JCheckBox cb = new JCheckBox("C");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("C++");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("Java");  
cb.addItemListener(this);  
add(cb);  
  
cb = new JCheckBox("Perl");  
cb.addItemListener(this);  
add(cb);  
  
// Create the label and add it to the content pane.  
jlab = new JLabel("Select languages");  
add(jlab);
```

```

}

// Handle item events for the check boxes.
public void itemStateChanged(ItemEvent ie) {
/*
Object getItem( ) method belongs to ItemEvent class which is a common base class for many GUI
components in swings, hence return type is generic in nature.
*/
JCheckBox cb = (JCheckBox)ie.getItem();
if(cb.isSelected())
    jlab.setText(cb.getText() + " is selected");
else
    jlab.setText(cb.getText() + " is cleared");
}
}

```

JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront.

JTabbedPane defines three constructors. Default constructor is considered, which creates an empty control with the tabs positioned across the top of the pane.

JTabbedPane uses the **SingleSelectionModel** model.

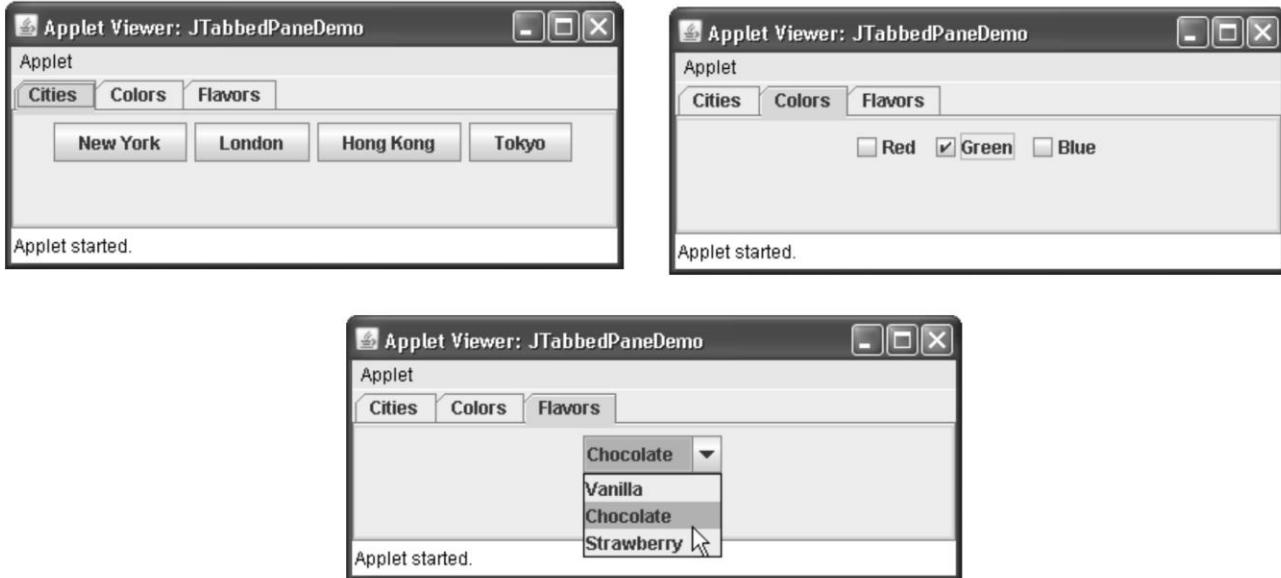
Tabs are added by calling **addTab()**. Here is one of its forms:

void addTab(String name, Component comp)

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

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

1. Create an instance of **JTabbedPane**.
2. Add each tab by calling **addTab()**.
3. Add the tabbed pane to the content pane.



Ex:

//Demonstrate JTabbedPane.

```
import javax.swing.*;
```

```
public class Test extends JApplet {
public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run()
        { makeGUI(); }
    } );
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    add(jtp);
}
//End of class Test
```

```
//Make the panels that will be added to the tabbed pane.  
class CitiesPanel extends JPanel {  
public CitiesPanel() {  
JButton b1 = new JButton("New York");  
add(b1);  
JButton b2 = new JButton("London");  
add(b2);  
JButton b3 = new JButton("Hong Kong");  
add(b3);  
JButton b4 = new JButton("Tokyo");  
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 FlavorsPanel extends JPanel {  
public FlavorsPanel() {  
JComboBox jcb = new JComboBox();  
jcb.addItem("Vanilla");  
jcb.addItem("Chocolate");  
jcb.addItem("Strawberry");  
add(jcb);  
} }
```

JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component.

The component being scrolled can either be an individual component, such as a table, or a group of components contained within another lightweight container, such as a **JPanel**.

If the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed.

In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed.

For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors.

JScrollPane(Component comp)

The component to be scrolled is specified by *comp*.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.



```

Ex://Demonstrate JScrollPane.
import java.awt.*;      import javax.swing.*;

public class Test extends JApplet {
public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run() {
            makeGUI();
        }
    });
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
// Add 400 buttons to a panel.
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++)
    for(int j = 0; j < 20; j++)
    {
        jp.add(new JButton("Button " + b));
        ++b;
    }
// Create the scroll pane.
JScrollPane jsp = new JScrollPane(jp);
// Add the scroll pane to the content pane.
add(jsp);
}
}//End of class Test

```

JList

JList supports the selection of one or more items from a list.

Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed.

JList provides several constructors.

JList(Object[] items)

This creates a **JList** that contains the items in the array specified by *items*.

Although a JList will work properly by itself, most of the time it will be wrapped inside JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**, which specifies only one method, called **valueChanged()**, which is shown here:

void valueChanged(ListSelectionEvent le)

Here, *le* is a reference to the object that generated the event.

```
interface ListSelectionListener
{
    void valueChanged(ListSelectionEvent le);
}
```

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode()**, which is defined by **JList**. It is shown here:

void setSelectionMode(int mode)

Here, *mode* specifies the selection mode. It must be one of these values defined by

ListSelectionMode:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, allows the user to select multiple ranges of items within a list.

With single-interval selection, the user can select one range of items.

With single selection, the user can select only a single item.



Ex:

```
//Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class Test extends JApplet {
JList jlst;
JLabel jlab;
 JScrollPane jsclrp;

// Create an array of cities.
String Cities[] = { "New York", "Chicago", "Houston",
                    "Denver", "Los Angeles", "Seattle",
                    "London", "Paris", "New Delhi",
                    "Hong Kong", "Tokyo", "Sydney" };

public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run() {
            makeGUI();
        }
    });
} catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
// Change to flow layout.
```

```

setLayout(new FlowLayout());

//Create a JList.
jlst = new JList(Cities);

// Set the list selection mode to single selection.
jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Add the list to a scroll pane.
jscrlp = new JScrollPane(jlst);

// Make a label that displays the selection.
jlab = new JLabel("Choose a City");

// Add selection listener for the list.
jlst.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent le)
    {
        // Get the index of the changed item.
        int idx = jlst.getSelectedIndex();
        // Display selection, if item was selected.
        if(idx != -1)
            jlab.setText("Current selection: " + Cities[idx]);
        else // Otherwise, reprompt.
            jlab.setText("Choose a City");
    }
});
// Add the list and label to the content pane.
add(jscrlp);
add(jlab);  } }

```

JComboBox

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

A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry.

A combo box can also be created that lets the user enter a selection into the text field.

JComboBox constructor used by the example is shown here:

JComboBox(Object[] items)

Here, *items* is an array that initializes the combo box.

JComboBox uses the **ComboBoxModel**.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem()** method,

void addItem (Object obj)

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

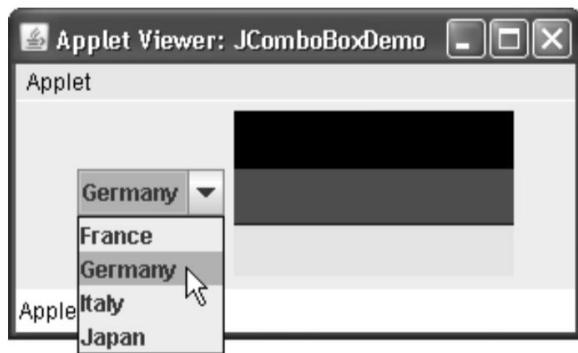
JComboBox generates an **action event** when the user selects an item from the list.

JComboBox also generates an **item event** when the state of selection changes, which occurs when an item is selected or deselected.

To obtain the item selected in the list is to call **getSelectedItem()** on the combo box.

Object getSelectedItem()

There is a need to cast the returned value into the type of object stored in the list.



Ex:

```
//Demonstrate JComboBox.  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class Test extends JApplet {  
    JLabel jlab;  
    ImageIcon france, germany, italy, japan;  
    JComboBox jcb;  
    String flags[] = { "france", "usa", "aus", "Japan" };  
public void init()  
{  
    try {  
        SwingUtilities.invokeAndWait( new Runnable()  
        {  
            public void run()  
            { makeGUI(); }  
        }  
    );  
    }  
    catch (Exception exc)  
    {System.out.println("Can't create because of " + exc);}  
}  
  
private void makeGUI()  
{  
    // Change to flow layout.  
    setLayout(new FlowLayout());  
  
    // Instantiate a combo box and add it to the content pane.  
    jcb = new JComboBox(flags);  
    add(jcb);
```

```

// Handle selections.
jcb.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent ae)
    {
        String s= "/Users/hemanth/eclipse-workspace/mergesort/src/mergesort/";
        s = s + (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".jpeg"));
    }
}
);
;

// Create a label and add it to the content pane.
jlab = new JLabel(new ImageIcon("france.gif"));
add(jlab);
}
}

```

JTable

JTable is a component that displays rows and columns of data.

Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.

JTable has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

JTable is a component that consists of one or more columns of information.

At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table.

JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors.

JTable (Object data[][], Object colHeads[])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models.

The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format.

The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**.

The **third** model determines how items are selected, and it is specified by the **ListSelectionModel**.

A **JTable** can generate several different events.

The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**.

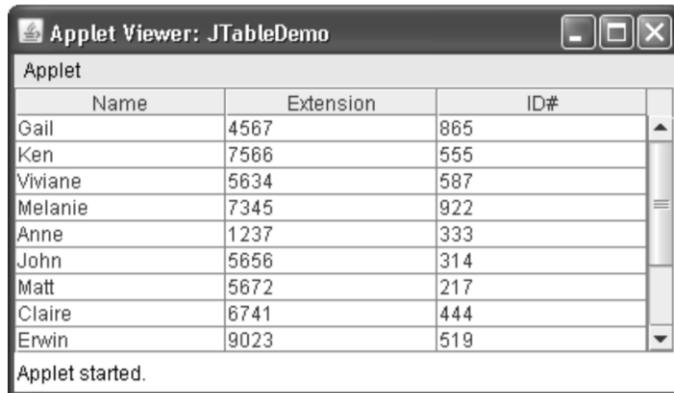
A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected.

A **TableModelEvent** is fired when that table's data changes in some way.

If **JTable** is used only to display data then it is not necessary to handle events.

Steps required to set up a simple **JTable** which can be used to display data:

1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.



Ex:

```
//Demonstrate JTable.  
import java.awt.*;  
import javax.swing.*;
```

```

public class Test extends JApplet {
public void init() {
try {
    SwingUtilities.invokeAndWait( new Runnable()
    {
        public void run() {
            makeGUI();
        }
    }
);
}
catch (Exception exc) {System.out.println("Can't create because of " + exc);}
}

private void makeGUI() {
// Initialize column headings.
String[] colHeads = { "Name", "Extension", "ID#" };

// Initialize data.
Object[][] data = {
{ "Gail", "4567", "865" },
{ "Ken", "7566", "555" },
{ "Viviane", "5634", "587" },
{ "Melanie", "7345", "922" },
{ "Anne", "1237", "333" },
{ "John", "5656", "314" },
{ "Matt", "5672", "217" },
{ "Claire", "6741", "444" },
{ "Erwin", "9023", "519" },
{ "Ellen", "1134", "532" },
{ "Jennifer", "5689", "112" },
{ "Ed", "9030", "133" },
{ "Helen", "6751", "145" }
};
// Create the table.
JTable table = new JTable(data, colHeads);

// Add the table to a scroll pane.
JScrollPane jsp = new JScrollPane(table);

// Add the scroll pane to the content pane.
add(jsp);
} }

```