ADVANCED JAVA PROGRAMMING

LECTURE NOTES (Semester-II)

for

Master of Science in Computer Science

Department of Computer Science and Applications

Vinayaka Mission's Research Foundation

School of Arts And Science, Av Campus

Chennai-603104

Lecture Note Prepared By

S.MAHALAKSHMI, Asst.Professor

SYLLUBUS

DSC –IV: ADVANCED JAVA PROGRAMMING

Objectives:

- At the end of the course, the student should be able to do advanced programming using Java swing, AWT, JDBC, java Servlets.
- To enable the student to learn the advanced programming concepts in Java.

UNIT I

Multithreading: Java Thread Model-Main Thread-Creating a Thread-Creating Multiple Threads-Using isAlive() and join()-Synchronization-Interthread Communication-Suspending, Resuming and Stopping Threads-Using Multithreading.

I/O Exploring java.io: Java I/O classes and interfaces-File-Closeable and Flushable Interfaces- The stream classes-Byte Streams-Character Streams-Console Class-Using Stream I/O-Serialization.

Networking: Basics-Networking classes and interface-Inet Address-Inet4 Address and Inet6Address-TCP/IP Client Socket-URL-URL connection-http URL Connection-URI class-Cookies-TCP/IP server socket-Datagrams.

Event Handling: Event Handling mechanisms-Delegation Event model-Event classes-Source of Events-Event Listener Interfaces-Using delegation Event model-Adapter classes-Inner classes.

UNIT II

AWT: AWT classes-Window Fundamentals-Working with frame windows-Creating a frame window in an applet-Creating a windowed program-Displaying information within a window-Working with Graphics, color and font-Managing text output using font metrics.**AWT Controls:** Control Fundamentals, Labels, Using Buttons, Checkboxes, Choice Control, List ,Scroll Bars and TextField, **AWT Layouts and Menus:** Understanding Layout Managers- Menu Bars and Menus-Dialog Boxes-File Dialog-Handling Events.

UNIT III

Images, Animation and Audio: File Format-Image fundamentals-Image Observer-Double Buffering-Media Tracker-Image Producer, Consumer and Filter-Cell Animation. **Swing**: Features of Swing-MVC Connection-Components and containers-Swing packages-Event handling-Creating a swing-Exploring swing. **JDBC**: Introduction-Relational Databases-SQL Manipulating Database with JDBC.

UNIT IV

Java Servlets: Life Cycle-Simple Servlet - Servlet API-javax.servlet package-javax.servlet.httpPackage-Handling HTTP requests and responses-cookies-session tracking. **Java Server Pages**:Overview-Implicit Objects-Scripting- Standard actions- Directives.

Remote MethodInvocation-Client/Server Application using RMI.

UNIT V

EJB: EJB Architecture-overview-Building and Deploying EJB-Roles in EJB-Design and Implementation-**EJB Session Bean**: Constraints-Life Cycle-Stateful Session Bean-StatelessSession Bean- **EJB Entity Bean**: Bean managed versus Container managed persistence – LifeCycle- Deployment.

TEXT BOOKS:

- Herbert Schildt, "The Complete Reference – JAVA," 7th Edition, TMH,2012
- Deitel H.M. &Deitel P.J, "Java: How To Program," Prentice-Hall of India, 5th Edition, 2003.
- Tom Valesky, "Enterprise JavaBeans – Developing component based DistributedApplications," Pearson 2000.

REFERENCE BOOKS:

- C.Muthu, "Programming with Java," Vijay Nicole Imprints Private Ltd., 2004
- Cay.S. Horstmann, Gary Cornel, "Core Java 2 – Vol. II- Advanced Features," Pearson
- Education, 2004.
- S.Gokila, "Advanced JAVA Programming," Vijay Nicole Imprints Private Ltd., 2014

Advanced Java Programming

Unit - I

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*.

In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.

Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.

Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.

In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

Threads exist in several states. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity.

A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.

Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.

Synchronization

If you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other.

That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*.

The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread.

Once a thread enters a monitor, all other threads must wait until that thread exits the monitor.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads.

Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

Main thread in Java

Java provides built-in support for multithreaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins.

Properties :

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

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method *currentThread( )* which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

Example:

```
public class Test extends Thread

{

public static void main(String[] args)

{

Thread t = Thread.currentThread();

System.out.println("Current thread: " + t.getName());

t.setName("Geeks");

System.out.println("After name change: " + t.getName());

System.out.println("Main thread priority: "+ t.getPriority());

t.setPriority(MAX_PRIORITY);

System.out.println("Main thread new priority: "+ t.getPriority());
for (int i = 0; i < 5; i++)

{

System.out.println("Main thread");

}

ChildThread ct = new ChildThread();

System.out.println("Child thread priority: "+ ct.getPriority());

ct.setPriority(MIN_PRIORITY);

System.out.println("Child thread new priority: "+ ct.getPriority());

ct.start();

}

}

class ChildThread extends Thread

{

@Override

public void run()

{

for (int i = 0; i < 5; i++)

{

System.out.println("Child thread");

}

}

}
```

Output:

```
Current thread: main

After name change: Geeks

Main thread priority: 5

Main thread new priority: 10

Main thread

Main thread

Main thread

Main thread

Main thread

Child thread priority: 10

Child thread new priority: 1

Child thread

Child thread

Child thread

Child thread

Child thread
```

Creating A thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

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

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r,String name)

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

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

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

A new thread starts(with new callstack).

The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

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

```
Output:thread is running...
```

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

```
Output:thread is running...
```

Creating Multithreading in Java

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

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and

multithreading, both are used to achieve multitasking.

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

They don't allocate separate memory area so saves memory, and context-switching between the

threads takes less time than process.

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

Advantages of Java Multithreading

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

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

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

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize

the CPU. Multitasking can be achieved in two ways:

1.Process-based Multitasking (Multiprocessing)
2.Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)
Each process has an address in memory. In other words, each process allocates a separate memory area.
A process is heavyweight.
Cost of communication between the process is high.
Switching from one process to another requires some time for saving and loading registers, memory

maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Thread Life Cycle in Java

There are various stages of life cycle of thread as shown in above diagram:

New
Runnable
Running
Waiting
Dead

New: In this phase, the thread is created using class "Thread class".
It remains in this state till the program **starts** the thread. It is also known as born thread.

Runnable: In this page, the instance of the thread is invoked with a start method.

The thread control is given to scheduler to finish the execution. It depends on the scheduler,

whether to run the thread.

Running: When the thread starts executing, then the state is changed to "running" state.

The scheduler selects one thread from the thread pool, and it starts executing in the application.

Waiting: This is the state when a thread has to wait. As there multiple threads are running in the application,

there is a need for synchronization between threads. Hence, one thread has to wait,

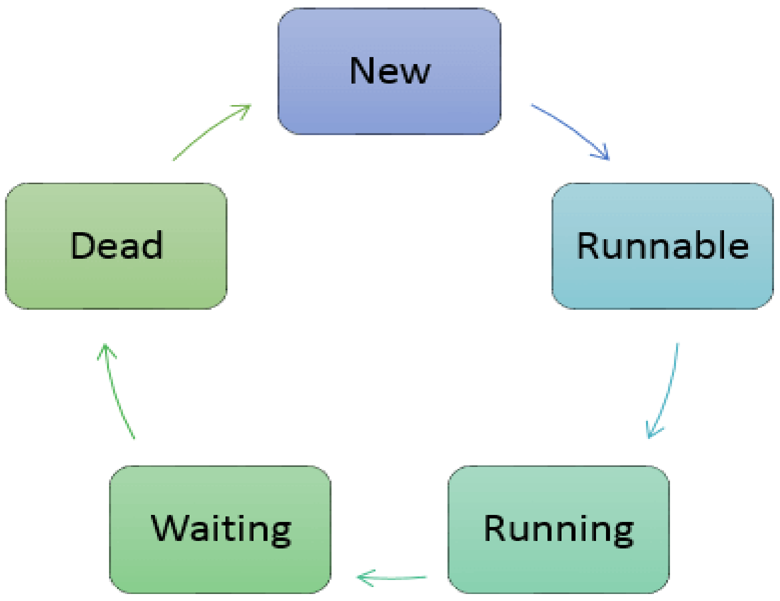
till the other thread gets executed. Therefore, this state is referred as waiting state.

Dead: This is the state when the thread is terminated. The thread is in running

state and as soon as it completed processing it is in "dead state".

Some of the commonly used methods for threads are:

Method	**Description**
start()	This method starts the execution of the thread and JVM calls the run() method on the thread.
Sleep(int milliseconds)	This method makes the thread sleep hence the thread's execution will pause for milliseconds provided and after that, again the thread starts executing. This help in synchronization of the threads.
getName()	It returns the name of the thread.
setPriority(int newpriority)	It changes the priority of the thread.
yield ()	It causes current thread on halt and other threads to execute.



Example

```
public class GuruThread1 implements Runnable{
    public static void main(String[] args) {
        Thread guruThread1 = new Thread("Guru1");
        Thread guruThread2 = new Thread("Guru2");
        guruThread1.start();
        guruThread2.start();
        System.out.println("Thread names are following:");
        System.out.println(guruThread1.getName());
        System.out.println(guruThread2.getName());
    }
    @Override
    public void run() {
    }
}
```

Output:

```
Thread names are following:
Guru1
Guru
```

isAlive() and join() methods of Thread Class in Java

Java multi-threading provides two ways to find that

isAlive() : It tests if this thread is alive. A thread is alive if it has been started and has not yet died. There is a transitional period from when a thread is running to when a thread is not running. After the run() method returns, there is a short period of time before the thread stops. If we want to know if the start method of the thread has been called or if thread has been terminated, we must use isAlive() method. This method is used to find out if a thread has actually been started and has yet not terminated.

General Syntax :

final boolean isAlive( )

Return Value: returns true if the thread upon which it is called is still running. It returns false otherwise.

Example:

```
public class oneThread extends Thread {

   public void run()

   {

      System.out.println("geeks ");

      try {

         Thread.sleep(300);

      }

      catch (InterruptedException ie) {

      }

      System.out.println("forgeeks ");

   }

   public static void main(String[] args)

   {

      oneThread c1 = new oneThread();

      oneThread c2 = new oneThread();

      c1.start();

      c2.start();

      System.out.println(c1.isAlive());

      System.out.println(c2.isAlive());
```

```
    }
}
```

Output:

```
geeks
true
true
geeks
forgeeks
forgeeks
```

join() : When the join() method is called, the current thread will simply wait until the thread it is joining with is no longer alive.Or we can say the method that you will more commonly use to wait for a thread to finish is called join( ). This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Syntax :

final void join( ) throws InterruptedException

Example

```
public class oneThread extends Thread {
    public void run()
    {
        System.out.println("geeks ");
        try {
            Thread.sleep(300);
        }
        catch (InterruptedException ie) {
        }
        System.out.println("forgeeks ");
    }
    public static void main(String[] args)
    {
        oneThread c1 = new oneThread();
        oneThread c2 = new oneThread();
        c1.start();
        try {
            c1.join(); // Waiting for c1 to finish
        }
        catch (InterruptedException ie) {
        }
        c2.start();
    }
}
```

Output:

```
geeks
forgeeks
geeks
forgeeks
```

Synchronization in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Syntax:

```
synchronized(sync_object)

{

   // Access shared variables and other

   // shared resources

}
```

This synchronization is implemented in Java with a concept called monitors. 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.

Example:

```
class Sender
{
    public void send(String msg)
    {
        System.out.println("Sending\t"  + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread  interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
    private String msg;
    Sender  sender;

    // Recieves a message object and a string
    // message to be sent
    ThreadedSend(String m,  Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
```

```
. synchronized(sender)
        {
           // synchronizing the snd object
           sender.send(msg);
        }
    }
}

class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );

        // Start two threads of ThreadedSend type
        S1.start();
        S2.start();

        // wait for threads to end
        try
        {
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

Output:

```
Sending    Hi

 Hi Sent

Sending    Bye

 Bye Sent
```

Inter-thread Communication in Java

The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is

true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.
For example, in a classic queuing problem where one thread is producing data and other is consuming it.

To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().** All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.
wait()-It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
notify()-It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
notifyAll()-It wakes up all the threads that called wait() on the same object.

Example:

```
import java.util.Scanner;

public class threadexample

{

    public static void main(String[] args)

                        throws InterruptedException

    {

        final PC pc = new PC();

        Thread t1 = new Thread(new Runnable()

        {

            @Override

            public void run()

            {

                try

                {

                    pc.produce();

                }

                catch(InterruptedException e)
        {
           e.printStackTrace();
        }
    }
});
Thread t2 = new Thread(new Runnable()
{
    @Override
    public void run()
    {
        try
        {
           pc.consume();
        }
        catch(InterruptedException e)
        {
           e.printStackTrace();
        }
    }
});
t1.start();
t2.start();
t1.join();
t2.join();
```

```
}
public static class PC
{
    public void produce()throws InterruptedException
    {
        synchronized(this)
        {
            System.out.println("producer thread running");
            wait();
            System.out.println("Resumed");
        }
    }
    public void consume()throws InterruptedException
    {
        // this makes the produce thread to run first.
        Thread.sleep(1000);
        Scanner s = new Scanner(System.in);
        synchronized(this)
        {
            System.out.println("Waiting for return key.");
            s.nextLine();
            System.out.println("Return key pressed");
            notify();
            Thread.sleep(2000);
            }
        }
    }
}
```

Output:

```
producer thread running
Waiting for return key.
Return key pressed
Resumed
```

Java Suspend Resume Stop Threads

Sometimes, suspending the execution of a thread is useful. For example, a separate thread can be used to display the time of day. If user does not want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

Prior to Java 2, a program used the suspend(), resume(), **and** stop() **methods, which are defined by the** Thread**, to pause, restart, and stop the execution of a thread.**

Java suspend Thread

The **suspend()** method of **Thread** class was deprecated by Java 2 several years ago. This was done because the **suspend()** can sometimes cause serious system failures.

Assume that a thread has obtained locks on the critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

Java resume Thread

The **resume()** method is also deprecated. It doest not cause problems, but cannot be used without the **suspend()** method as its counterpart.

Java stop Thread

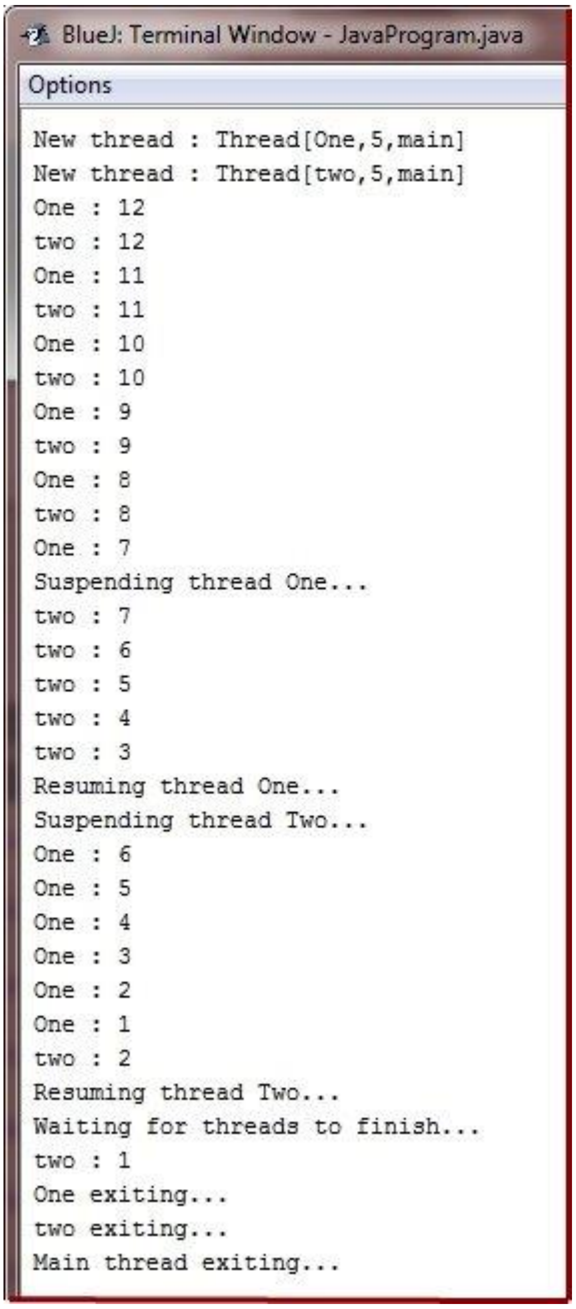
The **stop()** method of **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.

Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that, the **stop()** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Example:

```
class NewThread implements Runnable
 {
    String name;      //name of thread
    Thread thr;
    boolean suspendFlag;

    NewThread(String threadname)
    {
       name = threadname;
       thr = new Thread(this, name);
       System.out.println("New thread : " + thr);
       suspendFlag = false;
       thr.start();     // start the thread
    }

    /* this is the entry point for thread */
    public void run()
    {
       try
       {
          for(int i=12; i>0; i--)
          {
             System.out.println(name + " : " + i);
             Thread.sleep(200);
             synchronized(this)
             {
                while(suspendFlag)
                {
                   wait();
                }
             }
          }
       }
       catch(InterruptedException e)
       {
          System.out.println(name + " interrupted");
```

```
        }

        System.out.println(name + " exiting...");
    }

    synchronized void mysuspend()
    {
        suspendFlag = true;
    }

    synchronized void myresume()
    {
        suspendFlag = false;
        notify();
    }
}

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

        NewThread obj1 = new NewThread("One");
        NewThread obj2 = new NewThread("two");

        try
        {
            Thread.sleep(1000);
            obj1.mysuspend();
            System.out.println("Suspending thread One...");
            Thread.sleep(1000);
            obj1.myresume();
            System.out.println("Resuming thread One...");

            obj2.mysuspend();
            System.out.println("Suspending thread Two...");
            Thread.sleep(1000);
            obj2.myresume();
            System.out.println("Resuming thread Two...");
        }
        catch(InterruptedException e)
        {
            System.out.println("Main thread Interrupted..!!");
        }

        /* wait for threads to finish */
        try
        {
            System.out.println("Waiting for threads to finish...");
            obj1.thr.join();
            obj2.thr.join();
        }
        catch(InterruptedException e)
        {
            System.out.println("Main thread Interrupted..!!");
        }

        System.out.println("Main thread exiting...");

    }
}
```



```
New thread : Thread[One,5,main]
New thread : Thread[two,5,main]
One : 12
two : 12
One : 11
two : 11
One : 10
two : 10
One : 9
two : 9
One : 8
two : 8
One : 7
Suspending thread One...
two : 7
two : 6
two : 5
two : 4
two : 3
Resuming thread One...
Suspending thread Two...
One : 6
One : 5
One : 4
One : 3
One : 2
One : 1
two : 2
Resuming thread Two...
Waiting for threads to finish...
two : 1
One exiting...
two exiting...
Main thread exiting...
```

Java I/O and Interface

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.

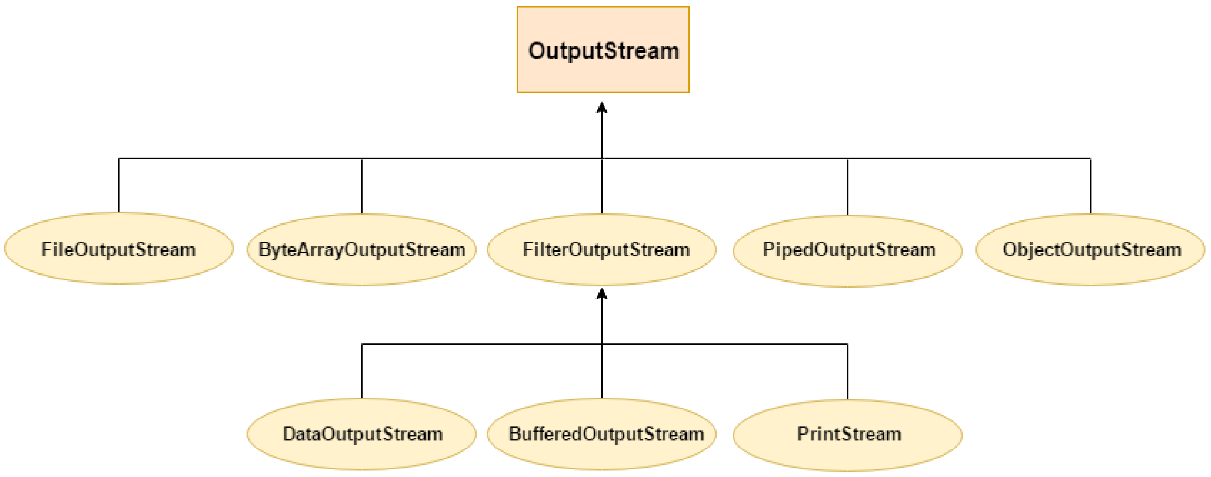
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



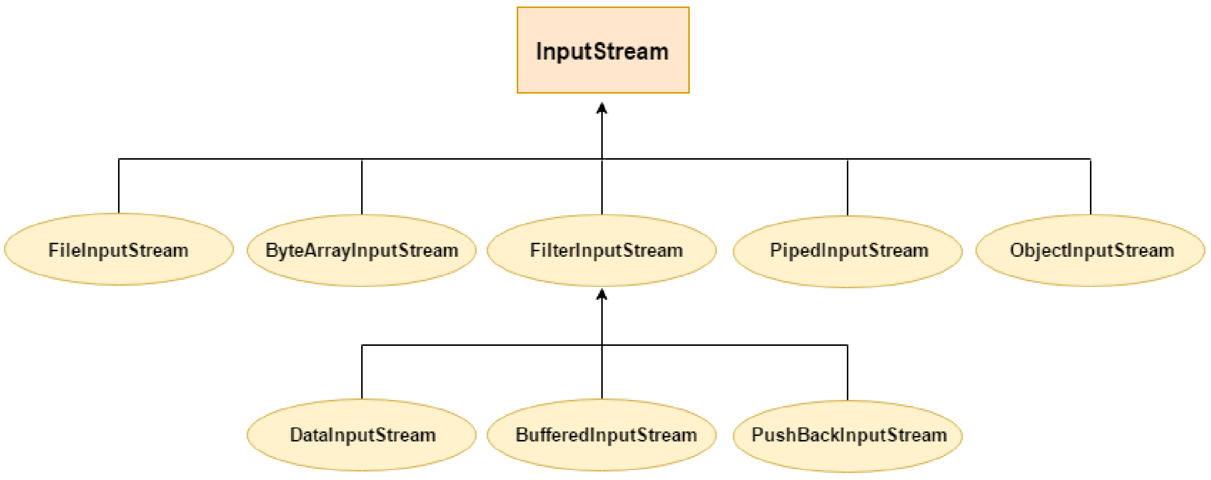
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Fields

Modifier	Type	Field	Description
static	String	pathSeparator	It is system-dependent path-separator character, represented as a string for convenience.
static	char	pathSeparatorChar	It is system-dependent path-separator character.
static	String	separator	It is system-dependent default name-separator character, represented as a string for convenience.
static	char	separatorChar	It is system-dependent default name-separator character.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Methods

Modifier and Type	**Method**	**Description**
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract

		pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Example:

```
import java.io.*;
public class FileDemo {
   public static void main(String[] args) {

      try {
         File file = new File("javaFile123.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
         }
```

```
        } catch (IOException e) {
          e.printStackTrace();
        }

    }
}
```

Output:

```
New File is created!
```

Closeable and Flushable interfaces

The Closeable and Flushable interfaces define a uniform way for specifying that a stream can be closed or flushed.

Closeable: The objects that implement the Closeable interface can be closed. This interface defines the close()method.

Void close( ) throws IOException

It closes the invoking stream and releases all the resources. It is implemented by all the I/O classes that open a stream.

Flushable: The objects that implement the Flushable interface can force buffered output to be written to a stream to which the objects are attached. This interface defines the flush() method.

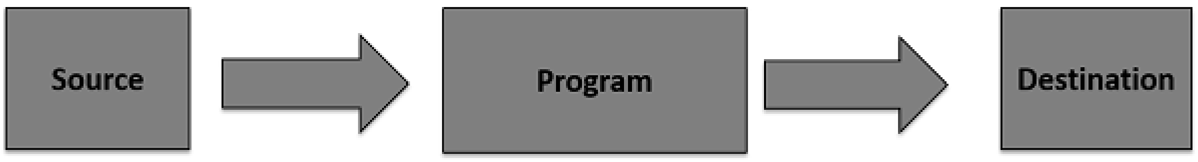
void flush() throws IOException

Stream Class

A stream can be defined as a sequence of data. There are two kinds of Streams −

InPutStream − The InputStream is used to read data from a source.

OutPutStream − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O.

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

```
import java.io.*;

public class CopyFile {

   public static void main(String args[]) throws IOException {

      FileInputStream in = null;

      FileOutputStream out = null;

      try {

         in = new FileInputStream("input.txt");

         out = new FileOutputStream("output.txt");

         int c;

         while ((c = in.read()) != -1) {

            out.write(c);

         }

      }finally {

         if (in != null) {

            in.close();

         }

         if (out != null) {

            out.close();

         }

      }
```

```
  }

}
```

Output:

Now let's have a file **input.txt** with the following content −

This is test for copy file.

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode.

Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**.

Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;

public class CopyFile {

  public static void main(String args[]) throws IOException {

    FileReader in = null;

    FileWriter out = null;

    try {

      in = new FileReader("input.txt");

      out = new FileWriter("output.txt");

      int c;

      while ((c = in.read()) != -1) {

        out.write(c);

      }

    }finally {
        if (in != null) {
          in.close();
        }
        if (out != null) {
          out.close();
        }
      }
    }
}
```

Output:

Now let's have a file **input.txt** with the following content −

This is test for copy file.

Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally.

Let's see a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console class declaration

Let's see the declaration for Java.io.Console class:

public final class Console **extends** Object **implements** Flushable

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

Example

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

Output:

```
Enter your name: Nakul Jain

Welcome Nakul
```

Serialization

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

For serializing the object, we call the **writeObject()** method *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (which is known as marshaling).

java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

Example:

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
class Persist{
public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

Output:

```
success
```

Networking

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols −

TCP − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

UDP − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects −

Socket Programming − This is the most widely used concept in Networking and it has been explained in very detail.

URL Processing − This would be covered separately. Click here to learn about URL Processing in Java language.

Java is a premier language for network programming. **java.net** package encapsulate large number of classes and interface that provides an easy-to use means to access network resources.

NETWORKING CLASSES & INTERFACES

- **The java.net package provides the interfaces and classes for implementing networking applications.**

Interfaces of java.net package

Interface	**Use**
ContentHandlerFactory	This interface defines a factory for content handlers.
CookiePolicy	CookiePolicy implementations decide which cookies should be accepted and which should be rejected.
CookieStore	A CookieStore object represents storage for cookie.
DatagramSocketImplFactory	This interface defines a factory for datagram socket implementations.
FileNameMap	A simple interface which provides a mechanism to map between a file name and a MIME type string.
SocketImplFactory	This interface defines a factory for socket implementations.
SocketOptions	Interface of methods to get/set socket options.
URLStreamHandlerFactory	This interface defines a factory for URL stream protocol

	handlers.

Classes of java.net package

The classes in the networking package fall into three general categories:

Web interface classes

- URL and URLConnection classes
- Raw network interface classes

Class	**Use**
ContentHandler	Super class of the classes that turn MIME objects into corresponding Java objects
CookieHandler	A CookieHandler object provides a callback mechanism to hook up a HTTP state management policy implementation into the HTTP protocol handler.
CookieManager	CookieManager provides a concrete implementation of CookieHandler, which separates the storage of cookies from the policy surrounding accepting and rejecting cookies.
DatagramPacket	This class represents a datagram packet.
DatagramSocket	This class represents a socket for sending and receiving datagram packets.
Inet4Address	This class represents an Internet Protocol version 4 (IPv4) address.
Inet6Address	This class represents an Internet Protocol version 6 (IPv6) address.
InetAddress	This class represents an Internet Protocol (IP) address.
NetPermission	This class is for various network permissions.

PasswordAuthentication	The class PasswordAuthentication is a data holder that is used by Authenticator.
Proxy	This class represents a proxy setting, typically a type (http, socks) and a socket address.
ResponseCache	Represents implementations of URLConnection caches.
ServerSocket	This class implements server sockets.
Socket	This class implements client sockets (also called just "sockets").
URI	Represents a Uniform Resource Identifier(URI) reference.
URL	Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
URLConnection	The abstract class URLConnection is the superclass of all classes that represent a communications link between the application and a URL.

Java InetAddress class

Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of address types: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

Commonly used methods of InetAddress class

Method	**Description**

public static InetAddress getByName(String host) throws UnknownHostException	it returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	it returns the instance of InetAdddress containing local host name and address.
public String getHostName()	it returns the host name of the IP address.
public String getHostAddress()	it returns the IP address in string format.

Example

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.javatpoint.com");

System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}
}
}
```

Output:

```
Host Name: www.javatpoint.com
IP Address: 206.51.231.148
```

Java.net.Inet4Address

This class extends InetAddress class and represents an IPv4 address. It provides methods to interpret and display useful information about ip addresses.
Methods of this class takes input in 4 formats:
d.d.d.d : When this format is used as input, each of the given values are assigned to 4 bytes of the IP address from left to right.
d.d.d : When this format is used as input, the last part is interpreted as 16 bit number and assigned to the rightmost 2 bytes as the host address. This is generally used for specifying class-B address.
d.d : When this format is used as input, the last part is interpreted as 24 bit number and assigned to the rightmost 3 bytes as the host address. This is generally used for specifying class-A address.
d : When this format is used as input, the given value is directly stored as network address without any rearrangement.
Methods :

getAddress() : returns raw IP address of this InetAddress object as an array. The order in which bytes appear in array are same as in IP address i.e. getAddress[0] will contain highest order byte.
Syntax : public byte[] getAddress()
getHostAddress() : returns IP address in textual form.
Syntax :public String getHostAddress()
isAnyLocalAddress() : returns true if this address represents a local address.
Syntax :public boolean isAnyLocalAddress()
isLinkLocalAddress() : returns true if this address is a link local address.
Syntax :public boolean isLinkLocalAddress()
isLoopbackAddress() : returns true if this address is a loopback address.
Syntax :public boolean isLoopbackAddress()
isMCGlobal() : returns true if this multicast address has global scope.
Syntax :public boolean isMCGloabal()
isMCLinkLocal() : returns true if this multicast address has link scope.
Syntax :public boolean isMCLinkLocal()
isMCNodeLocal() : returns true if this multicast address has node scope.
Syntax :public boolean isMCNodeLocal()
isMCOrgLocal() : returns true if this multicast address has organisation scope.
Syntax :public boolean isMCOrgLoacal()
isMCSiteLocal() : returns true if this multicast address has site scope.
Syntax :public boolean isMCSiteLocal()
isMulticastAddress() : returns true if this address is an IP multicast address. Multicast addresses have 1110 as their first 4 bits.
Syntax :public boolean isMulticastAddress()
isSiteLocalAddress(): returns true if this address is a site local address.
Syntax :public boolean isSiteLocalAddress()()
hashCode() : returns the hashcode associated with this address object.
Syntax : public int hashCode()
equals() : returns true if this ip address is same as that of the object specified. Equals() method don’t consider host names while comparing and only consider IP address associated.
Syntax : public boolean equals(Object obj)
Parameters :

obj : object to compare with

Example:

```
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;

public class inet4add
{
    public static void main(String args[]) throws UnknownHostException
    {
        String url = "www.geeksforgeeks.org";
        Inet4Address ip1 = (Inet4Address) Inet4Address.getByName(url);
        Inet4Address ip2 = (Inet4Address) InetAddress.getByName("www.yahoo.com");


        System.out.println("Address : " + Arrays.toString(ip1.getAddress()));


        System.out.println("Host Address : " + ip1.getHostAddress());


        System.out.println("isAnyLocalAddress : " + ip1.isAnyLocalAddress());


        System.out.println("isLinkLocalAddress : " + ip1.isLinkLocalAddress());


        System.out.println("isLoopbackAddress : " + ip1.isLoopbackAddress());


        System.out.println("isMCGlobal : " + ip1.isMCGlobal());


        System.out.println("isMCLinkLocal : " + ip1.isMCLinkLocal());


        System.out.println("isMCNodeLocal : " + ip1.isMCNodeLocal());


    }
}
```

Output :

```
Address : [52, 84, 102, -116]
Host Address : 52.84.102.140
isAnyLocalAddress : false
isLinkLocalAddress : false
isLoopbackAddress : false
```

```
isMCGlobal : false
isMCLinkLocal : false
isMCNodeLocal : false
```

Java.net.Inet6Address

This class represents IPv6 address and extends the InetAddress class. Methods of this class provide facility to represent and interpret IPv6 addresses.

Methods of this class takes input in the following formats:

x:x:x:x:x:x:x:x –This is the general form of IPv6 address where each x can be replaced with a 16 bit hexadecimal value of the address. Note that there must be a value in place of every 'x' when using this format. For example,

```
4B0C:0:0:0:880C:99A8:4B0:4411
```

When the address contains multiple set of 8 bits as '0', a special format can be used to compress the address. In such cases '::' is replaced in place of 0's to make the address shorter. For example, the address in previous example can be written as-

```
4B0C::880C:99A8:4B0:4411
```

x:x:x:x:x:x:d.d.d.d –A third format is used when hybrid addressing(IPv6 + IPv4) has to be taken care of. In such cases the first 12 bytes are used for IPv6 addressing and remaining 4 bytes are used for IPv4 addressing. For example,

```
F334::40CB:152.16.24.142
```

::FFFF:d.d.d.d – This type of addressing is known as **IPv4-mapped addressing**. It is used to aid in the deployment of IPv6 addressing. It allows the use of same structure and socket to communicate via both IPv6 and IPv4 connected network. First 80 bits are filled with 0's represented by '::'. Next 32 bits are all '1' and remaining 32 bits represent the IPv4 address. For example,

```
::FFFF:152.16.24.123
```

Methods :

getByAddress(String host, byte[] addr, int scope_id) : This is used to create an Inet6Address object by setting a IPv6 scope id to the given value. Object returned is similar to as created by InetAddress.getByAddress(String, byte[]) with additional info about scope id.

getByAddress(String host, byte[] addr, NetworkInterface nif): An overloaded method getByAddress() can be used to specify the network interface to be used with the address. In this case, the scope id corresponding to network interface is used as scope id.

getScopeId() : Returns the scope id associated with this address or 0 if none set.
Syntax : public int getScopeId()
getScopedInterface() : Returns the network interface associated with this address or null if none set.
Syntax : public NetworkInterface getScopedInterface()
getAddress() : returns raw IP address of this InetAddress object as an array. The order in which bytes appear in array are same as in IP address i.e. getAddress[0] will contain highest order byte.
Syntax : public byte[] getAddress()
getHostAddress() : returns IP address in textual form.
Syntax :public String getHostAddress()

Example:

```
import java.net.Inet6Address;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Arrays;

public class inet6add
{

   public static void main(String[] args) throws UnknownHostException
   {

       String host = "localhost";
       byte add[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 };

       Inet6Address ip1 = Inet6Address.getByAddress(host, add, 5);
       Inet6Address ip2 = Inet6Address.getByAddress(null, add, 6);


       System.out.println("Scope Id : " + ip1.getScopeId());


       System.out.println("Scoped Interface : " + ip1.getScopedInterface());


       System.out.println("Address : " + Arrays.toString(ip1.getAddress()));


       System.out.println("Host Address : " + ip1.getHostAddress());



   }

}
```

Output:

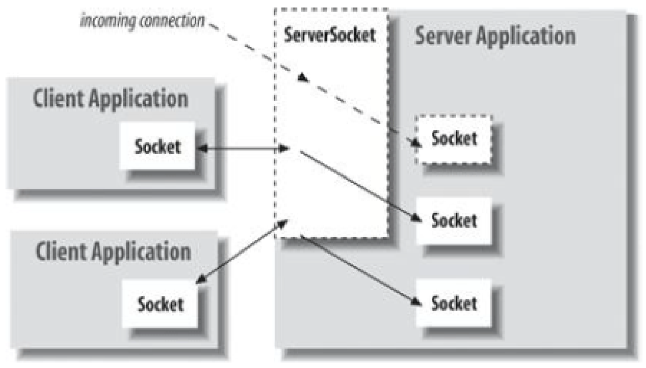
```
Scope Id : 5
Scoped Interface : null
Address : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Host Address : 0:0:0:0:0:0:0:1%5
isAnyLocalAddress : false
isLinkLocalAddress : false
isLoopbackAddress : true
isMCGlobal : false
```

TCP/IP CLIENT SOCKETS

TCP/IP sockets are used to implement t reliable, bidirectional, persistent, point-to-point, and stream - based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

TCP/IP Client and Server Sockets in Java

Clients and servers, Sockets and Server Sockets



Applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The Server Socket class is designed to be a listener, which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges.

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Client Socket API

The **Socket**class represents a socket client. You use this class to make connection to a server, send data to and read data from that server. The following steps are applied for a typical communication with the server:

1. The client initiates connection to a server specified by hostname/IP address and port number.
2. Send data to the server using an OutputStream.
3. Read data from the server using an InputStream.

4. Close the connection.

The steps 2 and 3 can be repeated many times depending on the nature of the communication.

Now, let’s study how to use the Socket class to implement these steps.

Initiate Connection to a Server:

To make a connection to a server, create a new Socket object using one of the following constructors:

- Socket(InetAddress address, int port)

- Socket(String host, int port)

- Socket(InetAddress address, int port, InetAddress localAddr, int localPort)

You see, it requires the IP address/hostname of the server and the port number.

With the first two constructors, the system automatically assigns a free port number and a local address for the client computer. With the third constructor, you can explicitly specify the address and port number of the client if needed. The first constructor is often used because of its simplicity.
These constructors can throw the following checked exceptions:

- IOException: if an I/O error occurs when creating the socket.

- UnknownHostException: if the IP address of the host could not be determined.

That means you have to catch (or re-throw) these checked exceptions when creating a Socket instance. The following line of code demonstrates how to create a client socket that attempts to connect to google.com at port number 80:

```
Socket socket = new Socket("google.com", 80)
```

Send Data to the Server:

To send data to the server, get the OutputStream object from the socket first:

```
OutputStream output = socket.getOutputStream();
```

Then you can use the **write()** method on the OutputStream to write an array of byte to be sent:

```
byte[] data = ….
output.write(data);
```

And you can wrap the OutputStream in a PrintWriter to send data in text format, like this:

```
PrintWriter writer = new PrintWriter(output, true);
writer.println(“This is a message sent to the server”);
```

The argument true indicates that the writer flushes the data after each method call (auto flush).

Read Data from the Server:

Similarly, you need to obtain an InputStream object from the client socket to read data from the server:

```
InputStream input = socket.getInputStream();
```

Then use the **read()** method on the InputStream to read data as an array of byte, like this:

```
byte[] data =…
input.read(data);
```

You can wrap the InputStream object in an InputStreamReader or BufferedReader to read data at higher level (character and String). For example, using InputStreamReader:

```
InputStreamReader reader = new InputStreamReader(input);
int character = reader.read();  // reads a single character
```

And using BufferedReader:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String line = reader.readLine();    // reads a line of text
```

Close the Connection:

Simply call the **close()** method on the socket to terminate the connection between the client and the server:

```
socket.close();
```

Example:

```
import java.net.*;
import java.io.*;

/**
 * This program is a socket client application that connects to a time server
 * to get the current date time.
 *
 * @author www.codejava.net
 */
public class TimeClient {

    public static void main(String[] args) {
        String hostname = "time.nist.gov";
        int port = 13;
        try (Socket socket = new Socket(hostname, port)) {

            InputStream input = socket.getInputStream();
            InputStreamReader reader = new InputStreamReader(input);

            int character;
            StringBuilder data = new StringBuilder();

            while ((character = reader.read()) != -1) {
                data.append((char) character);
            }

            System.out.println(data);


        } catch (UnknownHostException ex) {

            System.out.println("Server not found: " + ex.getMessage());

        } catch (IOException ex) {

            System.out.println("I/O error: " + ex.getMessage());
        }
    }
}
```

URL class

The URL class is the gateway to any of the resource available on the internet. A Class URL represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or to a search engine

What is a URL?
As many of you must be knowing that Uniform Resource Locator-URL is a string of text that identifies all the resources on Internet, telling us the address of the resource, how to communicate with it and retrieve something from it.
A Simple URL looks like:
http://www.geeksforgeeks.com/

ComponentsofaURL:-A URL can have many forms. The most general however follows three-components system-

Protocol: HTTP is the protocol here

Hostname: Name of the machine on which the resource lives.
File Name: The path name to the file on the machine.
Port Number: Port number to which to connect (typically optional).

Some constructors for URL class:-

URL(String address) throws MalformedURLException: It creates a URL object from the specified String.
URL(String protocol, String host, String file): Creates a URL object from the specified protcol, host, and file name.
URL(String protocol, String host, int port, String file): Creates a URL object from protocol, host, port and file name.
URL(URL context, String spec): Creates a URL object by parsing the given spec in the given context.
URL(String protocol, String host, int port, String file, URLStreamHandler handler):- Creates a URL object from the specified protocol, host, port number, file, and handler.
URL(URL context, String spec, URLStreamHandler handler):- Creates a URL by parsing the given spec with the specified handler within a specified context.

Example:

```
import java.net.MalformedURLException;

import java.net.URL;

public class URLclass1

{

   public static void main(String[] args)

                throws MalformedURLException

   {

      URL url1 =

      new URL("https://www.google.co.in/?gfe_rd=cr&ei=ptYq" +

            "WK26I4fT8gfth6CACg#q=geeks+for+geeks+java");

      URL url2 = new URL("http", "www.geeksforgeeks.org",

                  "/jvm-works-jvm-architecture/");

      URL url3 = new URL("https://www.google.co.in/search?"+

                    "q=gnu&rlz=1C1CHZL_enIN71" +

                    "4IN715&oq=gnu&aqs=chrome..69i57j6" +
                    "9i60l5.653j0j7&sourceid=chrome&ie=UTF" +
                    "-8#q=geeks+for+geeks+java");

        System.out.println(url1.toString());
        System.out.println(url2.toString());
        System.out.println();
        System.out.println("Different components of the URL3-");
        System.out.println("Protocol:- " + url3.getProtocol());
        System.out.println("Hostname:- " + url3.getHost());
        System.out.println("Default port:- " +
                                    url3.getDefaultPort());
        System.out.println("Query:- " + url3.getQuery());
        System.out.println("Path:- " + url3.getPath());
        System.out.println("File:- " + url3.getFile());
        System.out.println("Reference:- " + url3.getRef());
    }
}
```

Output:

```
Different components of the URL3-
Protocol:- https
Hostname:- www.google.co.in
Default port:- 443
```

URLConnection class

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

How to get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

public URLConnection openConnection()**throws** IOException{}

Displaying source code of a webpage by URLConnecton class

The URLConnection class provides many methods, we can display all the data of a webpage by using the getInputStream() method. The getInputStream() method returns all the data of the specified URL in the stream that can be read and displayed.

Example:

```
import java.io.*;
import java.net.*;
public class URLConnectionExample {
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
URLConnection urlcon=url.openConnection();
InputStream stream=urlcon.getInputStream();
int i;
while((i=stream.read())!=-1){
System.out.print((char)i);
}
}catch(Exception e){System.out.println(e);}
}
}
```

HttpURLConnection class

The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.

The java.net.HttpURLConnection is subclass of URLConnection class.

How to get the object of HttpURLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

public URLConnection openConnection()**throws** IOException{}

```
URL url=new URL("http://www.javatpoint.com/java-tutorial");
HttpURLConnection huc=(HttpURLConnection)url.openConnection();
```

Example:

```
import java.io.*;
import java.net.*;
public class HttpURLConnectionDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
HttpURLConnection huc=(HttpURLConnection)url.openConnection();
for(int i=1;i<=8;i++){
System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
}
huc.disconnect();
}catch(Exception e){System.out.println(e);}
}
}
```

Output:

```
Date = Wed, 10 Dec 2014 19:31:14 GMT
Set-Cookie = JSESSIONID=D70B87DBB832820CACA5998C90939D48; Path=/
Content-Type = text/html
Cache-Control = max-age=2592000
Expires = Fri, 09 Jan 2015 19:31:14 GMT
Vary = Accept-Encoding,User-Agent
Connection = close
Transfer-Encoding = chunked
```

URI class

This class provides methods for creating URI instances from its components or by parsing the string form of those components, for accessing and retrieving different components of a URI instance.

What is URI?

URI stands for Uniform Resource Identifier. A Uniform Resource Identifier is a sequence of characters used for identification of a particular resource. It enables for the interaction of the representation of the resource over the network using specific protocols.

URI Syntax :

scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]

Example:

```
import java.net.*;

class URIDemo2
{
 public static void main (String [] args) throws Exception
 {
  if (args.length != 1)
  {
    System.err.println ("usage: java URIDemo2 uri");
    return;
  }

  URI uri = new URI (args [0]);

  System.out.println ("Normalized URI = " +
            uri.normalize ().toString ());
 }
}
```

Output:

```
Normalized URI = x/z/q
```

Cookies

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.

Types of Cookie

There are 2 types of cookies in servlets.

Non-persistent cookie
Persistent cookie

Non-persistent cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

Simplest technique of maintaining the state.

Cookies are maintained at client side.

Disadvantage of Cookies

It will not work if cookie is disabled from the browser.

Only textual information can be set in Cookie object.

Example:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    Cookie ck[]=request.getCookies();
    out.print("Hello "+ck[0].getValue());

    out.close();

        }catch(Exception e){System.out.println(e);}
    }
}
```

Datagram

A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

Datagrams plays a vital role as an alternative.
Datagrams are bundles of information passed between machines. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response and it is crucial point to note.
Java implements datagrams on top of the UDP (User Datagram Protocol) protocol by using two classes:

DatagramPacket object is the data container.
DatagramSocket is the mechanism used to send or receive the DatagramPackets.

DatagramSocket Class

DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException : It creates a DatagramSocket bound to any unused port on the local computer.
DatagramSocket(int port) throws SocketException : It creates a DatagramSocket bound to the port specified by port.
DatagramSocket(int port, InetAddress ipAddress) throws SocketException : It constructs a DatagramSocket bound to the specified port and InetAddress.
DatagramSocket(SocketAddress address) throws SocketException : It constructs a DatagramSocket bound to the specified SocketAddress.

DatagramPacket Class

DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte data[ ], int size) : It specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket
DatagramPacket(byte data[ ], int offset, int size) : It allows you to specify an offset into the buffer at which data will be stored.
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port) : It specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port) : It transmits packets beginning at the specified offset into the data.

Example:

```
import java.net.*;
import java.io.*;

public class Sender {
	public static void main(String[] args) {
		try {
			BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

			System.out.println("Enter the content to be send: ");

			String content = br.readLine();
			DatagramPacket packet = new DatagramPacket(content.getBytes(),
content.getBytes().length,
					InetAddress.getLocalHost(), 8888);
			DatagramSocket socket = new DatagramSocket();
			socket.send(packet);
			System.out.println("packet is sent successfully...");
		} catch (Exception e) {
			e.printStackTrace();
		}

	}
}
```

Output:

```
Enter the content to be send:
onlyjavatech
packet is sent successfully...
```

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

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

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

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

Example:

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
b.addActionListener(this);
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

Delegation event model

The event model is based on the Event Source and Event Listeners. Event Listener is an object that receives the messages / events. The Event Source is any object which creates the message / event. The Event Delegation model is based on – The Event Classes, The Event Listeners, Event Objects.

There are three participants in event delegation model in Java;
Event Source – the class which broadcasts the events

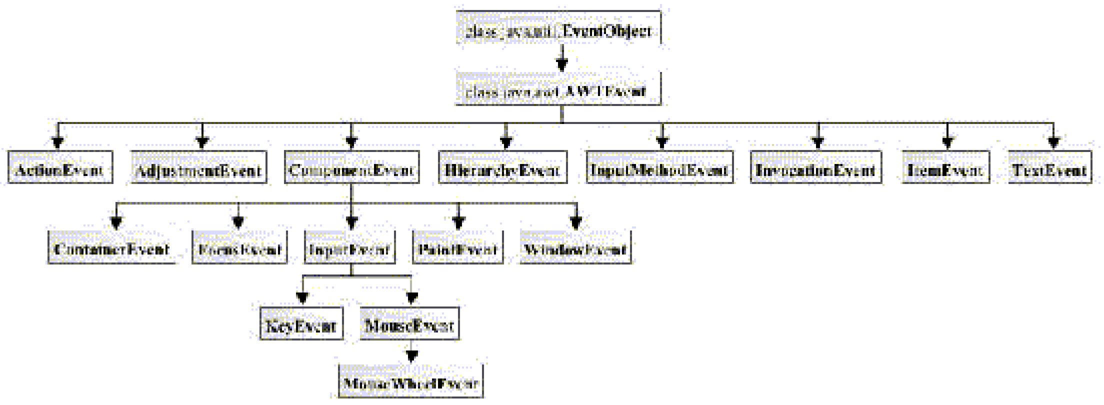
Event Listeners – the classes which receive notifications of events

EventObject– the class object which describes the event.

An event occurs (like mouse click, key press, etc) which is followed by the event is broadcasted by the event source by invoking an agreed method on all event listeners. The event object is passed as argument to the agreed-upon method. Later the event listeners respond as they fit, like submit a form, displaying a message / alert etc

Event Classes

Java has a number of classes that describe different categories of events. The following figure shows the hierarchy of a Java event class :



The Action Event class

? ActionEvent is generated by an AWT component, such as a button, when a component-specific action isperformed.
? The action event is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
? The following syntax shows the declaration of the constructor of the ActionEvent class is:

```
public ActionEvent(Object source, int id, String command)
```

? The main methods included in the Action Event class are:
? String getActionCommand

The MouseEvent class

? The MouseEvent class extends the java.awt.event.InputEvent class.
? The mouse event indicates that a mouse action has occurred on a component.

?The mouse events include:
? Pressing a mouse button
? Releasing a mouse button
? Clicking a mouse button
? Entering of mouse in a component area
? Exiting of mouse from a component area
? The mouse event class defines some integer constants that can be used to identify several types of mouse events.

Given below the various integer constants of the Event class :

Constants	**Description**
MOUSE_CLICKED	Identifies the event of mouse clicking.
MOUSE_DRAGGED	Identifies the event of dragging of mouse.
MOUSE_MOVED	Identifies the event of mouse moving.
MOUSE_PRESSED	Identifies the event of mouse pressing.
MOUSE_RELEASED	Identifies the event of mouse releasing.
MOUSE_ENTERED	Identifies the event of mouse entering an AWT component.
MOUSE_EXITED	Identifies the event of mouse exiting an AWT component.

Given below Various methods of the MouseEvent class :

Methods	**Description**
public int getX()	Returns the horizontal x coordinate of the mouse position relative to a source component.
public int getY()	Returns the vertical y coordinate of the mouse position relative to a source component.
public point getPoint()	Returns the Point object. The Point object contains the x and y coordinates of the mouse position relative to a source component.
public void translatePoint(int x, int y)	Translates the coordinates of a mouse event to a new position by adding x and

	y offsets.
public int getClickCount()	Returns the number of mouse clicks associated with an event.

Events and event source

As you have seen from the ShowActionEvent program, when you run Java GUI programs, the program allows users to interact with it. The events generated from the interaction drive the program's execution.

Events are generated by external user actions such as mouse movements, mouse clicks, and keystrokes, or by the operating system, such as a timer. An event can be defined as a type of signal to the program that something has happened. There are different kinds of events, e.g. button pressed, item selected, window closed. Java has pre-defined classes to represent different kinds of events. Each kind of event is defined as a class. For example, ActionEvent defines events such as pressing a button; WindowEvent defines events such as closing or opening a window; KeyEvent defines events such as pressing or releasing a key. An event is an object of an event class.

The object in which an event is generated is called the source object. In the ShowActionEvent example, the two button objects are the source objects of the events generated when the buttons are pressed. An event object contains relevant properties to the event. For example, you can identify the source object of an event using the getSource() method in the event class. The following list shows the relationship between some user actions, source objects and related types of events.

User Action	**Source Object**	**Generated Event Type**
Click a button	JButton	ActionEvent
Press return on a text field	JTextField	ActionEvent
Select an item	JList	ListSelectionEvent

Event Listener Interface

The Event listeners represent the interfaces which are responsible to handle a particular event.

Java have provided us various Event listeners classes and Every method of an event listener has a single and only argument as an object which is the subclass of EventObject class.

For example- the mouse event listener method will accept all instances of MouseEvent, where MouseEvent derived from the EventObject.

The EventListner interface is a interface where every listener interface has to be extend.

This class is defined in the java.util package.

Class declaration

The Following is the declaration for java.util.EventListener interface as

public interface EventListener

AWT Event Listener Interfaces

The Following is the list of commonly used event listeners as

.

Sr. No.	**Control & Description**
1	ActionListener This interface is used for receiving the action events.
2	ComponentListener This interface is used for receiving the component events.
3	ItemListener This interface is used for receiving the item events.
4	KeyListener This interface is used for receiving the key events.
5	MouseListener This interface is used for receiving the mouse events.
6	TextListener This interface is used for receiving the text events.
7	WindowListener

	This interface is used for receiving the window events.
8	AdjustmentListener This interface is used for receiving the adjusmtent events.
9	ContainerListener This interface is used for receiving the container events.
10	MouseMotionListener This interface is used for receiving the mouse motion events.
11	FocusListener This interface is used for receiving the focus events.

Java Adapter Classes

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event** packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

Adapter class	**Listener interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener

ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

Example:

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
    Frame f;
    AdapterExample(){
        f=new Frame("Window Adapter");
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                f.dispose();
            }
        });
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String[] args) {
    new AdapterExample();
}
}
```

Java Inner Classes

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Example:

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output:

```
data is 30
```

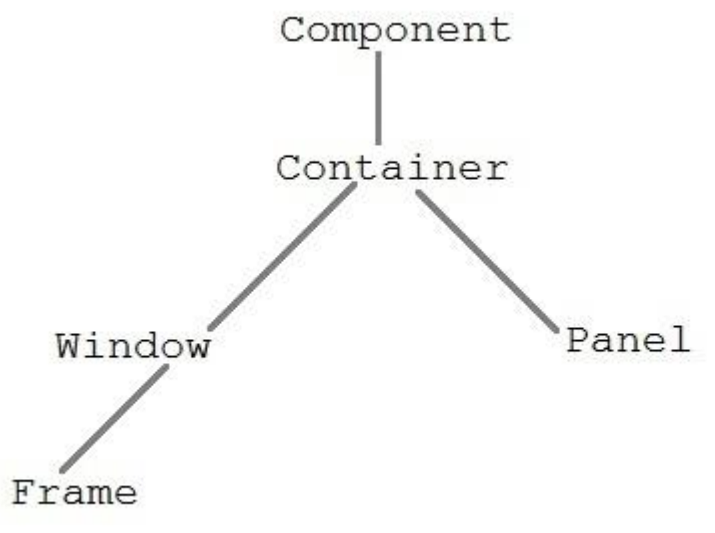
Unit-II

Java Abstract Window Toolkit(AWT)

AWT contains large number of classes and methods that allows you to create and manage graphical user interface (GUI) applications, such as windows, buttons, scroll bars,etc. The AWT was designed to provide a common set of tools for GUI design that could work on a variety of platforms. The tools provided by the AWT are implemented using each platform's native GUI toolkit, hence preserving the look and feel of each platform. This is an advantage of using AWT.But the disadvantage of such an approach is that GUI designed on one platform may look different when displayed on another platform.

AWT is the foundation upon which Swing is made i.e Swing is a set of GUI interfaces that extends the AWT. But now a days AWT is merely used because most GUI Java programs are implemented using Swing because of its rich implementation of GUI controls and light-weighted nature.

Java AWT Hierarchy



Component class

Component class is at the top of AWT hierarchy. Component is an abstract class that encapsulates all the attributes of visual component. A component object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

Container is a component in AWT that contains another component like button, text field, tables etc. **Container** is a subclass of component class. **Container** class keeps track of components that are added to another component.

Panel

Panel class is a concrete subclass of **Container**. Panel does not contain title bar, menu bar or border. It is container that is used for holding components.

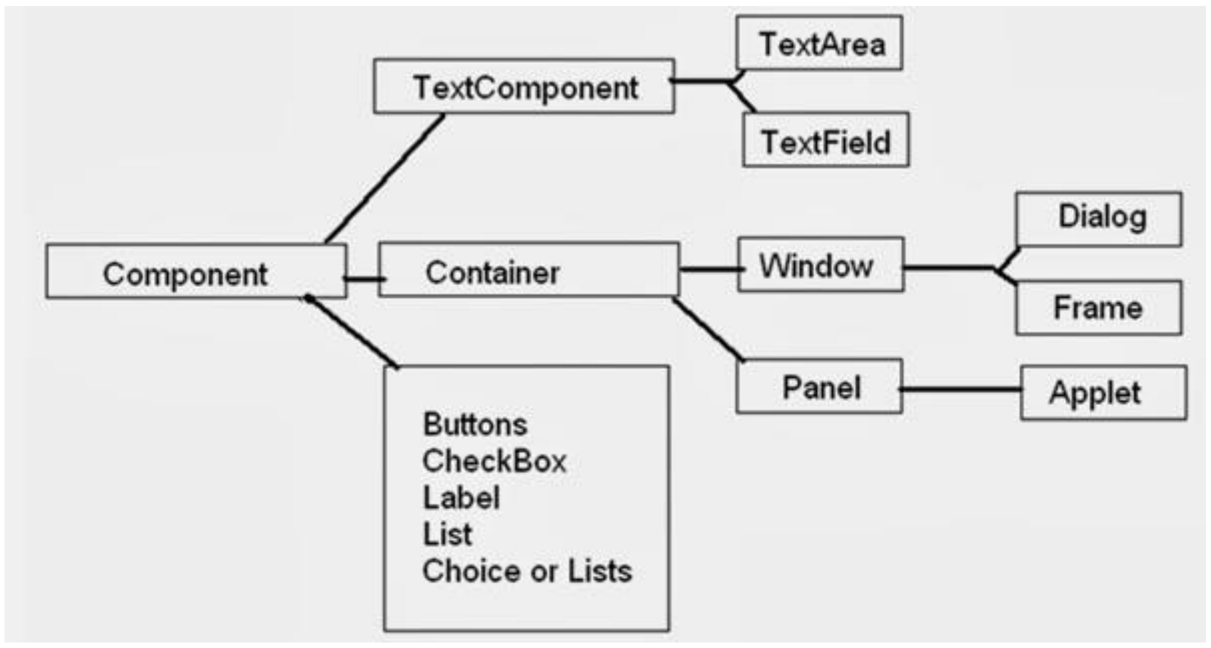
WINDOW FUNDAMENTALS

Java Abstract window tool kit package is used for displaying the data within a GUI Environment. Features of AW T Package are as Followings:

1. It provides us a set of user interface components including windows buttons text fields scrolling list etc.
2. It provides us the way to laying out these above components.
3. It provides to create the events upon these components.

The main purpose for using the AWT is using for all the components displaying on the screen. Awt defines all the windows according to a class hierarchy those are useful at a specific level or we can say arranged according to their functionality.

The most commonly used interface is the panels those are used by applets and those are derived from frame which creates a standard window. The hierarchy of this Awt is:-



As you see this Figure this Hierarchical view display all the classes and also their Sub Classes. All the Classes are Contained as a Multi-Level inheritance , As You see that Component Class contains a Panel Class which again Contains a Applet Class so that in the AWT Package are Stored in the form of Multi-values inheritance.

Working With Frames

Frame Class is also a Sub Class of Window Class and frame class allows to Create a pop-Menus. And Frame Class Provides a Special Type of Window which has a title bar, menu bar , border. Resizing corners and it is a subclass of window class when we creates A frame object within a applet , will display a warning message that a frame window is created by an applet not by any software. But when a frame window is created by a program other than an applet then a normal window is created. The Various Methods those are Provided by the Frame Class are as follows :-

1. **Frame()** : This is name of Class and used for Creating an object of this Class and this will not display any title on the title bar of Frame.
2. **Frame(String Title)** : This will also create an object of Frame Class and this will also Display a Title on the Title bar of the Window.
3. **Void setsize()** : this is Method of Frame Class which will be Accessed by the object of Frame Class and this will takes two Arguments to set the Size in the Form of width and Height .
4. **Void show()** : This is used for displaying or showing a Frame from a Main Window.
5. **Void setBackground(Color c)** : This is used to set the Background Color of the frame.
6. **Void setLocation(int x,int y)** : This Method will display the frame on the window according to the values of x and y coordinates those are Specified Always Remember that For Displaying a Frame you have to set the Layout and the default Layout of the Frame is Border Layout.

Creating a Frame

There are two ways to create a Frame. They are,

By Instantiating Frame class

By extending Frame class

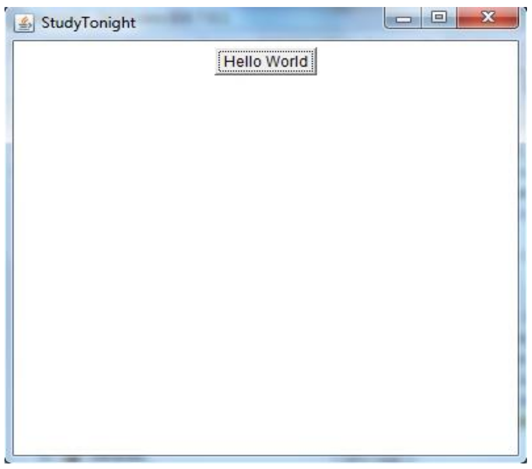
Creating Frame Window by Instantiating Frame class

```
import java.awt.*;

public class Testawt

{

  Testawt()

  {

    Frame fm=new Frame();    //Creating a frame

    Label lb = new Label("welcome to java graphics");   //Creating a label

    fm.add(lb);

    fm.setSize(300, 300);

    fm.setVisible(true);

  }

  public static void main(String args[])

  {

    Testawt ta = new Testawt();

  }

}
```

Creating Frame window by extending Frame class

```
package testawt;

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

public class Testawt extends Frame

{

   public Testawt()

   {

      Button btn=new Button("Hello World");

      add(btn);

      setSize(400, 500);

      setTitle("StudyTonight");

      setLayout(new FlowLayout());

      setVisible(true);

   }

   public static void main (String[] args)

   {

      Testawt ta = new Testawt();   //creating a frame.

   }

}
```

Output:

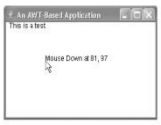


Creating a Windowed Program

Although creating applets is a common use for Java's AWT, it is also possible to create stand-alone AWT-based applications. To do this, simply create an instance of the window or windows you need inside **main( )**. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Create a frame window.
public class AppWindow extends Frame {
String keymsg = "This is a test.";
String mousemsg = "";
int mouseX=30, mouseY=30;
public AppWindow() {
addKeyListener(new MyKeyAdapter(this));
addMouseListener(new MyMouseAdapter(this));
addWindowListener(new MyWindowAdapter());
}
public void paint(Graphics g) {
g.drawString(keymsg, 10, 40);
g.drawString(mousemsg, mouseX, mouseY);
}
// Create the window.
public static void main(String args[]) {
AppWindow appwin = new AppWindow();
appwin.setSize(new Dimension(300, 200));
appwin.setTitle("An AWT-Based Application");
appwin.setVisible(true);
}
}
class MyKeyAdapter extends KeyAdapter {
AppWindow appWindow;
public MyKeyAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void keyTyped(KeyEvent ke) {
appWindow.keymsg += ke.getKeyChar();
appWindow.repaint();
};
}
class MyMouseAdapter extends MouseAdapter {
AppWindow appWindow;
public MyMouseAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void mousePressed(MouseEvent me) {
appWindow.mouseX = me.getX();
appWindow.mouseY = me.getY();
appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
", " + appWindow.mouseY;
appWindow.repaint();
}
}
class MyWindowAdapter extends WindowAdapter {
public void windowClosing(WindowEvent we) {
System.exit(0);
}
}
```

Output:



```
This is a test
Mouse Down at 81, 97
```

Displaying Information Within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window's ability to present high quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items. For this reason, the remainder of this chapter discusses Java's text-, graphics-, and font handling capabilities.
As you will they are both powerful and flexible.

Working with Graphics

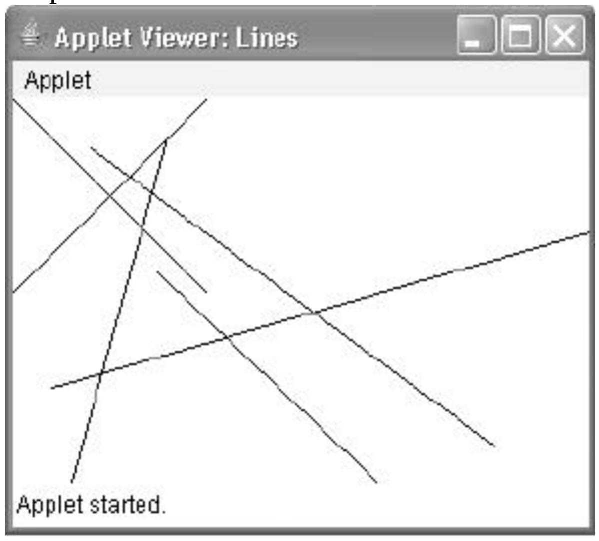
The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:
· It is passed to an applet when one of its various methods, such as **paint( )** or **update( )**, is called.
· It is returned by the **getGraphics( )** method of **Component**.
For the sake of convenience the remainder of the examples in this chapter will demonstrate graphics in the main applet window. However, the same techniques will apply to any other window.
The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

Drawing Lines

Lines are drawn by means of the **drawLine( )** method, shown here:
void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)
drawLine( ) displays a line in the current drawing color that begins at *startX*,*startY* and ends at *endX*,*endY*.
The following applet draws several lines:

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
}
}
```

Output:



Drawing Rectangles

The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively.
They are shown here:
void drawRect(int *top*, int *left*, int *width*, int *height*)
void fillRect(int *top*, int *left*, int *width*, int *height*)

The upper-left corner of the rectangle is at *top*,*left*. The dimensions of the rectangle are specified by *width* and *height*.

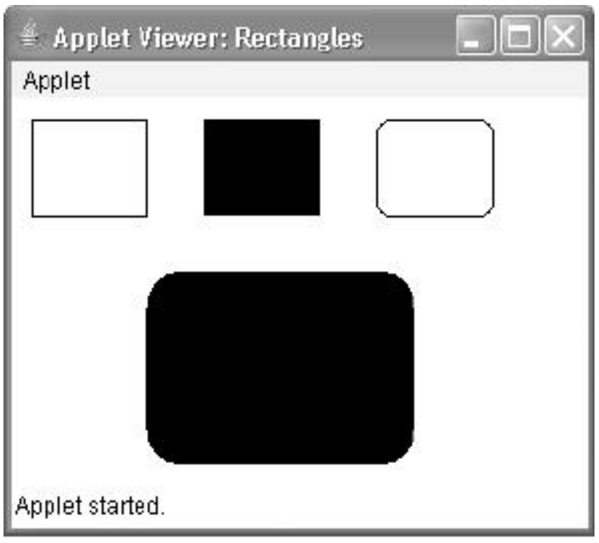
To draw a rounded rectangle,use **drawRoundRect()** or **fillRoundRect()**,
both shown here:
void drawRoundRect(int *top*, int *left*, int *width*, int *height*,int *xDiam*, int *yDiam*)
void fillRoundRect(int *top*, int *left*, int *width*, int *height*,int *xDiam*, int *yDiam*)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top*,*left*.
The dimensions of the rectangle are specified by *width* and *height*. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

The following applet draws several rectangles:

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
g.drawRect(10, 10, 60, 50);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}
```

Output:



Drawing Ellipses and Circles

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown here:

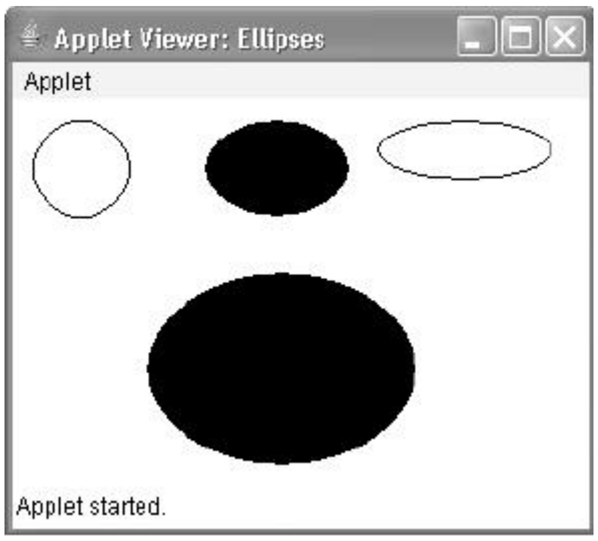
void drawOval(int *top*, int *left*, int *width*, int *height*)
void fillOval(int *top*, int *left*, int *width*, int *height*)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle.

The following program draws several ellipses:

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
}
}
```

Output:



Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.
Color defines several constants (for example, **Color.black**) to
specify a number of common colors. You can also create your own colors, using one of the
color constructors. Three commonly used forms are shown here:
Color(int *red*, int *green*, int *blue*)
Color(int *rgbValue*)
Color(float *red*, float *green*, float *blue*)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:
new Color(255, 100, 100); // light red
The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
The final constructor, **Color(float, float, float)**, takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.
Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground()** and **setBackground( )** methods

Example:

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

public class AWTGraphicsDemo extends Frame {

  public AWTGraphicsDemo(){
    super("Java AWT Examples");
    prepareGUI();
  }

  public static void main(String[] args){
    AWTGraphicsDemo  awtGraphicsDemo = new AWTGraphicsDemo();
    awtGraphicsDemo.setVisible(true);
  }

  private void prepareGUI(){
    setSize(400,400);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
```

```
            }
        });
    }

    @Override
    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        Font plainFont = new Font("Serif", Font.PLAIN, 24);
        g2.setFont(plainFont);
        g2.setColor(Color.red);
        g2.drawString("Welcome to TutorialsPoint", 50, 70);
        g2.setColor(Color.GRAY);
        g2.drawString("Welcome to TutorialsPoint", 50, 120);
    }
}
```

Working with Font

The Font class provides a method of specifying and using fonts. The Font class constructor constructs font objects using the font's name, style (PLAIN, BOLD, ITALIC, or BOLD + ITALIC), and point size. Java's fonts are named in a platform independent manner and then mapped to local fonts that are supported by the operating system on which it executes. The getName() method returns the logical Java font name of a particular font and the getFamily() method returns the operating system-specific name of the font. The standard Java font names are Courier, Helvetica, TimesRoman etc.

The font can be set for a graphics context and for a component.

Font getFont() It is a method of Graphics class used to get the font property

setFont(Font f) is used to set a font in the graphics context

There are following logical font names which are standard on all platforms and are mapped to actual fonts on a particular platform:

"Serif" variable pitch font with serifs

"SansSerif" variable pitch font without serifs

"Monospaced" fixed pitch font

"Dialog" font for dialogs

"DialogInput" font for dialog input

"Symbol" mapped to the Symbol font

Font style is specified using constants from the Font class:

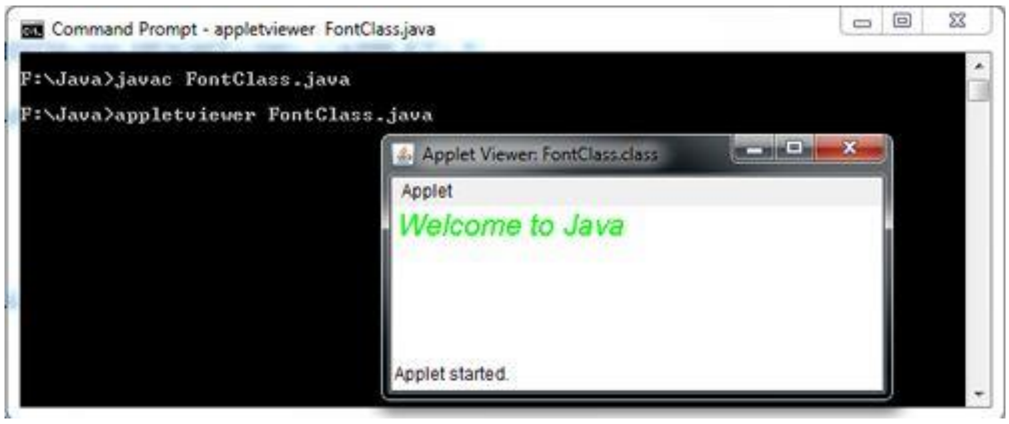
Font.BOLD

Font.ITALIC

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET CODE ="FontClass.class" WIDTH=300 HEIGHT=200> </APPLET> */
public class FontClass extends java.applet.Applet
```

```
 {
    Font f;
    String m;
      public void init()
        {
           f=new Font("Arial",Font.ITALIC,20);
           m="Welcome to Java";
           setFont(f);
        }
           public void paint(Graphics g)
              {
                 Color c=new Color(0,255,0);
                 g.setColor(c);
                 g.drawString(m,4,20);
              }
}
```

Output:



```
F:\Java>javac FontClass.java

F:\Java>appletviewer FontClass.java
```

Command Prompt - appletviewer FontClass.java
Applet Viewer: FontClass.class
Applet
Welcome to Java
Applet started.

AWT Controls

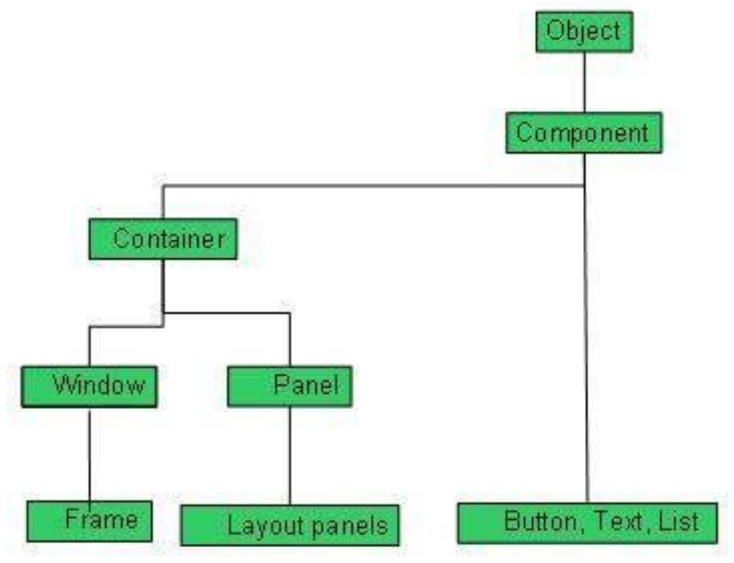
Control Fundamentals:

Every user interface considers the following three main aspects:

UI elements : Thes are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.

Layouts: They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in Layout chapter.

Behavior: These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.



Every AWT controls inherits properties from Component class.

Sr. No.	Control & Description
1	Component A Component is an abstract super class for GUI controls and it represents an object with graphical representation.

AWT UI Elements:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	Label A Label object is a component for placing text in a container.
2	Button This class creates a labeled button.
3	Check Box A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.

4	Check Box Group The CheckboxGroup class is used to group the set of checkbox.
5	List The List component presents the user with a scrolling list of text items.
6	Text Field A TextField object is a text component that allows for the editing of a single line of text.
7	Text Area A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	Choice A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
9	Canvas A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
10	Image An Image control is superclass for all image classes representing graphical images.
11	Scroll Bar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
12	Dialog A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
13	File Dialog A FileDialog control represents a dialog window from which the user can select a file.

Label

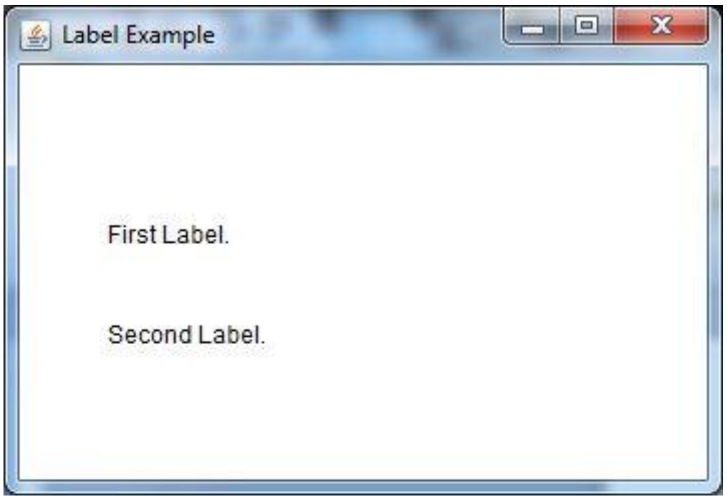
The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

AWT Label Class Declaration
public class Label extends Component implements Accessible

Example

```
import java.awt.*;
class LabelExample{
public static void main(String args[]){
   Frame f= new Frame("Label Example");
   Label l1,l2;
   l1=new Label("First Label.");
   l1.setBounds(50,100, 100,30);
   l2=new Label("Second Label.");
   l2.setBounds(50,150, 100,30);
   f.add(l1); f.add(l2);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}
}
```

Output:



Button

The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

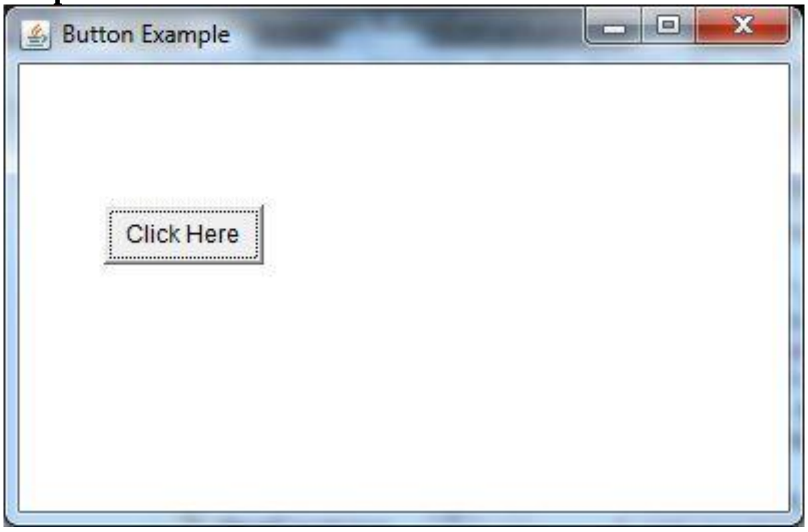
AWT Button Class declaration

public class Button extends Component implements Accessible

Example

```
import java.awt.*;
public class ButtonExample {
public static void main(String[] args) {
    Frame f=new Frame("Button Example");
    Button b=new Button("Click Here");
    b.setBounds(50,100,80,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Java AWT Checkbox

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

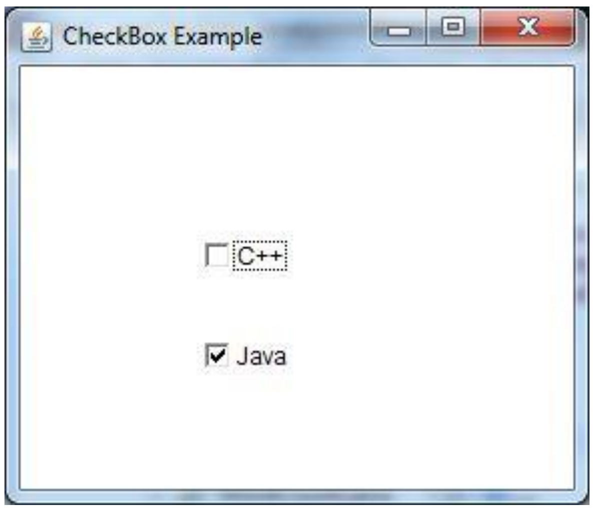
AWT Checkbox Class Declaration

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

Example

```
import java.awt.*;
public class CheckboxExample
{
     CheckboxExample(){
       Frame f= new Frame("Checkbox Example");
        Checkbox checkbox1 = new Checkbox("C++");
        checkbox1.setBounds(100,100, 50,50);
        Checkbox checkbox2 = new Checkbox("Java", true);
        checkbox2.setBounds(100,150, 50,50);
        f.add(checkbox1);
        f.add(checkbox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
    new CheckboxExample();
}
}
```

Output:



Choice Control

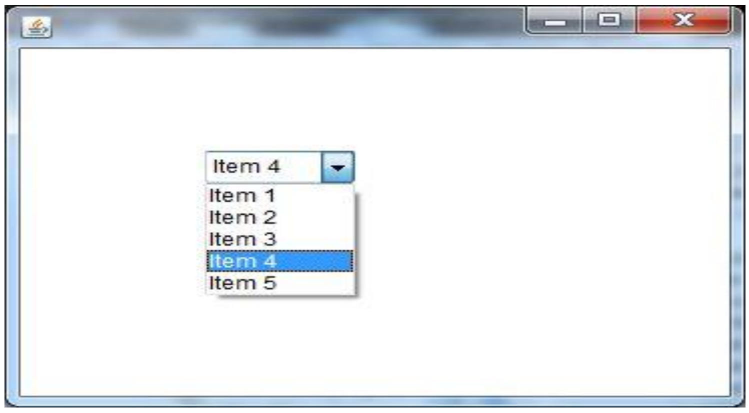
The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

AWT Choice Class Declaration
public class Choice extends Component implements ItemSelectable, Accessible

Example

```
import java.awt.*;
public class ChoiceExample
{
        ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
  new ChoiceExample();
}
}
```

Output:



List

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

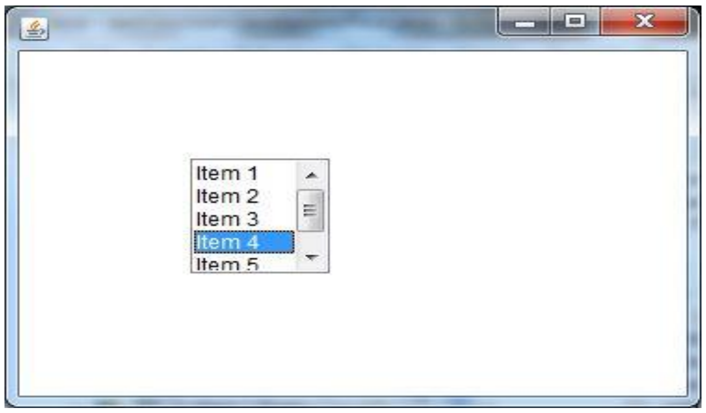
AWT List class Declaration

```
public class List extends Component implements ItemSelectable, Accessible
```

Example

```
import java.awt.*;
public class ListExample
{
    ListExample(){
       Frame f= new Frame();
       List l1=new List(5);
       l1.setBounds(100,100, 75,75);
       l1.add("Item 1");
       l1.add("Item 2");
       l1.add("Item 3");
       l1.add("Item 4");
       l1.add("Item 5");
       f.add(l1);
       f.setSize(400,400);
       f.setLayout(null);
       f.setVisible(true);
    }
public static void main(String args[])
{
  new ListExample();
} }
```

Output:



Scrollbar

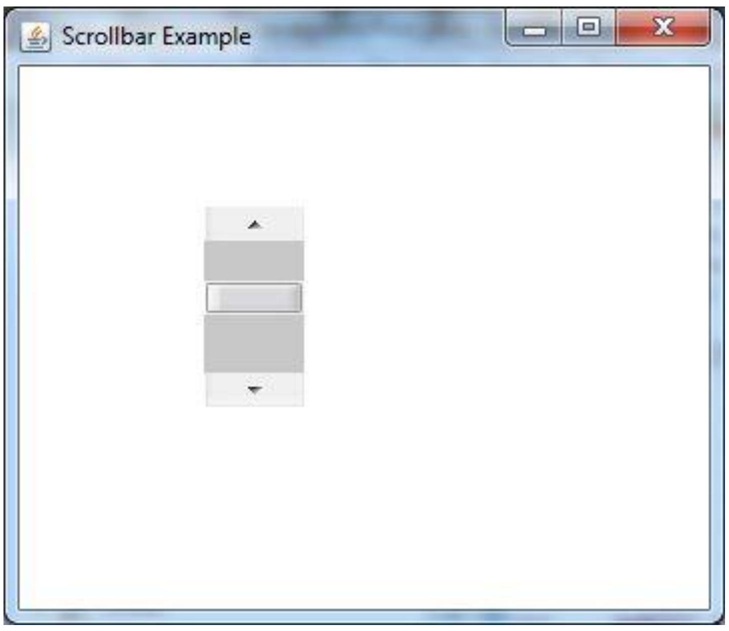
The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

AWT Scrollbar class declaration
public class Scrollbar extends Component implements Adjustable, Accessible

Example

```
import java.awt.*;
class ScrollbarExample{
ScrollbarExample(){
        Frame f= new Frame("Scrollbar Example");
        Scrollbar s=new Scrollbar();
        s.setBounds(100,100, 50,100);
        f.add(s);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[]){
    new ScrollbarExample();
}
}
```

Output:



TextField

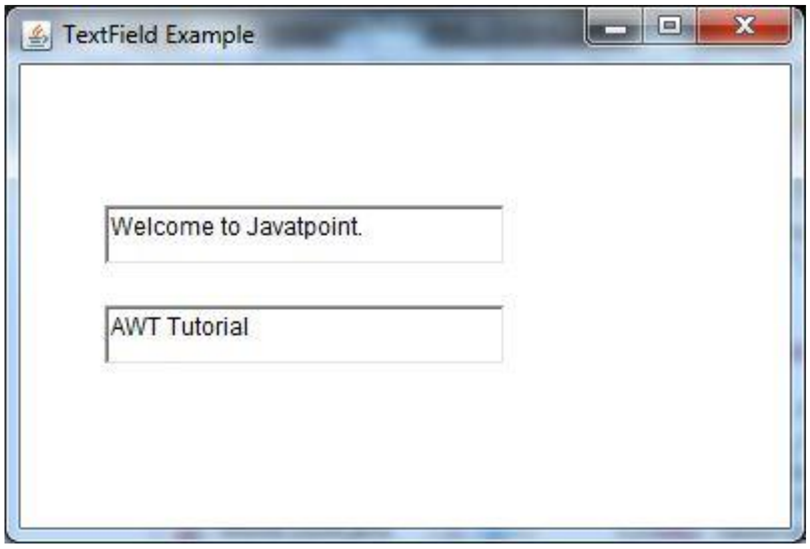
The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

AWT TextField Class Declaration
public class TextField extends TextComponent

Example

```
import java.awt.*;
class TextFieldExample{
public static void main(String args[]){
    Frame f= new Frame("TextField Example");
    TextField t1,t2;
    t1=new TextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new TextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

java.awt.BorderLayout

java.awt.FlowLayout

java.awt.GridLayout

java.awt.CardLayout

java.awt.GridBagLayout

javax.swing.BoxLayout

javax.swing.GroupLayout

javax.swing.ScrollPaneLayout

javax.swing.SpringLayout etc.

BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region:

public static final int NORTH

public static final int SOUTH

public static final int EAST

public static final int WEST

public static final int CENTER

Constructors of BorderLayout class:

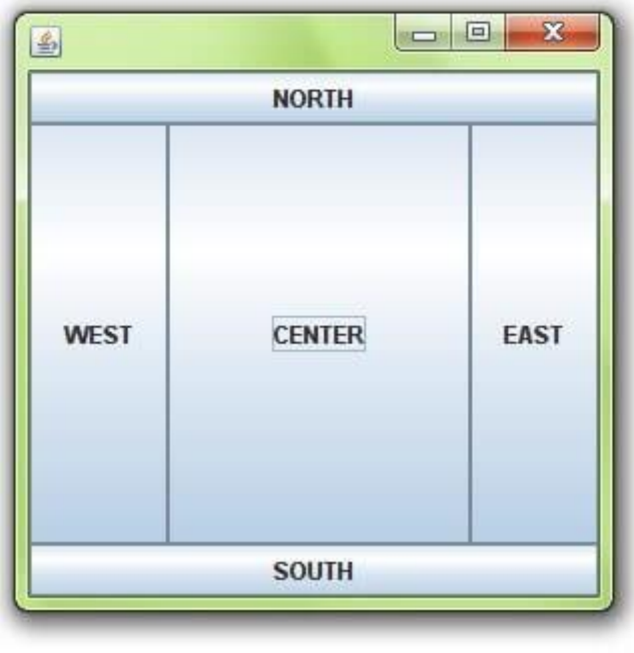
BorderLayout(): creates a border layout but with no gaps between the components.

JBorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Example:

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

public class Border {
JFrame f;
Border(){
   f=new JFrame();

   JButton b1=new JButton("NORTH");;
   JButton b2=new JButton("SOUTH");;
   JButton b3=new JButton("EAST");;
   JButton b4=new JButton("WEST");;
   JButton b5=new JButton("CENTER");;

   f.add(b1,BorderLayout.NORTH);
   f.add(b2,BorderLayout.SOUTH);
   f.add(b3,BorderLayout.EAST);
   f.add(b4,BorderLayout.WEST);
   f.add(b5,BorderLayout.CENTER);

   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new Border();
}
}
```

Output:



CardLayout

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class

CardLayout(): creates a card layout with zero horizontal and vertical gap.

CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class

public void next(Container parent): is used to flip to the next card of the given container.

public void previous(Container parent): is used to flip to the previous card of the given container.

public void first(Container parent): is used to flip to the first card of the given container.

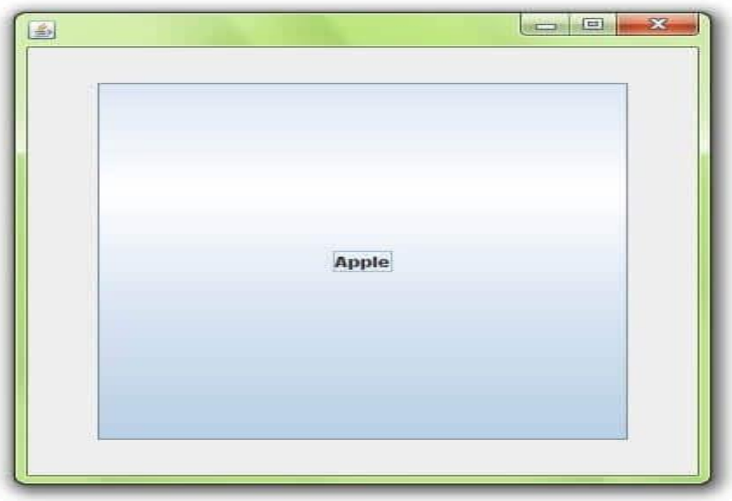
public void last(Container parent): is used to flip to the last card of the given container.

public void show(Container parent, String name): is used to flip to the specified card with the given name.

Example:

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

import javax.swing.*;

public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
    CardLayoutExample(){

        c=getContentPane();
        card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
        c.setLayout(card);

        b1=new JButton("Apple");
        b2=new JButton("Boy");
        b3=new JButton("Cat");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        c.add("a",b1);c.add("b",b2);c.add("c",b3);

    }
    public void actionPerformed(ActionEvent e) {
    card.next(c);
    }

    public static void main(String[] args) {
        CardLayoutExample cl=new CardLayoutExample();
        cl.setSize(400,400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

Output:



GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

GridLayout(): creates a grid layout with one column per component in a row.

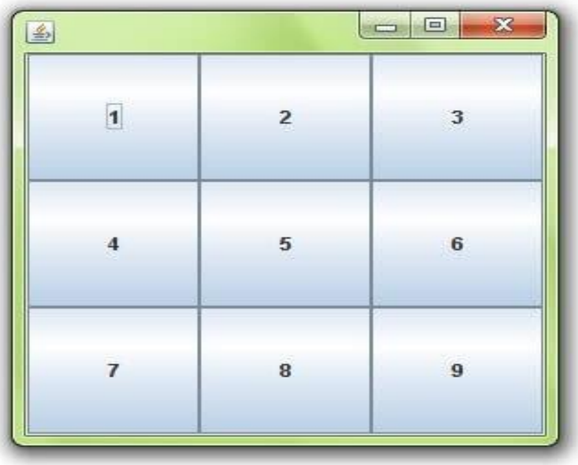
GridLayout(int rows, int columns): creates a grid layout with the given rows and columns but no gaps between the components.

GridLayout(int rows, int columns, int hgap, int vgap): creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example:

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

public class MyGridLayout{
JFrame f;
MyGridLayout(){
   f=new JFrame();

   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
      JButton b6=new JButton("6");
      JButton b7=new JButton("7");
   JButton b8=new JButton("8");
      JButton b9=new JButton("9");
   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
   f.add(b6);f.add(b7);f.add(b8);f.add(b9);
   f.setLayout(new GridLayout(3,3));
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyGridLayout();
}
}
```

Output:



FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Fields of FlowLayout class

public static final int LEFT

public static final int RIGHT

public static final int CENTER

public static final int LEADING

public static final int TRAILING

Constructors of FlowLayout class

FlowLayout(): creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.

FlowLayout(int align): creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.

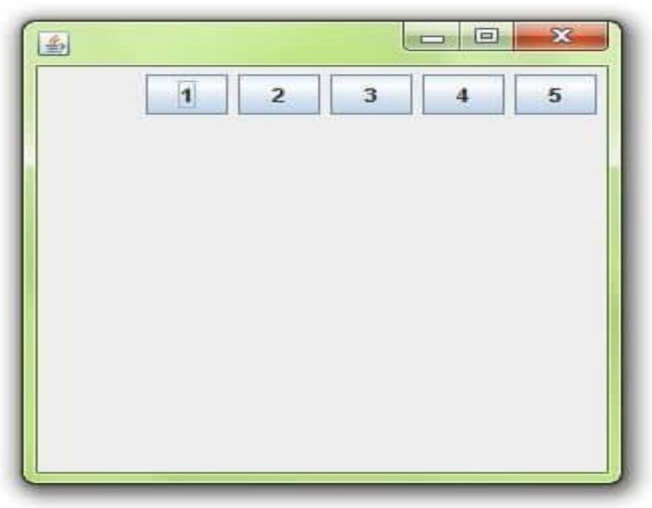
FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example:

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

public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
    f=new JFrame();

    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");

    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);

    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment
```

```
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyFlowLayout();
}
}
```

Output:



MenuBars and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

public class MenuItem **extends** MenuComponent **implements** Accessible

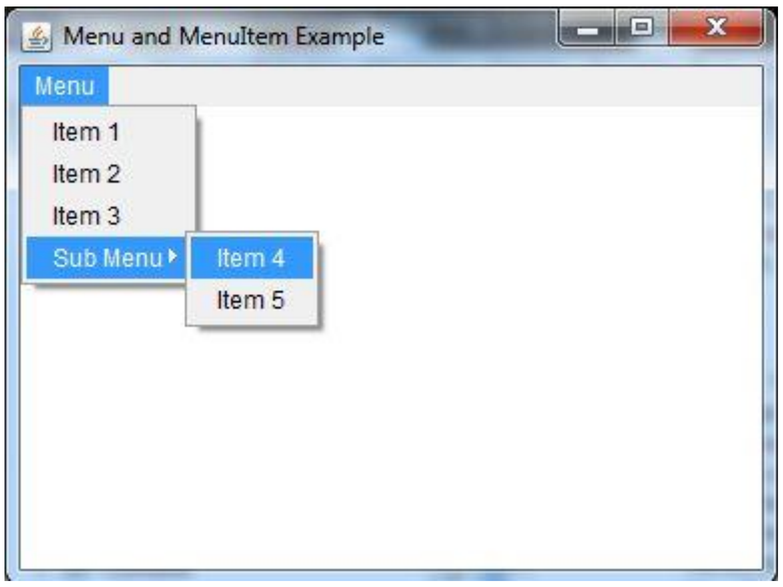
AWT Menu class declaration

public class Menu **extends** MenuItem **implements** MenuContainer, Accessible

Example:

```
import java.awt.*;
class MenuExample
{
    MenuExample(){
       Frame f= new Frame("Menu and MenuItem Example");
       MenuBar mb=new MenuBar();
       Menu menu=new Menu("Menu");
       Menu submenu=new Menu("Sub Menu");
       MenuItem i1=new MenuItem("Item 1");
       MenuItem i2=new MenuItem("Item 2");
       MenuItem i3=new MenuItem("Item 3");
       MenuItem i4=new MenuItem("Item 4");
       MenuItem i5=new MenuItem("Item 5");
       menu.add(i1);
       menu.add(i2);
       menu.add(i3);
       submenu.add(i4);
       submenu.add(i5);
       menu.add(submenu);
       mb.add(menu);
       f.setMenuBar(mb);
       f.setSize(400,400);
       f.setLayout(null);
       f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Output:



Dialogboxes

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

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

Frame vs Dialog

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

AWT Dialog class declaration

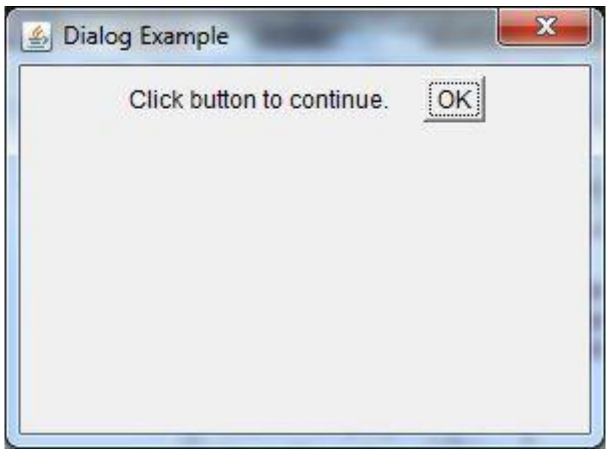
```
public class Dialog extends Window
```

Example

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
    private static Dialog d;
    DialogExample() {
        Frame f= new Frame();
        d = new Dialog(f , "Dialog Example", true);
        d.setLayout( new FlowLayout() );
        Button b = new Button ("OK");
        b.addActionListener ( new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                DialogExample.d.setVisible(false);
            }
        });
        d.add( new Label ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public static void main(String args[])
    {
        new DialogExample();
    }
```

}

Output:



Handling Event

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	**Listener Interfaces**
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener

ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

Register the component with the Listener

Registration Methods

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

Button

public void addActionListener(ActionListener a){}

MenuItem

public void addActionListener(ActionListener a){}

TextField

public void addActionListener(ActionListener a){}

public void addTextListener(TextListener a){}

TextArea

public void addTextListener(TextListener a){}

Checkbox

public void addItemListener(ItemListener a){}

Choice

public void addItemListener(ItemListener a){}

List

public void addActionListener(ActionListener a){}

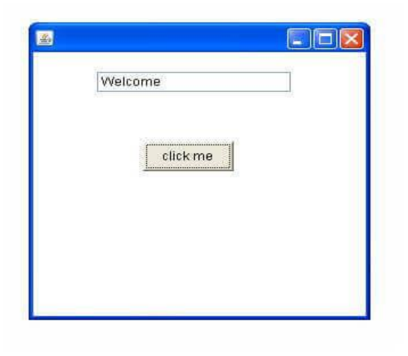
public void addItemListener(ItemListener a){}

Example:

```
import java.awt.event.*;
class Outer implements ActionListener{
AEvent2 obj;
Outer(AEvent2 obj){
this.obj=obj;
}
public void actionPerformed(ActionEvent e){
```

```
obj.tf.setText("welcome");
}
}
```

Output:



FileDialog

Introduction

FileDialog control represents a dialog window from which the user can select a file.

Class declaration

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

```
public class FileDialog
   extends Dialog
```

Field

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

static int LOAD -- This constant value indicates that the purpose of the file dialog window is to locate a file from which to read.

static int SAVE -- This constant value indicates that the purpose of the file dialog window is to locate a file to which to write.

Class constructors

S.N.	Constructor & Description
1	**FileDialog(Dialog parent)**

	Creates a file dialog for loading a file.
2	**FileDialog(Dialog parent, String title)** Creates a file dialog window with the specified title for loading a file.
3	**FileDialog(Dialog parent, String title, int mode)** Creates a file dialog window with the specified title for loading or saving a file.

Class methods

S.N.	Method & Description
1	**void addNotify()** Creates the file dialog's peer.
2	**String getDirectory()** Gets the directory of this file dialog.
3	**String getFile()** Gets the selected file of this file dialog.
4	**FilenameFilter getFilenameFilter()** Determines this file dialog's filename filter.

Example:

```
public File showOpenDialog() {

    JFileChooser chooser = createFileChooser();

    if( Boolean.getBoolean("nb.native.filechooser") ) { //NOI18N

        FileDialog fileDialog = createFileDialog( chooser.getCurrentDirectory() );

        if( null != fileDialog ) {

            return showFileDialog(fileDialog, FileDialog.LOAD );
    }
  }
  chooser.setMultiSelectionEnabled(false);
  int dlgResult = chooser.showOpenDialog(findDialogParent());
  if (JFileChooser.APPROVE_OPTION == dlgResult) {
    File result = chooser.getSelectedFile();
    if (result != null && !result.exists()) {
      result = null;
    }
    return result;
  } else {
    return null;
  }
```

Unit-III

File Format

The information about the combination of color, pixels, and dimensions make an image file. They need to be stored in memory. As a result, the information has to be represented in an array of bits.

An image file becomes quite large when represented in an simple array of bits. So, a mechanism is devised to compress them so that it not only reduces the file size but also retains visual quality. This is exactly what the encoding technique does. It arranges the bits in a manner that can be stored in a digital medium or transmitted easily in a network. Each encoding technique defines an image file format.

File formats are the data structure on how image information is encoded and stored in a repository. Understandably, these encoding techniques are quite complex and heavily depend upon complex mathematical formulation.

Many of them are standardized and commonly used on the Web. The three most common file formats that we always encounter are JPEG (Joint Photographer Experts Group), GIF (Graphics Interchange Format), and PNG (Portable Network Graphics). There are many others, such as TIFF, Exif, BMP, and so on. Let's stick to only three of them for now:

JPEG files have an extension *jpeg* or *jpg*. They use *lossy compression* based on DCT (Discrete Cosine Transformation) to encode an image. The JPEG encoding technique discards some image information in favor of compression. This makes the file size considerably smaller. The information is discarded in such a manner that the loss of quality is almost unrecognizable to the human eye. Also, there is another lossless JPEG standard, which is not that popular, possibly because there is not much to complain about the quality of images produced by lossy compression of JPEG.

PNG files are used for storing bit-mapped images using lossless data compression. The extension of this file is *png*. It is a raster graphics file format, often used in image processing as a raw data. PNG images are not suitable for print and meant to be used for transferring images through network and display. In fact PNG is an improved substitute of GIF images. While GIF supports animation; the corresponding PNG file format that supports animation is MNG.

GIF files are bitmap images with the extension *gif*. They also use palette-based, lossless data compression. GIF files represent graphics in a manner that can be used to create a type of rudimentary animation. Back in the 1980s, GIF files rocked Web pages with animation. Later, in 1985, when this file format was patented, controversies peaked over its licensing. This led to the emergence of a non-patented PNG file format. GIF files are particularly used for graphics, logos, and small images, much like PNG files, and are actually a predecessor of PNG.

Image Fundamentals

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources.

Creating an Image Object

You might expect that you create a memory image using something like the following:
Image test = new Image(200, 100); // Error -- won't work
Because images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen. Therefore, the **Component** class in **java.awt** has a factory method called **createImage( )** that is used to create **Image** objects.

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

The first form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface.
The second form returns a blank (that is, empty) image that has the specified width and height.

Here is an example:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage()** method to actually make an **Image** object.

Loading an Image

The other way to obtain an image is to load one. One way to do this is to use the **getImage( )** method defined by the **Applet** class. It has the following forms:

Image getImage(URL *url*)
Image getImage(URL *url*, String *imageName*)

The first version returns an **Image** object that encapsulates the image found at the location specified by *url*. The second version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName.*

Displaying an Image
Once you have an image, you can display it by using **drawImage( )**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:
boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper left corner specified by *left* and *top. imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT components.

An *image observer* is an object that can monitor an image while it loads. With **getImage()** and **drawImage( )**, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image. The file **seattle.jpg** is loaded, but you can substitute any GIF, JPG, or PNG file you like

Example:

```
/*
* <applet code="SimpleImageLoad" width=248 height=146>
*  <param name="img" value="seattle.jpg">
* </applet>
*/
import java.awt.*;
import java.applet.*;
public class SimpleImageLoad extends Applet
{
Image img;
public void init() {
img = getImage(getDocumentBase(), getParameter("img"));
}
public void paint(Graphics g) {
g.drawImage(img, 0, 0, this);
}
}
```

ImageObserver

ImageObserver is an interface that provides constants and the callback mechanism to receive asynchronous information about the status of an image as it loads.

public interface **ImageObserver**

imageUpdate(Image img, int infoflags, int x, int y, int width, int height)

The Image class is used to load and display images. To load an image the getimage ()method of the Image class is used and to display the image the draw Image ()method of the Graphics class is used.

The general form of the getImage () method is

Image getimage(URL pathname, String filename)

Image getimage(URL pathname)

where,

pathname is the address of the image file on web. When the image file and the source file are in the same directory, getCodeBase ()method is used as first parameter to the method.

filename is the name of the image file

The general form of the drawimage ()method is

boolean drawimage(Image image, int startx, int starty,

int width, int height, ImageObserver img_obj)

where,

image is the image to be loaded in the applet.

startx is the pixels space from the left comer of the screen.

starty is the pixels space from the upper comer of the screen.

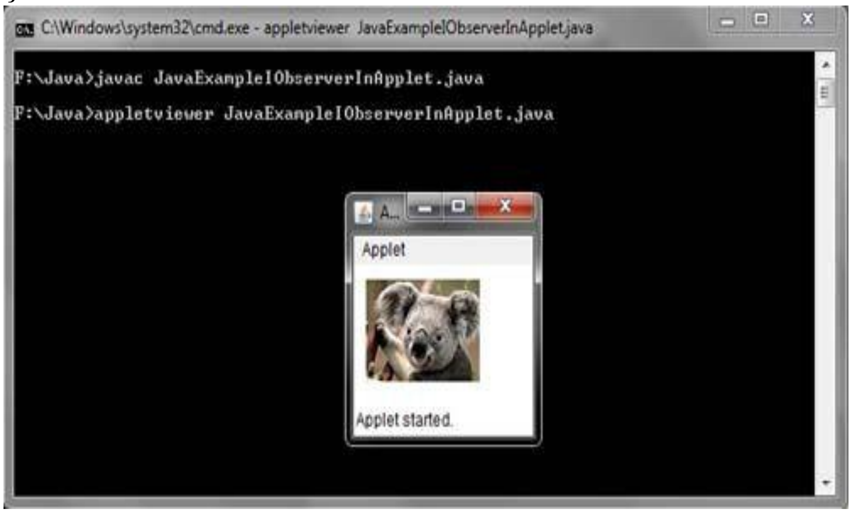
width is the width of the image.

height is the height of the image.

img_obj is object of the class that implements ImageObserver interface.

```
import java.awt.*;
import java.applet.*;
/*<APPLET CODE=JavaExampleIObserverInApplet.class WIDTH=610 HEIGHT=160></APPLET>*/
public class JavaExampleIObserverInApplet extends Applet
 {
   Image img;
   public void init()
   {
      img = getImage(getDocumentBase(),"Koala.jpg");
   }
```

```
    public void paint(Graphics gr)
     {
       gr.drawImage(img,10,10,this);
     }
       public boolean imageUpdate(Image img1,int flags, int xAxis, int yAxis, int width, int height)
      {
        if ((flags & ALLBITS) != 0)
           {
             repaint(xAxis,yAxis,width,height);
            }
             return (flags & ALLBITS) == 0;
      }
}
```



```
C:\Windows\system32\cmd.exe - appletviewer  JavaExampleIObserverInApplet.java

F:\Java>javac JavaExampleIObserverInApplet.java

F:\Java>appletviewer JavaExampleIObserverInApplet.java
```

Double-Buffering

Double-buffering is the process of drawing graphics into an off-screen image buffer and then copying the contents of the buffer to the screen all at once. For complex graphics, using double-buffering can reduce flickering.

Swing automatically supports double-buffering for all of its components. To enable it, simply call the setDoubleBuffered() method (inherited from JComponent) to set the doubleBuffered property to true for any components that should use double-buffered drawing.

Remember that double-buffering is memory intensive. Its use is typically only justified for components that are repainted very frequently or have particularly complex graphics to display.

Note, however, that if a container uses double-buffering, any double-buffered children it has share the off-screen buffer of the container, so the required off-screen buffer is never larger than the on-screen size of the application.

Java MediaTracker

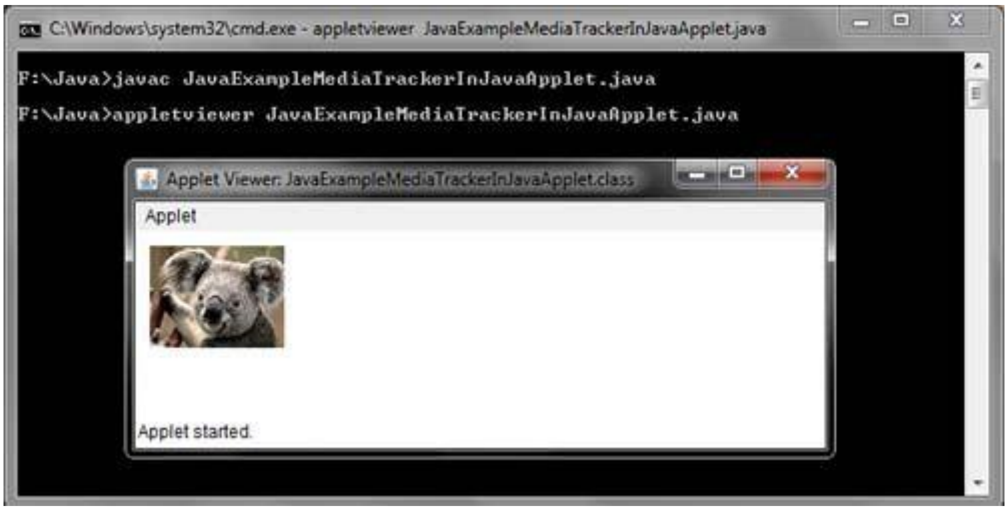
The java.awt.MediaTracker class is a useful class.It can monitor the progress of any number of images, so you can expect to all images are loaded or to be loaded only specific. Furthermore, can check for errors. The constructor is: public MediaTracker (Component comp) This constructor creates a new MediaTracker for pictures to be displayed in the variable comp, but really this parameter is not particularly important.

To upload images, a MediaTracker object provides the method following : public void addImage (Image image , int id)
This method adds the thumbnail image to the image list to load the object MediaTracker receiving the message . The id variable is used as the number of image identification when using other methods concerning this class.

Example:

```
import java.awt.*;
import java.applet.*;
public class JavaExampleMediaTrackerInJavaApplet extends Applet
{
    Image Img;
    public void init()
     {
        MediaTracker Trckr = new MediaTracker(this);
        Img = getImage(getDocumentBase(),"Koala.jpg");
        Trckr.addImage(Img,0);
        try
          {
            Trckr.waitForAll();
          }
           catch (InterruptedException e1) { }
     }
           public void paint(Graphics gr)
            {
              gr.drawImage(Img, 10, 10, this);
            }
}
/*<APPLET CODE=JavaExampleMediaTrackerInJavaApplet.class WIDTH=600 HEIGHT=150>
</APPLET>*/
```

Output:



```
F:\Java>javac JavaExampleMediaTrackerInJavaApplet.java

F:\Java>appletviewer JavaExampleMediaTrackerInJavaApplet.java
```

C:\Windows\system32\cmd.exe - appletviewer JavaExampleMediaTrackerInJavaApplet.java
Applet Viewer: JavaExampleMediaTrackerInJavaApplet.class
Applet
Applet started.

IMAGE PRODUCER

ImageProducer is an interface for objects that want to produce data for images. An object that implements the ImageProducer interface will supply integer or byte arrays that represent image data and produce Image objects. As you saw earlier, one form of the createImage( ) method takes an ImageProducer object as its argument. There are two image producers contained in java.awt.image: MemoryImageSource and FilteredImageSource. Here, we will examine MemoryImageSource and create a new Image object from data generated in anapplet.

MemoryImageSource

MemoryImageSource is a class that creates a new Image from an array of data. It defines several constructors. Here is the one we will be using:

MemoryImageSource(int width, int height, int pixel[ ], int offset, int scanLineWidth)

The MemoryImageSource object is constructed out of the array of integers specified by pixel, in the default RGB color model to produce data for an Image object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in width and height. The starting point in the pixel array to begin reading data is passed in offset. The width of a scan line (which is often the same as the width of the image) is passed in scanLineWidth.

The following short example generates a MemoryImageSource object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel)

```
/*
* <applet code="MemoryImageGenerator" width=256 height=256>
* </applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class MemoryImageGenerator extends Applet {
Image img;
public void init() {
Dimension d = getSize();
int w = d.width;
int h = d.height;
int pixels[] = new int[w * h];
int i = 0;
for(int y=0; y<h; y++) {
for(int x=0; x<w; x++) {
int r = (x^y)&0xff;
int g = (x*2^y*2)&0xff;
int b = (x*4^y*4)&0xff;
pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
}
}
img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
}
public void paint(Graphics g) {
g.drawImage(img, 0, 0, this);
}
}
```

The data for the new MemoryImageSource is created in the init( ) method. An array of integers is created to hold the pixel values; the data is generated in the nested for loops where the r, g, and b values get shifted into a pixel in the pixels array. Finally, createImage( ) is called with a new instance of a MemoryImageSource created from the raw pixel data as its parameter.

Image Consumer

Image consumers, on the other hand, are objects that consume **image** data. Once the **image** data is consumed, the object is then free to use (or modify) it. All **image consumers** implement the **ImageConsumer** interface.

Example:

```
import java.awt.image.ImageProducer;
import java.awt.image.ImageConsumer;

import java.util.List;

import java.util.ArrayList;

public abstract class AbstractImageSource implements ImageProducer {

    private List<ImageConsumer> consumers = new ArrayList<ImageConsumer>();

    protected int width;

    protected int height;

    protected int xOff;

    protected int yOff;

    public void addConsumer(final ImageConsumer pConsumer) {

        if (consumers.contains(pConsumer)) {

            return;

        }


        consumers.add(pConsumer);

        try {

            initConsumer(pConsumer);

            sendPixels(pConsumer);

            if (isConsumer(pConsumer)) {

                pConsumer.imageComplete(ImageConsumer.STATICIMAGEDONE);

                if (isConsumer(pConsumer)) {

                    pConsumer.imageComplete(ImageConsumer.IMAGEERROR);

                    removeConsumer(pConsumer);

                }

            }

        }

        catch (Exception e) {
            e.printStackTrace();


            if (isConsumer(pConsumer)) {
                pConsumer.imageComplete(ImageConsumer.IMAGEERROR);
            }
        }
    }


    public void removeConsumer(final ImageConsumer pConsumer) {
        consumers.remove(pConsumer);
    }
public void requestTopDownLeftRightResend(final ImageConsumer pConsumer) {
        // ignore
    }
    public void startProduction(final ImageConsumer pConsumer) {
        addConsumer(pConsumer);
    }
    public boolean isConsumer(final ImageConsumer pConsumer) {
        return consumers.contains(pConsumer);
    }
    protected abstract void initConsumer(ImageConsumer pConsumer);
    protected abstract void sendPixels(ImageConsumer pConsumer);
}
```

Image Filters

The Java image model also enables you to filter images easily. The concept of a filter is similar to the idea of a filter in photography. It is something that sits between the image consumer (the film) and the image producer (the outside world). The filter changes the image before it is delivered to the consumer.

The CropImageFilter is a pre-defined filter that crops an image to a certain dimension (it only shows a portion of the whole image). You create a CropImageFilter by passing the x, y, width, and height of the cropping rectangle to the constructor:

```
public CropImageFilter(int x, int y, int width, int height)
```

Once you have created an image filter, you can lay it on top of an existing image source by creating a FilteredImageSource:

```
public FilteredImageSource(ImageProducer imageSource, ImageFilter filter)
```

Example:

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
public class CropImage extends Applet
{
private Image originalImage;
private Image croppedImage;
private ImageFilter cropFilter;
public void init()
{
originalImage = getImage(getDocumentBase(), "samantha.gif");
cropFilter = new CropImageFilter(25, 30, 75, 75);
croppedImage = createImage(new FilteredImageSource(
originalImage.getSource(), cropFilter));
}
public void paint(Graphics g)
{
g.drawImage(originalImage, 0, 0, this);
g.drawImage(croppedImage, 0, 200, this);
}
```

What Is Animation?

Before getting into animation as it relates to Java, it's important to understand the basics of what animation is and how it works. So let's begin by asking the fundamental question: What is animation? Put simply, animation is the illusion of movement. Am I telling you that every animation you've ever seen is really just an illusion? That's exactly right! And probably the most surprising animated illusion is one that captured our attention long before modern computers-the television. When you watch television, you see lots of things moving around, but what you perceive as movement is really just a trick being played on your eyes.

New Term
Animation is the process of simulating movement.

In the case of television, the illusion of movement is created by displaying a rapid succession of images with slight changes in content. The human eye perceives these changes as movement because of its low visual acuity. I'll spare you the biology lesson of why this is so; the point is that our eyes are fairly easy to trick into falling for the illusion of animation. More specifically, the human eye can be tricked into perceiving animated movement with as low as 12 frames of movement per second. Animation speed is measured in frames per second (fps), which is the number of animation frames, or image changes, presented every second.

New Term
Frames per second (fps) is the number of animation frames, or image changes, presented every second.

Although 12fps is technically enough to fool our eyes into seeing animation, animations at speeds this low often end up looking somewhat jerky. Most professional animations therefore use a higher frame rate. Television, for example, uses 30fps. When you go to the movies, you see motion pictures at about 24fps. It's pretty apparent that these frame rates are more than enough to captivate our attention and successfully create the illusion of movement.

When programming animation in Java, you typically have the ability to manipulate the frame rate a decent amount. The most obvious limitation on frame rate is the speed at which the computer can generate and display the animation frames. In Java, this is a crucial point because Java applets aren't typically known to be speed demons. However, the recent release of just-in-time Java compilers has helped speed up Java applets, along with alleviating some of the performance concerns associated with animation.

Note
Currently, both Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0

support just-in-time compilation of Java applets.

Types of Animation

I know you're probably itching to see some real animation in Java, but there are a few more issues to cover before getting into the details of animation programming. More specifically, it's important for you to understand the primary types of animation used in Java programming. There are actually a lot of different types of animation, all of which are useful in different instances. However, for the purposes of implementing animation in Java, I've broken animation down into two basic types: frame-based animation and cast-based animation.

Frame-Based Animation

The most simple type of animation is *frame-based* animation, which is the primary type of animation found on the Web. Frame-based animation involves simulating movement by displaying a sequence of pregenerated, static frame images. A movie is a perfect example of frame-based animation; each frame of the film is a frame of animation, and when the frames are shown in rapid succession, they create the illusion of movement.

New Term
Frame-based animation simulates movement by displaying a sequence of pregenerated, static frame images.

Frame-based animation has no concept of a graphical object distinguishable from the background; everything appearing in a frame is part of that frame as a whole. The result is that each frame image contains all the information necessary for that frame in a static form. This is an important point because it distinguishes frame-based animation from cast-based animation, which you'll learn about next.

Note
Much of the animation used in Web sites is implemented using animated GIF images, which involves storing multiple animation frames in a single GIF image file. Animated GIFs are a very good example of frame-based animation.

Cast-Based Animation

A more powerful animation technique often employed in games and educational software is *cast-based* animation, which is also known as *sprite animation*. Cast-based animation involves graphical objects that move independently of a background. At this point, you may be a little confused by my usage of the term "graphical object" when referring to parts of an animation. In this case, a graphical object is something that logically can be thought of as a separate entity from the background of an animation image. For example, in an animation of the solar system, the planets would be separate graphical objects that are logically independent of the starry background.

New Term
Cast-based animation simulates movement using graphical objects that move independently of a background.

Each graphical object in a cast-based animation is referred to as a *sprite* and can have a position that varies over time. In other words, sprites have a velocity associated with them that determines how their position changes over time. Almost every computer game uses sprites to some degree. For example, every object in the classic Asteroids game is a sprite that moves independently of the black background.

New Term
A *sprite* is a graphical object that can move independently of a background or other objects.

Note
You may be wondering where the term *cast-based animation* comes from. It comes from the fact that sprites can be thought of as cast members moving around on a stage. This analogy of relating computer animation to theatrical performance is very useful. By thinking of sprites as cast members and the background as a stage, you can take the next logical step and think of an animation as a theatrical performance. In fact, this isn't far from the mark, because the goal of theatrical performances is to entertain the audience by telling a story through the interaction of the cast members. Likewise, cast-based animations use the interaction of sprites to entertain the user, while often telling a story or at least getting some point across.

Even though the fundamental principle behind sprite animation is the positional movement of a graphical object, there is no reason you can't incorporate frame-based animation into a sprite. Incorporating frame-based animation into a sprite allows you to change the image of the sprite as well as alter its position. This hybrid type of animation is what you will implement later today in the Java sprite classes.

I mentioned in the frame-based animation discussion that television is a good example of frame-based animation. But can you think of something on television that is created in a manner similar to cast-based animation (other than animated movies and cartoons)? Have you ever wondered how weatherpeople magically appear in front of a computer-generated map showing the weather? The news station uses a technique known as *blue-screening*, which enables them to overlay the weatherperson on top of the weather map in real time. It works like this: The person stands in front of a blue backdrop, which serves as a transparent background. The image of the weatherperson is overlaid onto the weather map; the trick is that the blue background is filtered out when the image is overlaid so that it is effectively transparent. In this way, the weatherperson is acting exactly like a sprite!

Transparency

The weatherperson example brings up a very important point regarding sprites: transparency. Because bitmapped images are rectangular by nature, a problem arises when sprite images aren't rectangular in shape. In sprites that aren't rectangular in shape, which is the majority of sprites, the pixels surrounding the sprite image are unused. In a graphics system without transparency, these unused pixels are drawn just like any others. The end result is sprites that have visible rectangular borders around them, which completely destroys the effectiveness of having sprites overlaid on a background image.

What's the solution? Well, one solution is to make all your sprites rectangular. Unless you're planning to write an applet showing dancing boxes, a more realistic solution is transparency, which allows you to define a certain color in an image as unused, or transparent. When pixels of this color are encountered by graphics drawing routines, they are simply skipped, leaving the original background intact. Transparent colors in images act exactly like the weatherperson's blue screen.

New Term
Transparent colors are colors in an image that are unused, meaning that they aren't drawn when the rest of the colors in the image are drawn.

You're probably thinking that implementing transparency involves a lot of low-level bit twiddling and image pixel manipulation. In some programming environments you would be correct in this assumption, but not in Java. Fortunately, transparency is already supported in Java by way of the GIF 89a image format. In the GIF 89a image format, you simply specify a color of the GIF image that serves as the transparent color. When the image is drawn, pixels matching the transparent color are skipped and left undrawn, leaving the background pixels unchanged. No more dancing boxes!

Z-Order

In many instances, you will want some sprites to appear on top of others. For example, in the solar system animation you would want to be able to see some planets passing in front of others. You handle this problem by assigning each planet sprite a screen depth, which is also referred to as *Z-order*.

New Term
Z-order is the relative depth of sprites on the screen.

The depth of sprites is called *Z-order* because it works sort of like another dimension-like a Z axis. You can think of sprites moving around on the screen in the XY plane. Similarly, the Z axis can be thought of as another axis projected into the screen that determines how the sprites overlap each other. To put it another way, Z-order determines a sprite's depth within the screen. By making use of a Z axis, you might think that Z-ordered sprites are 3D. The truth is that Z-ordered sprites aren't 3D because the Z axis is a hypothetical axis that is used only to determine how sprite objects hide each other. A real 3D sprite would be able to move just as freely in the Z axis as it does in the XY plane.

Just to make sure that you get a clear picture of how Z-order works, let's go back for a moment to the good old days of traditional animation. Traditional animators, such as those at Disney, used celluloid sheets to draw animated objects. They drew on these because they could be overlaid on a background image and moved independently. This was known as *cel animation* and should sound vaguely familiar. (Cel animation is an early version of sprite animation.) Each cel sheet corresponds to a unique Z-order value, determined by where in the pile of sheets the sheet is located. If an image near the top of the pile happens to be in the same location on the cel sheet as any lower images, it conceals them. The location of each image in the stack of cel sheets is its Z-order, which determines its visibility precedence. The same thing applies to sprites in cast-based animations, except that the Z-order is determined by the order in which the sprites are drawn, rather than the cel sheet location. This concept of a pile of cel sheets representing all the sprites in a sprite system will be useful later today when you develop the sprite classes.

Collision Detection

Although collision detection is primarily useful only in games, it is an important component of sprite animation. *Collision detection* is the process of determining whether sprites have collided with each other. Although collision detection doesn't directly play a role in creating the illusion of movement, it is tightly linked to sprite animation and extremely useful in some scenarios, such as games.

New Term
Collision detection is the process of determining if sprites have collided with each other.

Cell Animation

Now that we have presented an overview of the image APls, WI! .in put together an interesting applet that will display a sequence of animation cels, .he animation cells are taken from a single image that can arrange the cells in a grid specified via the rows and cols cparam tags.

Each cell in the image is snipped out in a way similar to that , used in the Tile Image example earlier, W" obtain the sequence in which to display the cells from the sequence <param> tag, This is a comma-separated list (If cell numbers that is zero-based and proceeds across the grid from left to right I, top to bottom.

Once the' applet has parsed the <param> tags and loaded the source image, it cuts the image into a number of small subimages. Then, a thread is started that Causes the images to be displayed according to the order described in sequence.

Example:

```
import java.awt.*;
import java.applet.*;
public class AnimationExample extends Applet {
  Image picture;

  public void init() {
    picture =getImage(getDocumentBase(),"bike_1.gif");
  }

  public void paint(Graphics g) {
    for(int i=0;i<500;i++){
      g.drawImage(picture, i,30, this);

      try{Thread.sleep(100);}catch(Exception e){}
    }
  }
}
```

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

Swing

What is Swing?

Swing is the collection of user interface components for the Java programs. It is part of Java foundation classes that are referred to as JFC. In simple words, Swing is the graphical user interface toolkit that is used for developing the windows based java applications or programs. Swing is the successor of AWT which is known as Abstract window toolkit API for Java and AWT components are a mainly heavyweight.

The components are lightweight as compared to AWT components. It provides a good interface to the user for all the platforms. It is not specifically for one platform. The components are written in Java and platform independent as well. The Java foundation classes were first appeared in 1997 and then later on it is termed as Swing. To use the swing in java, javax. swing package needs to be used or import. It is also known as Java Swing

Features of the Swing

1. Platform Independent: It is platform independent, the swing components that are used to build the program are not platform specific. It can be used at any platform and anywhere.

2. Lightweight: Swing components are lightweight which helps in creating the UI lighter. Swings component allows it to plug into the operating system user interface framework that includes the mappings for screens or device and other user interactions like key press and mouse movements.

3. Plugging: It has a powerful component that can be extended to provide the support for the user interface that helps in good look and feel to the application. It refers to the highly modular-based architecture that allows it to plug into other customized implementations and framework for user interfaces. Its components are imported through a package called java.swing.

4. Manageable: It is easy to manage and configure. Its mechanism and composition pattern allows changing the settings at run time as well. The uniform changes can be provided to the user interface without doing any changes to application code.

5. MVC: They mainly follows the concept of MVC that is Model View Controller. With the help of this, we can do the changes in one component without impacting or touching other components. It is known as loosely coupled architecture as well.

6. Customizable: Swing controls can be easily customized. It can be changed and the visual appearance of the swing component application is independent of its internal representation.

The MVC Connection

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

In the MVC terminology we have ,

Model:

the Model corresponds to the state information which is associated with a component. For example-in case of a check box, the model contains a field which indicates whether the box is checked or unchecked.

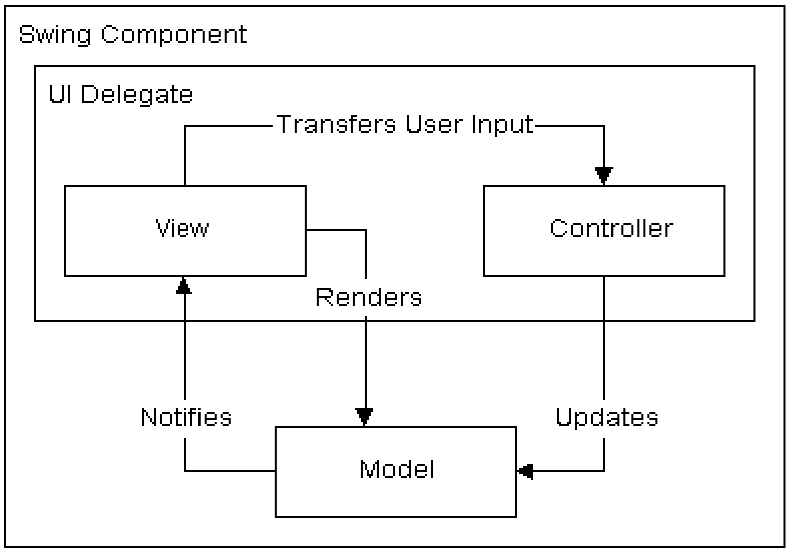
View:

The view determines how a component has displayed on the screen, including any aspects of view that are affected by the current state of the model.

Controller:

The controller determines how the component will react to the user.

For example, when user clicks a check box, the controller will reacts by changing model to reflect the user's choice (checked or unchecked) which then results in the view being updated.



Components and Containers

Components
JComponent class is a base class for all the components in a swing.

The frequently used components include,
JButton
JTextField
JTextArea
JRadioButton
JComboBox etc.
All these components should be added to the container if not, it will not appear on the application.

Example

To create the button instance,
JButton clickButton=new JButton();

To add the button to the container,
myFrame.add();

Containers are an integral part of SWING GUI components. A container provides a space where a component can be located. A Container in AWT is a component itself and it provides the capability to add a component to itself. Following are certain noticable points to be considered.

Sub classes of Container are called as Container. For example, JPanel, JFrame and JWindow.

Container can add only a Component to itself.

A default layout is present in each container which can be overridden using **setLayout** method.

SWING Containers

Following is the list of commonly used containers while designed GUI using SWING.

Sr.No.	**Container & Description**
1	Panel JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.
2	Frame A JFrame is a top-level window with a title and a border.

3	Window A JWindow object is a top-level window with no borders and no menubar.

Example:

```
import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class ContainerDemo {

public static void main(String[] args) {

JFrame baseFrame =new JFrame();

baseFrame.setTitle("Base Container");

JPanel contentPane=new JPanel();

contentPane.setBackground(Color.pink);

baseFrame.setSize(400, 400);

baseFrame.add(contentPane);

baseFrame.setDefaultCloseOperation(baseFrame.EXIT_ON_CLOSE);

baseFrame.setVisible(true);

}

}
```

Swing Packages

Unlike AWT, **Java Swing** provides platform-independent and lightweight components. The javax. **swing package** provides classes for **java swing** API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Some of the packages of swing components that are used most are the following:

- **Javax.swing**
- **javax.swing.event**

• javax.swing.plaf.basic

• javax.swing.table

• javax.swing.border

The largest of the swing packages, *javax.swing*, contains most of the user-interface classes

There are totally 16 packages in the swings packages and javax.swing is one of them. A brief description of all the packages in swing is given below.

Packages	**Description**
javax.swing	Provides a set of "lightweight" (all-Java language) components to the maximum degree possible, work the same on all platforms.
javax.swing.border	Provides classes and interface for drawing specialized borders around a Swing component.
javax.swi ng.colorchooser	Contains classes and interfaces used by the JcolorChooser component.
javax.swing.event	Provides for events fired by Swing components
javax.swing.filechooser	Contains classes and interfaces used by the JfileChooser component.
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities.
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look and feel.
javax.swing.plaf.metal	Provides user interface objects built according to the Java look and feel (once condenamed Metal), which is the default look and feel.
javax.swi ng.plaf.mult	Provides user interface objects that combine two or more look and feels.
javax.swing.table	Provides classes and interfaces for dealing with javax.swing.jtable
javax.swing.text	Provides classes and interfaces that deal with editable and noneditable text components

javax.swing.text.html	Provides the class HTML Editor Kit and supporting classes for creating HTML text editors.
javax.swing.text.html.parser	Provides the default HTML parser, along with support classes.
javax.swing.text.rtf	Provides a class RTF Editor Kit for creating Rich-Text-Format text editors.
javax.swing.tree	Provides classes and interfaces for dealing with javax.swing.jtree

Event Handling in Java swing

Event:

An event is a signal to the program that something has happened. It can be triggered by typing in a text field, selecting an item from the menu etc. The action is initiated outside the scope of the program and it is handled by a piece of code inside the program. Events may also be triggered when timer expires, hardware or software failure occurs, operation completes, counter is increased or decreased by a value etc.

Event handler: The code that performs a task in response to an event. is called event handler.

• **Event handling:** It is process of responding to events that can occur at any time during execution of a program.

• **Event Source:** It is an object that generates the event(s). Usually the event source is a button or the other component that the user can click but any Swing component can be an event source. The job of the event source is to accept registrations, get events from the user and call the listener's event handling method.

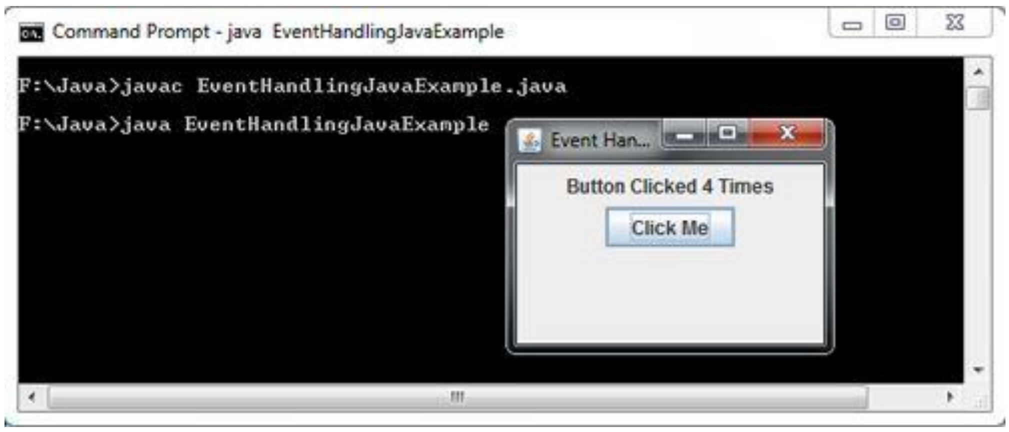
• **Event Listener:** It is an object that watch for (i.e. listen for) events and handles them when they occur. It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the eventhandling.

• **Listener interface**: It is an interface which contains methods that the listener must implement and the source of the event invokes when the event occurs.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class EventExample extends JFrame implements ActionListener
{
   private int count =0;
   JLabel lblData;
   EventExample()
   {
    setLayout(new FlowLayout());
    lblData = new JLabel("Button Clicked 0 Times");
    JButton btnClick=new JButton("Click Me");
    btnClick.addActionListener(this);
    add(lblData);
    add(btnClick);
  }
  public void actionPerformed(ActionEvent e)
  {
    count++;
    lblData.setText("Button Clicked " + count +" Times");
  }
}
 class EventHandlingJavaExample
{
  public static void main(String args[])
  {
     EventExample frame = new EventExample();
     frame.setTitle("Event Handling Java Example");
     frame.setBounds(200,150,180,150);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setVisible(true);
  }
}
```

Output:



```
F:\Java>javac EventHandlingJavaExample.java

F:\Java>java EventHandlingJavaExample
```

Creating a swing

Swing in java is part of Java foundation class which is lightweight and platform independent. It is used for creating window based applications. It includes components like button, scroll bar, text field etc. Putting together all these components makes a graphical user interface. In this article, we will go through the concepts involved in the process of building applications using swing in Java.

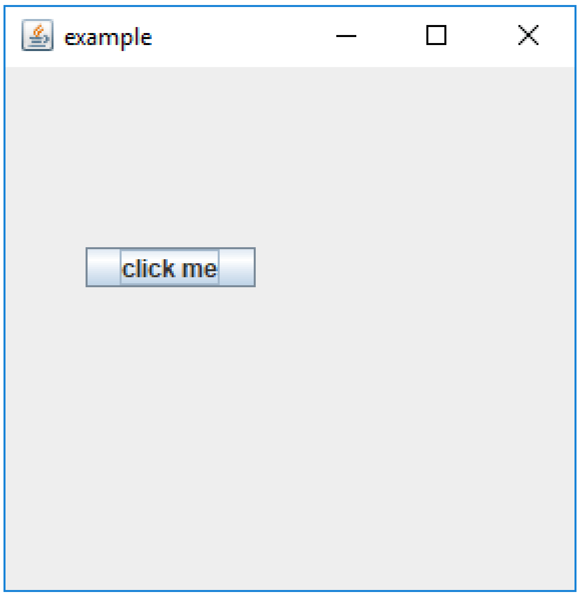
Swing in Java is a lightweight GUI toolkit which has a wide variety of widgets for building optimized window based applications. It is a part of the JFC(Java Foundation Classes). It is build on top of the AWT API and entirely written in java. It is platform independent unlike AWT and has lightweight components.

It becomes easier to build applications since we already have GUI components like button, checkbox etc. This is helpful because we do not have to start from the scratch.

Example:

```
import javax.swing.*;

public class example{

public static void main(String args[]) {

JFrame a = new JFrame("example");

JButton b = new JButton("click me");

b.setBounds(40,90,85,20);

a.add(b);

a.setSize(300,300);

a.setLayout(null);

a.setVisible(true);

}

}
```

Output:

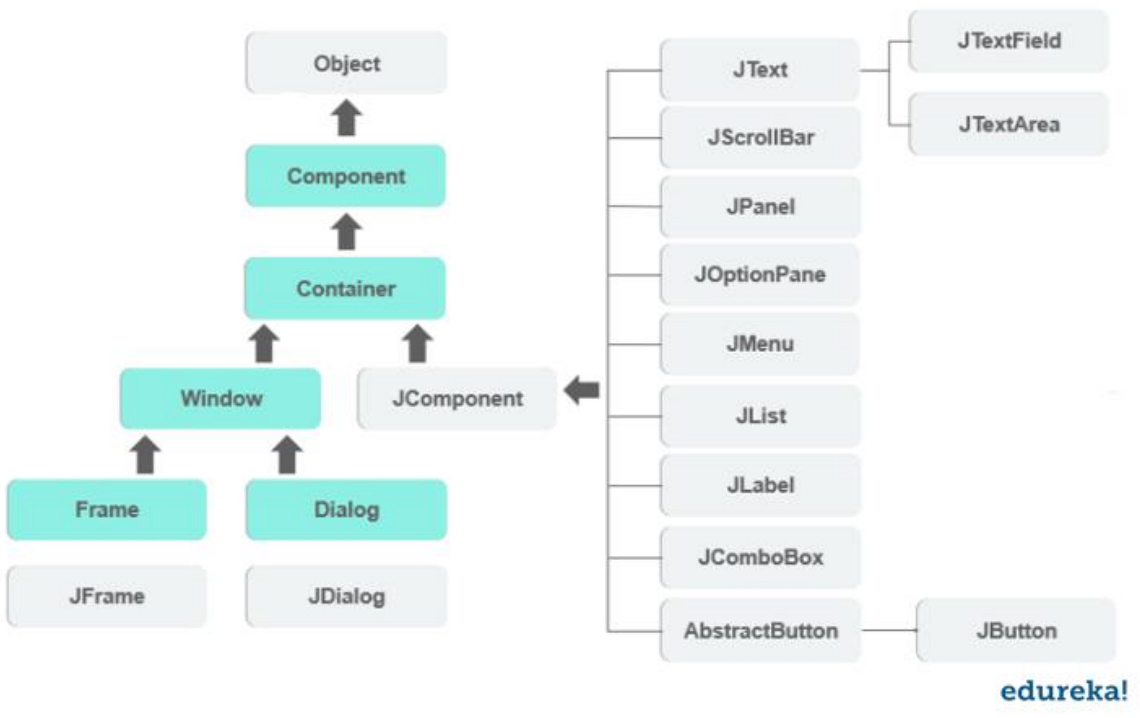


Exploring swing

Difference Between AWT and Swing

AWT	SWING
Platform Dependent	Platform Independent
Does not follow MVC	Follows MVC
Lesser Components	More powerful components
Does not support pluggable look and feel	Supports pluggable look and feel
Heavyweight	Lightweight

Java Swing Class Hierarchy



All the components in swing like JButton, JComboBox, JList, JLabel are inherited from the JComponent class which can be added to the container classes. Containers are the windows like frame and dialog boxes. Basic swing components are the building blocks of any gui application. Methods like setLayout override the default layout in each container. Containers like JFrame and JDialog can only add a component to itself.

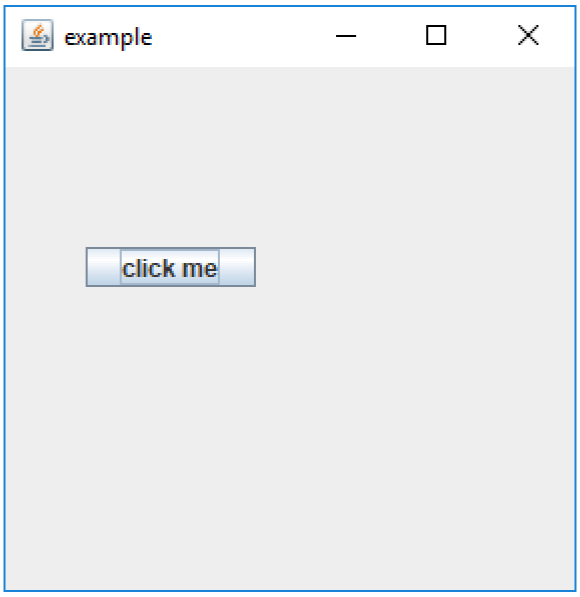
JButton Class

It is used to create a labelled button. Using the ActionListener it will result in some action when the button is pushed. It inherits the AbstractButton class and is platform independent.

Example:

```
import javax.swing.*;

public class example{

public static void main(String args[]) {

JFrame a = new JFrame("example");

JButton b = new JButton("click me");

b.setBounds(40,90,85,20);

a.add(b);

a.setSize(300,300);

a.setLayout(null);

a.setVisible(true);

}

}
```

Output



Java JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

JDBC-ODBC Bridge Driver,

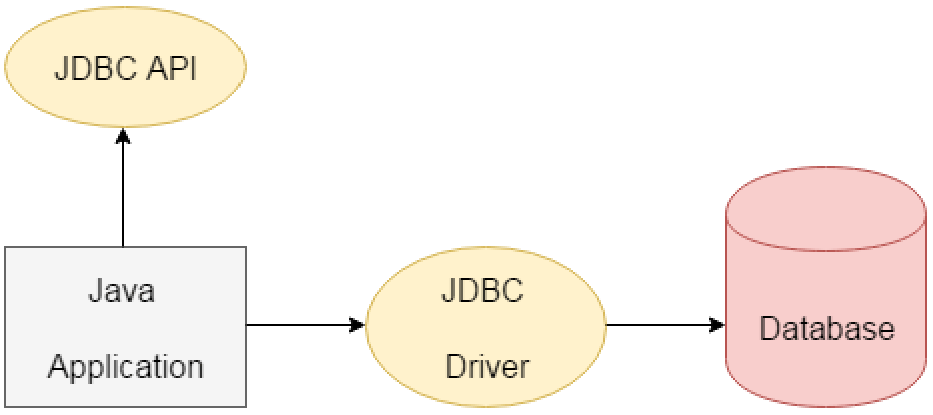
Native Driver,

Network Protocol Driver, and

Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

Driver interface

Connection interface

Statement interface

PreparedStatement interface

CallableStatement interface

ResultSet interface

ResultSetMetaData interface

DatabaseMetaData interface

RowSet interface

A list of popular *classes* of JDBC API are given below:

DriverManager class

Blob class

Clob class

Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

Connect to the database

Execute queries and update statements to the database

Retrieve the result received from the database.

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

Connecting to a relational database with Java

Requirements

Flexmonster Pivot version 2.213 or higher

Java 7 or higher

A driver for the database

Supported databases

Oracle Database – driver

MySQL – driver

Microsoft SQL Server – driver

PostgreSQL – driver

Other JDBC databases

Step 1: Embed the component into your web page

Set up an empty component in your HTML page. If Flexmonster is not yet embedded – return to Quick start. Your code should look similar to the following example:

```
<div id="pivotContainer">The component will appear here</div>

<script src="flexmonster/flexmonster.js"></script>

<script>

        var pivot = new Flexmonster({

                container: "pivotContainer",

                toolbar: true,

                licenseKey: "XXXX-XXXX-XXXX-XXXX-XXXX"

        });

</script>
```

Step 2: Setup the Data Compressor on the server

Add the following dependencies to your project:

An appropriate database driver

flexmonster-compressor.jar – located in the Pivot Table for Databases/server/java/ folder of the download package.

Below is a connection and query sample for a MySQL database:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();

String connectionString = "jdbc:mysql://localhost:3306/foodmart";

Connection connection =

        DriverManager.getConnection(connectionString, "user", "pass");

String query = "SELECT * FROM customer";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);
```

Then the following line of code will convert ResultSet to an InputStream:

```
InputStream inputStream = Compressor.compressDb(resultSet);
```

Now you can create a response from the InputStream. For simple cases, it is suitable to read all

content at once:

```
Scanner s = new Scanner(inputStream).useDelimiter("\\A");

String output = s.hasNext() ? s.next() : "";
```

It is also possible to create a streaming response. This means that the end user will get the response with minimal delay and the server will use less memory. This is the recommended approach for large datasets:

```
response.setContentType("text/plain");

OutputStream outputStream = response.getOutputStream();

int length = 0;

byte[] buffer = new byte[10240];

while ((length = inputStream.read(buffer)) > 0) {

   outputStream.write(buffer, 0, length);

   outputStream.flush();

}
```

The full project is available at Pivot Table for Databases/server/java/ inside the download package.

Step 3: Enable cross-origin resource sharing (CORS)

By default, the browser prevents JavaScript from making requests across domain boundaries. CORS allows web applications to make cross-domain requests. Here are some useful links explaining how to setup CORS on your server:

Tomcat – Configuration Reference (CORS Filter)

Jetty – Jetty/Feature/Cross Origin Filter

JBOSS – CORS Filter Installation

Step 4: Configure the report with your own data

Now it's time to configure the pivot table on the web page. Let's create a minimal report for this (replace filename and other parameters with your specific values):

```
var pivot = new Flexmonster({

        container: "pivotContainer",

        toolbar: true,

        report: {
                dataSource: {
                        type: "csv",
                        /* URL to the Data Compressor Java */
                        filename: "http://localhost:8400/FlexmonsterCompressor/get"
                }
        },
        licenseKey: "XXXX-XXXX-XXXX-XXXX-XXXX"
});
```

Launch the web page from a browser — there you go! A pivot table is embedded into your project. Check out the example on JSFiddle.

Java Compressor API

Processing for streams:

InputStream compressStream(InputStream inputStream, char delimiter)

InputStream compressStream(InputStream inputStream)

Processing for Databases:

InputStream compressDb(ResultSet resultSet)

Examples

Saving to a file

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
String connectionString = "jdbc:mysql://localhost:3306/foodmart";
Connection connection =
        DriverManager.getConnection(connectionString, "user", "pass");
String query = "SELECT * FROM customer";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
InputStream inputStream = Compressor.compressDb(resultSet);
```

```
OutputStream outputStream = new FileOutputStream("data.csv");

int length = 0;

byte[] buffer = new byte[10240];

while ((length = inputStream.read(buffer)) > 0) {

   outputStream.write(buffer, 0, length);

}

inputStream.close();

outputStream.close();
```

Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

Driver class: The driver class for the mysql database is **com.mysql.jdbc.Driver**.

Connection URL: The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.

Username: The default username for the mysql database is **root**.

Password: It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Create table:

```
create database sonoo;
use sonoo;
create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
```

```
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Unit-IV

Life Cycle of a Servlet

The entire life cycle of a Servlet is managed by the **Servlet container** which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it. So, before creating a Servlet object let's first understand the life cycle of the Servlet object which is actually understanding that how the Servlet container manages the Servlet object.

Stages of the Servlet Life Cycle: The Servlet life cycle mainly goes through Five stages,

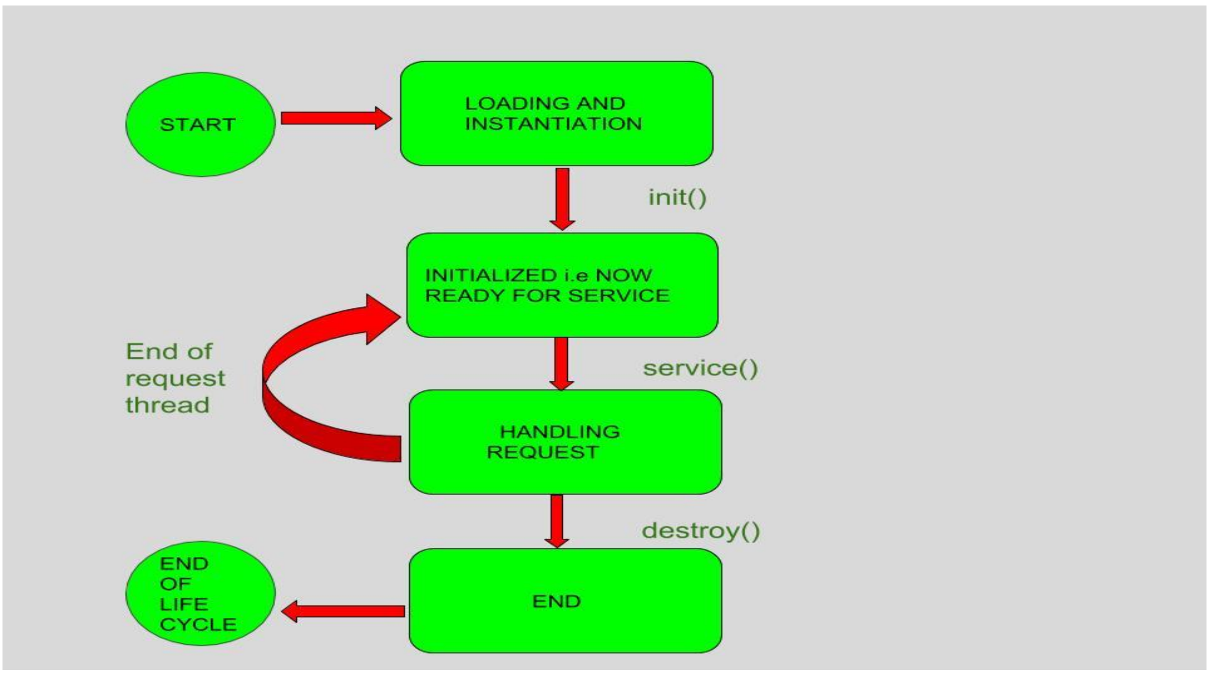
Servlet class is loaded.

Servlet instance is created.

init method is invoked.

service method is invoked.

destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is
is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

public void init(ServletConfig config) **throws** ServletException

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

public void destroy()

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

Simple Servlet

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context.

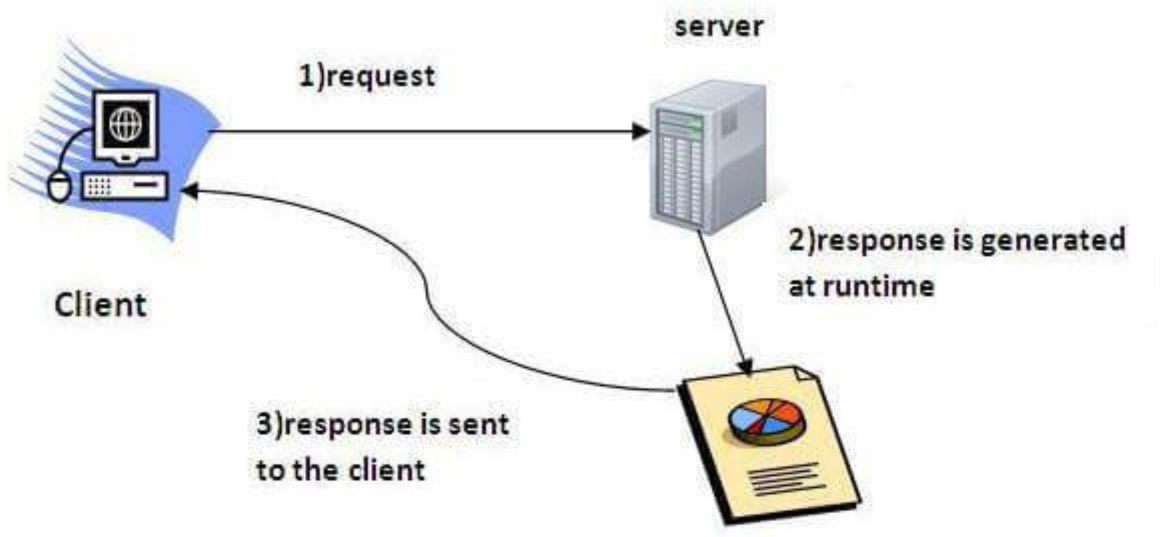
Servlet is a technology which is used to create a web application.

Servlet is an API that provides many interfaces and classes including documentation.

Servlet is an interface that must be implemented for creating any Servlet.

Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.

Servlet is a web component that is deployed on the server to create a dynamic web page.

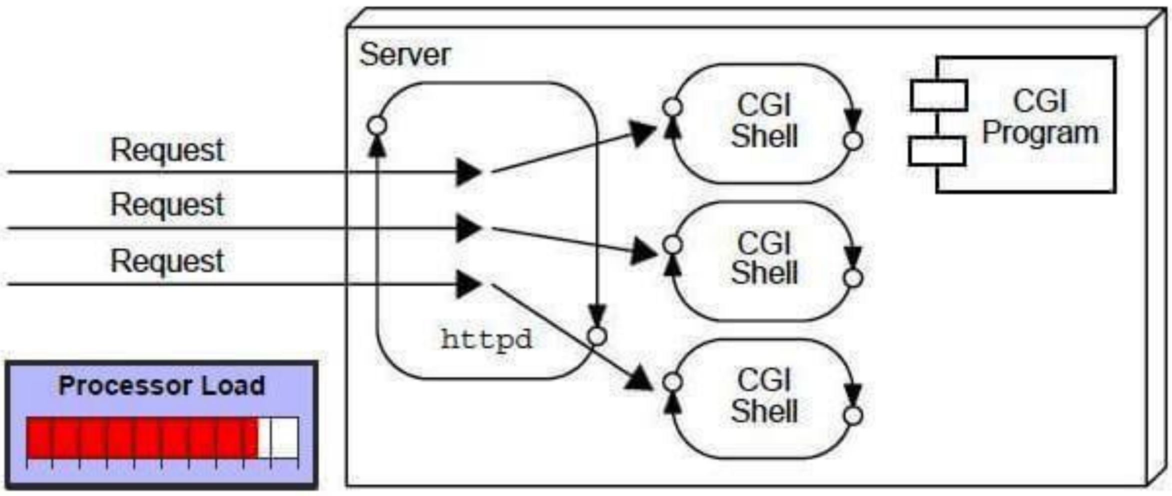

server
1)request
2)response is generated at runtime
Client
3)response is sent to the client

What is a web application?

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter, etc. and other elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request.

CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

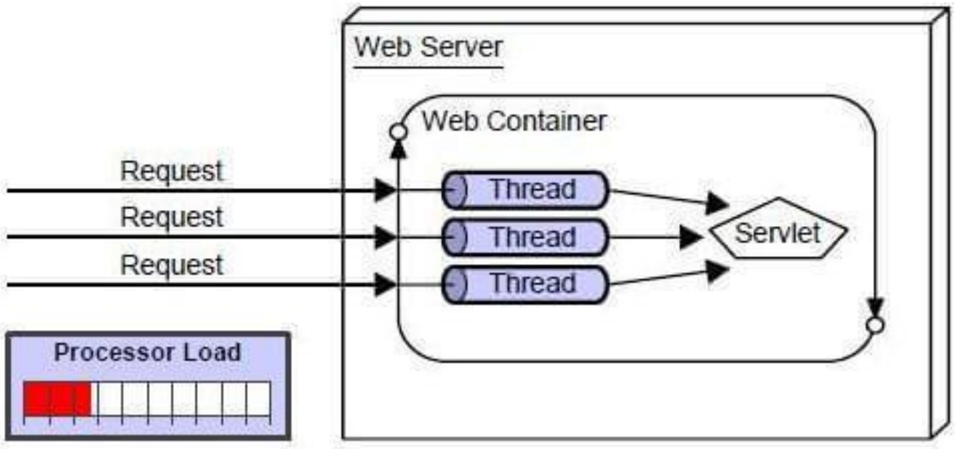


Disadvantages of CGI

There are many problems in CGI technology:

If the number of clients increases, it takes more time for sending the response.
For each request, it starts a process, and the web server is limited to start processes.
It uses platform dependent language e.g. C, C++, perl.

Advantages of Servlet



There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

Better performance: because it creates a thread for each request, not process.
Portability: because it uses Java language.
Robust: JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection,

etc.

Secure: because it uses java language.

Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

Servlet
ServletRequest
ServletResponse
RequestDispatcher
ServletConfig
ServletContext
SingleThreadModel
Filter
FilterConfig
FilterChain
ServletRequestListener
ServletRequestAttributeListener
ServletContextListener
ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

GenericServlet
ServletInputStream
ServletOutputStream
ServletRequestWrapper
ServletResponseWrapper
ServletRequestEvent

ServletContextEvent
ServletRequestAttributeEvent
ServletContextAttributeEvent
ServletException
UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

HttpServletRequest
HttpServletResponse
HttpSession
HttpSessionListener
HttpSessionAttributeListener
HttpSessionBindingListener
HttpSessionActivationListener
HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

HttpServlet
Cookie
HttpServletRequestWrapper
HttpServletResponseWrapper
HttpSessionEvent
HttpSessionBindingEvent
HttpUtils (deprecated now)

javax.servlet package

This package contains various servlet interfaces and classes which are capable of handling any type of protocol.

javax.servlet package interface

Some of the important interfaces are listed below.

Interface	**Overview**
Servlet	This interface is used to create a servlet class. Each servlet class must require to implement this interface either directly or indirectly.

ServletRequest	The object of this interface is used to retrieve the information from the user.
ServletResponse	The object of this interface is used to provide response to the user.
ServletConfig	ServletConfig object is used to provide the information to the servlet class explicitly.
ServletContext	The object of ServletContext is used to provide the information to the web application explicitly.

javax.servlet package classes
Some of the important classes are listed below.

Classes	Overview
GenericServlet	This is used to create servlet class. Internally, it implements the Servlet interface.
ServletInputStream	This class is used to read the binary data from user requests.
ServletOutputStream	This class is used to send binary data to the user side.
ServletException	This class is used to handle the exceptions occur in servlets.
ServletContextEvent	If any changes are made in servlet context of web application, this class notifies.

javax.servlet.http package

This package contains various interfaces and classes which are capable of handling a specific http type of protocol.
javax.servlet.http package interface
Some of the important interface of this package are listed below:

Interface	Overview
HttpServletRequest	The object of this interface is used to get the information from the user under http protocol.
HttpServletResponse	The object of this interface is used to provide the response of the request under http protocol.
HttpSession	This interface is used to track the information of users.
HttpSessionAttributeListener	This interface notifies if any change occurs in HttpSession attribute.

HttpSessionListener	This interface notifies if any changes occur in HttpSession lifecycle.

javax.servlet.http package classes

Some of the important interface of this package are listed below.

Class	Overview
HttpServlet	This class is used to create servlet class.
Cookie	This class is used to maintain the session of the state.
HttpSessionEvent	This class notifies if any changes occur in the session of web application.
HttpSessionBindingEvent	This class notifies when any attribute is bound, unbound or replaced in a session.

HTTP Requests and responses

The request sent by the computer to a web server, contains all sorts of potentially interesting information; it is known as HTTP requests.

The HTTP client sends the request to the server in the form of request message which includes following information:

The Request-line

The analysis of source IP address, proxy and port

The analysis of destination IP address, protocol, port and host

The Requested URI (Uniform Resource Identifier)

The Request method and Content

The User-Agent header

The Connection control header

The Cache control header

The HTTP request method indicates the method to be performed on the resource identified by the **Requested URI (Uniform Resource Identifier)**. This method is case-sensitive and should be used in uppercase.

The HTTP request methods are:

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

HTTP Response

HTTP Response sent by a server to the client. The response is used to provide the client with the resource it requested. It is also used to inform the client that the action requested has been carried out. It can also inform the client that an error occurred in processing its request.

An HTTP response contains the following things:

Status Line

Response Header Fields or a series of HTTP headers

Message Body

In the request message, each HTTP header is followed by a carriage returns line feed (CRLF). After the last of the HTTP headers, an additional CRLF is used and then begins the message body.

Status Line

In the response message, the status line is the first line. The status line contains three items:

a) HTTP Version Number

It is used to show the HTTP specification to which the server has tried to make the message comply.

Example

HTTP-Version = HTTP/1.1

b) Status Code

It is a three-digit number that indicates the result of the request. The first digit defines the class of the response. The last two digits do not have any categorization role. There are five values for the first digit, which are as follows:

Code and Description

1xx: Information

It shows that the request was received and continuing the process.

2xx: Success

It shows that the action was received successfully, understood, and accepted.

3xx: Redirection

It shows that further action must be taken to complete the request.

4xx: Client Error

It shows that the request contains incorrect syntax, or it cannot be fulfilled.

5xx: Server Error

It shows that the server failed to fulfil a valid request.

c) Reason Phrase

It is also known as the status text. It is a human-readable text that summarizes the meaning of the status code.

An example of the response line is as follows:

HTTP/1.1 200 OK

Here,

HTTP/1.1 is the HTTP version.
200 is the status code.

OK is the reason phrase.

Response Header Fields

The HTTP Headers for the response of the server contain the information that a client can use to find out more about the response, and about the server that sent it. This information is used to assist the client with displaying the response to a user, with storing the response for the use of future, and with making further requests to the server now or in the future.

```
response-header = Accept-Ranges
               | Age
               | ETag
               | Location
               | Proxy-Authenticate
               | Retry-After
               | Server
               | Vary
               | WWW-Authenticate
```

The name of the Response-header field can be extended reliably only in combination with a change in the version of the protocol.

Message Body

The response's message body may be referred to for convenience as a response body.

The body of the message is used for most responses. The exceptions are where a server is using certain status codes and where the server is responding to a client request, which asks for the headers but not the response body.

For a response to a successful request, the body of the message contains either some information about the status of the action which is requested by the client or the resource which is requested by the client. For the response to an unsuccessful request, the body of the message might provide further information about some action the client needs to take to complete the request successfully or about the reason for the error

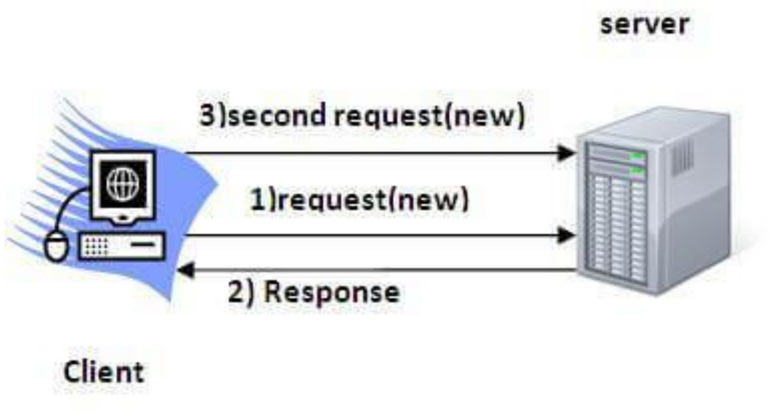
Session Tracking in Servlets

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

Cookies
Hidden Form Field
URL Rewriting
HttpSession

Cookies in Servlet

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.

Types of Cookie

There are 2 types of cookies in servlets.

Non-persistent cookie

Persistent cookie

Non-persistent cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

Simplest technique of maintaining the state.

Cookies are maintained at client side.

Disadvantage of Cookies

It will not work if cookie is disabled from the browser.

Only textual information can be set in Cookie object.

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

Useful Methods of Cookie class

There are given some commonly used methods of the Cookie class.

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.

public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfa

public void addCookie(Cookie ck):method of HttpServletResponse interface is used to add cookie in re
public Cookie[] getCookies():method of HttpServletRequest interface is used to return all the cookies fr

How to create Cookie?

Let's see the simple code to create cookie.

```
Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object
response.addCookie(ck);//adding cookie in the response
```

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

```
Cookie ck=new Cookie("user","");//deleting value of cookie
ck.setMaxAge(0);//changing the maximum age to 0 seconds
response.addCookie(ck);//adding cookie in the response
```

How to get Cookies?

Let's see the simple code to get all the cookies.

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++){
 out.print("<br>"+ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie
}
```

Java Server Pages (JSP) Overview

JSP stands for JAVA SERVER PAGES. It is a standard java extension used to simplify the creation and management of dynamic web pages. JSP's allow us to separate the dynamic content of webpage from static presentation content. A JSP page consists of HTML tags and JSP tags. HTML tags are used to create static page content and JSP tags are used to add dynamic content to web pages. JSP pages are compiled into a Java Servlet by a JSP translator.

Why Use JSP?

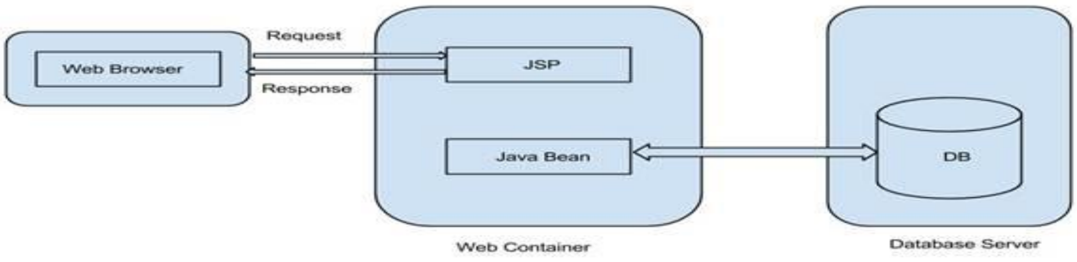
Building dynamic web pages by using JSP offers the following benefits:

Web pages created using JSP are portable and can be used across multiple platforms without making any changes.
Programming in JSP in easier than in Servlets.
JSP pages are automatically compiled by servers. Developers need to compile JSP pages when the source code of the JSP page is changed.
Most important benefit of JSP is the separation of business logic from presentation logic. JavaBeans contain business logic and JSP pages contain presentation logic. This separation makes an application created in JSP more secure and reusable.

Advantages of JSP

JSP pages easily combines static templates like HTML or XML fragments.
Before execution, JSP pages are compiled into servlets so we can easily update presentation code.
If JSP page is modified, then there is no need to recompile and redeploy the project. We only need to recompile and redeploy the servlet code.
It is easy to separate business logic from presentation logic. So JSP can be easily managed.
It is easy to learn and easy to implement.
JSP environment provides compilation of pages automatically.
Servlets cannot be accessed directly whereas a JSP page can be accessed directly as a simple HTML page.

JSP Architecture



As shown in above architecture, web browser directly accesses the JSP page of web container. The JSP pages interact with the web container's JavaBeans which represent the application model. When client sends request from a JSP page, the response is sent back to the client depending on the requests which are invoked by the client request. If response requires accessing the database, the JSP page uses JavaBeans to get data from the database.

Life Cycle of JSP

JSP pages follow these phases:

Translate the JSP to servlet code.
Compile the servlet to byte code.
Load the servlet class.
Create the servlet instance.
Call the jspInit method.
Call the _jspService method.
Call the jspDestroy method.

JSP Page Translation: In the first step, the web container translates the JSP file into a Java source file that contains a servlet class definition. The web container validates the correctness of JSP pages and tag files.

JSP Page Compilation: In the second step, the web container compiles the servlet source code into a Java class file.

JSP Page Class Loading: In the third step, the servlet class byte code is loaded into the web container's JVM software using class loader.

JSP Page Servlet Instance: In the fourth step, the web container creates an instance of the servlet class.

JSP Page Initialization: In the fifth step, the web container initializes the servlet by calling the jspInit()

method. This method is called immediately after the instance gets created. It is called only once during JSP life cycle.

To perform initialization we need to override jspInit() method:

```
public void jspInit (){
//code
}
```

JSP Page Service: The initialized servlet can now service the requests. The web container calls the _jspService () method to process the user request. This method is called for every request during its life cycle.

This method takes HttpServletRequest and HttpServletResponse parameters.

JSP Page Destroyed: When the web container removes the JSP servlet instance from service, it calls jspDestroy () method to allow JSP page to clean up the resources. This is end of the JSP life cycle.

This method can be written as follows:

```
public void jspDestroy()
{
//code
}
```

JSP Syntax

This topic defines about basic syntax in Java Server Page such as Scriptlet, Declarative, Expression and Comments.

Syntax Name	**Code**
Scriptlet Tag	<%statement or java code%>
Declarative tag	<%! Statement %>
Expression tag	<% = Statement %>
Comment tag	<%-----comments------ %>

The Scriptlet Tag: it is used to include java Technology Code in the JSP. The syntax is as follows:

```
<% java code %>
```

Listing 1. Example using scriptlet tag

```
<% int i=0; %>
```

The Declarative Tag: It is used to include members in the JSP servlet class either attributes or methods. The syntax is as follows:

```
<!  Statement %>
```

Listing 2. Example using declarative tag

```
<! Int counter=0; %>
```

The Expression Tag: It holds java language expression that is evaluated during HTTP request and is included in HTTP response stream. The syntax is as follows:

```
<%= expression %>
```

Listing 3. Example using expression tag

```
The current time and date is : <%= new java.util.Date () %>
```

The Comment Tag: It is used to hide part of code in JSP page or to ignore some statements in the code. The syntax is as follows:

```
<% - -comment -- %>
```

Listing 4. Example using comment tag

```
<%-- This is a JSP comment -- %>
```

Following example demonstrates use of above scripting elements.

Listing 5.example.jsp

```
<%!int fontSize=3;%><br>
  <html>
  <head>
  <title>firstexample</title>
  </head>
   <body>

  <%-- addition Program --%><br>
  <%
     int a = 5;<br>
     int b = 3; <br>
     int result = a+b;

  %><br>
    <font color="green" size="<%= fontSize %>"><br>
  <%
          out.print(" Total=" +result);

  %><br>
  </font><br>
  </body>
  </html>
```

The output of the above program is:

```
Total=8
```

JSP Directives

Directives tell how to translate JSP page into servlet. There are three types of directives:

page directive
include directive
taglib directive

Directives usually in the following form:

```
<% @ directive attribute =”value” %>
```

JSP page Directive: It is used to modify translation of Jsp page. Syntax of JSP page directive is as follows:

```
<%@ page attribute="value"%>
```

Attributes of JSP page directive are as follows:

import
contentType
extends
info
buffer
language
isELIgnored
isThreadSafe

errorPage
isErrorPage
autoFlush
pageEncoding
session

Import: It is used to define classes and packages in servlet class. For Example:

Listing 6. Example using import

```
<html>   <body>
 <%@ page import="java.util.Date" %>
 Today is: <%= new java.util.Date() %>
 </body>
 </html>
```

ContentType: It defines MIME(Multipurpose Internet Mail Extension) of the output stream. Default is text/html. For example:

Listing 7. Example using contentType

```
<html>
  <body>
  <%@ page contentType="text/html"%>
  </body>
  </html>
```

Extends: It defines super class of servlet class generated by Jsp page. It is rarely used.
Info: Defines information about Jsp page. Information can be accessed by using getServletInfo() method. For example:

Listing 8. Example using info

```
<html>
  <body>
  <%@ page info="hello world..." %>
  </body>
  </html>
```

It can be accessed with method getServletInfo() in the servlet. For example:

```
public String getServletInfo() {
  return "hello world...";
}
```

Buffer: Defines size of the buffer used in the output stream. Default buffer size is 8KiloBytes or greater.

Here is an example:

Listing 9. Example using buffer

```
<html>   <body>
  <%@ page buffer="16kb" %>
  </body>
  </html>
```

Language: It defines the language used in Jsp page. Default value is "java". For example:

Listing 10. Example using language

```
<%@ page language="java" %>
```

isELIgnored: It specifies whether EL (Expression Language) elements are not processed on the page. The value could be true or false. Default value is false. For example:

Listing 11. Example using isELIgnored

```
<%@ page isELIgnored="true"%>
//If set to true then EL will not be evaluated.
```

IsThreadSafe: To control behavior of Jsp page we can use isThreadSafe attribute. The default value is true. If set to false, it will wait until the JSP finishes responding to a request before passing another request to it. For example:

Listing 12. Example using IsThreadSafe

```
<%@ page isThreadSafe="false"%>
```

errorPage: It is used to display an error page. If error occurs in current page then it will be redirected to the error page. For example:

Listing 13. Example using errorPage

```
<html>
  <body>
  <%@ page errorPage="errexample.jsp"%>
  <%=100/0 %>
  </body>
  </html>
```

Listing 14. errexample.jsp

```
<html>
```

```
<body>
<%@ page isErrorPage="true" %>
 Exception occurred...The exception is: <%= exception %>
</body>
</html>
```

isErrorPage: To declare a current page as the error page. For example:

Listing 15. Example using isErrorPage

```
<html>   <body>
 <%@ page isErrorPage="true" %>
  Exception occurred…The exception is: <%= exception %>
 </body>
 </html>
```

AutoFlush: It defines whether the output should be flushed automatically when buffer is filled. The default is true. For example:

Listing 16. Example using autoFlush

```
<%@ page buffer="10kb" autoflush="false"%>
//value of the autoFlush is false.
```

So, the page will not be flushed automatically because buffered size should be with at least 8kb.

pageEncoding: It defines character encoding of output stream. The default is ISO-8859-1.
Session: It defines whether JSP page is involved in HTTP session. For example:

```
<%=session.getAttribute ("user") %>
```

JSP include directive: It is used to include contents of files such as JSP file, html file or text file. This happens during translation phase. By using this directive we can reuse the code.

Syntax of JSP include directive is as follows:

```
<%@ include file="file name" %>
```

For Example:

Listing 17. Example using include

```
<html>
 <body>
 <%@ include file="hello.html" %>
 Today is: <%= new java.util.Date () %>
```

```
</body>
</html>
```

JSP taglib directive: It is used to define tag library that defines custom tags. We can use Tag Library Descriptor (TLD) file to define the tags. This directive includes URI and custom tag prefix. The URI is used to specify a location. The prefix is used to distinguish custom tags from libraries.

Syntax is as follows:

```
<%@ taglib uri=”tag library uri” prefix=”tag library prefix” %>
```

Example:

Listing 18. Example using taglib

```
<%@ taglib uri="http://www.google.com" prefix="c"%>
```

JSP Action Elements

The Action tags are used to control the flow between pages and to be used with Java Bean. We can insert a file, forward one page to another page or we can create HTML page for java plugin. JSP actions are XML tags that control the behavior of the JSP engine.

Jsp Action Tags are as follows:

jsp : include	It is used to include contents of files such as JSP files, html files or text files. This happens during translation phase.
jsp : forward	It is used to forward request to another resource or page .It may be html page, JSP page or other resource.
jsp : usebean	Used to create specified object
jsp : setProperty	Gives details about java bean object
jsp : getProperty	It is used to retrieve value of given property
jsp : element	To create dynamic xml elements
jsp : attribute	To define dynamic XML elements attribute
jsp : body	To define dynamic XML elements body
jsp : text	Used to write text in JSP pages and documents

<jsp: include> Action Tag: It is used to include files such as JSP, html or servlet. This happens during translation phase. This tag includes file when the page is requested.

The Syntax of this tag is as follows:

```
<jsp: include page=”relative URL” />
```

For example, we will create JSP file called newdate.jsp whose content is included in includeexample.jsp.

Listing 19.includeexample.jsp

```
<html>
  <body>
  <jsp:include page="newdate.jsp"/>
  </body>
  </html>
```

Listing 20.newdate.jsp

```
  <html>
  <body>
  <%out.println("today's date is:"+new java.util.Date()%>
  </body>
  </html>
```

The <jsp: forward> Action Tag: It is used to forward request to another page it may be JSP, html or another resource. The Syntax of this tag is as follows:

```
<jsp: forward page=”Relative URL”/>
```

The <jsp: usebean> Action Tag: It is used to instantiate a bean class. It searches for id and scope variables. If object is not created, then it creates the specified object.

The Syntax of this tag is as follows:

```
<jsp: usebean id=”name” scope=”page|request|session|application”
  class=”packageName.classname” type=” packageName.classname”
    beanName=” packageName.classname”>
  </jsp: usebean>
```

JSP implicit Objects

Implicit objects are java objects that are created by the container while translating JSP pages to servlets. There are 9 implicit objects as listed below:

out
request
response

session
application
config
pageContext
page
exception

The out object: It is used to send content or output to the client. This object is instance of javax.servlet.jsp.JspWriter. It sends content in a response.

Listing 29. Example using out object

```
<html>
 <body>
 <% out.println ("today’s date is:"+new java.util.Date ());
 %>
 </body>
 </html>
```

The request object: When the client requests a page, the JSP engine creates a new object to that request. It is object of type HttpServletRequest.

Example: Create one html file as index.html

Listing 30.index.html

```
<html>
 <body>
 <form action="hello.jsp">
 <input type="text" name="username"/>
 <input type="submit" value="submit"/>
 </form>
 </body>

</html>
```

Now create jsp file as hello.jsp. This file should be the same as the one in html file.

Listing 31.hello.jsp

```
<html>
 <body>
 <%
 String name=request.getParameter ("username");
 out.println("hi"+name);
 %>
 </body>
 </html>
```

The response object: It creates an object to respond to the client. It is an object of type HttpServletResponse.

Example: Create one html file: sample.html

Listing 32.sample.html

```
<html>
 <body>
 <form action="myResponse.jsp">
 <input type="text" name="username"/>
 <input type="submit" value="submit"/>
 </form>
 </body>
 </html>
```

Now create jsp file as myResponse.jsp

Listing 33.myResponse.jsp

```
<html>
 <body>
 <%
 response.sendRedirect ("http://www.yahoo.com");
 %>
 </body>
 </html>
```

The session object: It is used to set or get session information. A session object is created by the container whenever we request a JSP page.

Listing 34. Example using session object

```
<html>
 <body>
 <%=session.getId () %>
 </body>
 </html>
```

The application object: It is used to get information and attributes in JSP. It also used to forward the request to another resource or to include the response from another resource.

Listing 35. Example using appli object

```
<html>
  <body>
  <%=application.getInitParameter ("User") %>
  </body>
  </html>
```

The config object: It is used to get configuration message for a jsp page. You can also use it get the init parameter present in web.xml.

Listing 36. Example using config object

```
<html>
  <body>
  <%=config. getInitParameter ("User") %>
  </body>
  </html>
```

The pageContext object: It is used to set and get attributes and forward request to other resources.

Listing 37. Example using pageContext object

```
<html>
  <body>
  <% pageContext.setAttribute ("tek", "buds"); %>
  <p>PageContext attribute</p> :{ Name="tek", Value="<%=pageContext.getAttribute ("tek") %>"}
  </body>
  </html>
```

The page object: It is an instance of the object class and references the current jsp page. It is rarely used . It is written as:

```
Object page=this
```

Listing 38. Example using page object

```
<html>
  <body>
  <%=page.getClass ().getName () %>
  </body>
  </html>
```

The exception Object: It is used to display exception in jsp error pages. It is an instance of the Throwable’s subclass. It is available only in error pages.

JSP Scripting Elements

In this tutorial we are going to learn how to use JSP Expressions. In JSP there are three types of scripting elements:

JSP Expressions: It is a small java code which you can include into a JSP page. The syntax is "<%= some java code %>"
JSP Scriptlet: The syntax for a scriptlet is "<% some java code %>". You can add 1 to many lines of Java code in here.
JSP Declaration: The syntax for declaration is "<%! Variable or method declaration %>", in here you can declare a variable or a method for use later in the code.

JSP Expressions

Using the JSP Expression you can compute a small expression, always a single line, and get the result included in the HTML which is returned to the browser. Using the code we have previously written, let's explore expressions.

Code

```
The time on the server is <%= new java.util.Date() %>
```

Output

```
The time on the server is Thursday January 21 07:21:43 GMT 2016.
```

Explanation

Here the "new java.util.Date()" is processed into the actual date and time shown through HTML on the browser. Let's explore expressions through a couple of more examples.

Examples

In the first example we are going to see an expression for converting a string from lower case to upper case. Here is the code:

The Expression: Converting a string to uppercase <%= new String("Hello World").toUpperCase() %>

Here we are creating a "String" object with "Hello World" set as the value for object. Following that we are calling a Java function ".toUpperCase" to convert the string from lower case to upper case.

The HTML: Converting a string to uppercase: HELLO WORLD

You can also make use of mathematical expressions in JSP.

The Expression: 25 multiplied to 4: <%= 25*4 %>

JSP Scriptlets

This JSP Scripting Element allows you to put in a lot of Java code in your HTML code. This Java code is processed top to bottom when the page is the processed by the web server. Here the result of the code isn't directly combined with the HTML rather you have to use "out.println()" to show what you want to mix with HTML. The syntax is pretty much the same only you don't have to put in an equal sign after the opening % sign.

Code

```
<h2> Hello World</h2>

<%

for(inti=0; i<= 5; i++)

{

out.println("<br/> I really love counting: " + i);

}

%>
```

Output:

```
I really love counting: 1

I really love counting: 2

I really love counting: 3
```

```
I really love counting: 4

I really love counting: 5
```

Explanation:

In this example we have set up a basic h2 heading and following that we have a “for loop” in the scriptlet. Just to remember println means print line. In every iteration of the loop we print the “I really love counting” and appends it with the integer value of the “i” printed through the HTML.

Just try to make sure that you don’t put in a lot of code in a scriptlet in JSP. This will make it readable and easy to manage. If you can’t help it, try to refactor this code into different Java classes and make use of MVC to keep it under control.

JSP Declarations

The declarations come in handy when you have a code snippet that you want executed more than once. Using the declaration, you can declare the method in the beginning of the code and then call the same method whenever you need in the same page. The syntax is simple:

```
<%!

//declare a variable or a method

%>
```

Code

Here is the method declaration:

```
<%!

String makeItLower(String data)

{

returndata.toLowerCase();

}

%>
```

Here is how you call it

```
Lower case “Hello World”:<%= makeItLower(“Hello World”) %>
```

Output

Lower case “Hello World”: hello world

Explanation

In the method declaration you have your standard java method with the return type of a String. You take a string as an argument and return it converted to lower case. Later we call this function through a JSP Expression.

Java RMI client/server Application

To write an RMI Java application, you would have to follow the steps given below −

Define the remote interface

Develop the implementation class (remote object)

Develop the server program

Develop the client program

Compile the application

Execute the application

Defining the Remote Interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

To create a remote interface −

Create an interface that extends the predefined interface Remote which belongs to the package.

Declare all the business methods that can be invoked by the client in this interface.

Since there is a chance of network issues during remote calls, an exception named RemoteException may occur; throw it.

Following is an example of a remote interface. Here we have defined an interface with the name Hello and it has a method called printMsg().

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Hello extends Remote {

   void printMsg() throws RemoteException;

}
```

Developing the Implementation Class (Remote Object)

We need to implement the remote interface created in the earlier step. (We can write an implementation class separately or we can directly make the server program implement this interface.)

To develop an implementation class −

Implement the interface created in the previous step.

Provide implementation to all the abstract methods of the remote interface.

Following is an implementation class. Here, we have created a class named ImplExample and implemented the interface Hello created in the previous step and provided body for this method which prints a message.

```
// Implementing the remote interface

public class ImplExample implements Hello {

   // Implementing the interface method

   public void printMsg() {

      System.out.println("This is an example RMI program");

   }

}
```

Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the RMIregistry.

To develop a server program −

Create a client class from where you want invoke the remote object.

Create a remote object by instantiating the implementation class as shown below.

Export the remote object using the method exportObject() of the class named UnicastRemoteObject which belongs to the package java.rmi.server.

Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry.

Bind the remote object created to the registry using the bind() method of the class named Registry. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

```
import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {

  public Server() {}

  public static void main(String args[]) {

    try {

      // Instantiating the implementation class

      ImplExample obj = new ImplExample();
```

```
         // Exporting the object of implementation class

         // (here we are exporting the remote object to the stub)

         Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);



         // Binding the remote object (stub) in the registry

         Registry registry = LocateRegistry.getRegistry();

         registry.bind("Hello", stub);

         System.err.println("Server ready");

      } catch (Exception e) {

         System.err.println("Server exception: " + e.toString());

         e.printStackTrace();

      }

   }

}
```

Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program −

Create a client class from where your intended to invoke the remote object.

Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry.

Fetch the object from the registry using the method lookup() of the class Registry which belongs to the package java.rmi.registry.

To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.

The lookup() returns an object of type remote, down cast it to the type Hello.

Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```
import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class Client {

  private Client() {}

  public static void main(String[] args) {

    try {

      // Getting the registry

      Registry registry = LocateRegistry.getRegistry(null);

      // Looking up the registry for the remote object

      Hello stub = (Hello) registry.lookup("Hello");

      // Calling the remote method using the obtained object

      stub.printMsg();

      // System.out.println("Remote method invoked");

    } catch (Exception e) {

      System.err.println("Client exception: " + e.toString());

      e.printStackTrace();

    }

  }
```

}

Compiling the Application

To compile the application –

Compile the Remote interface.

Compile the implementation class.

Compile the server program.

Compile the client program.

Executing the Application

Step 1 – Start the **rmi** registry using the following command.

start rmiregistry

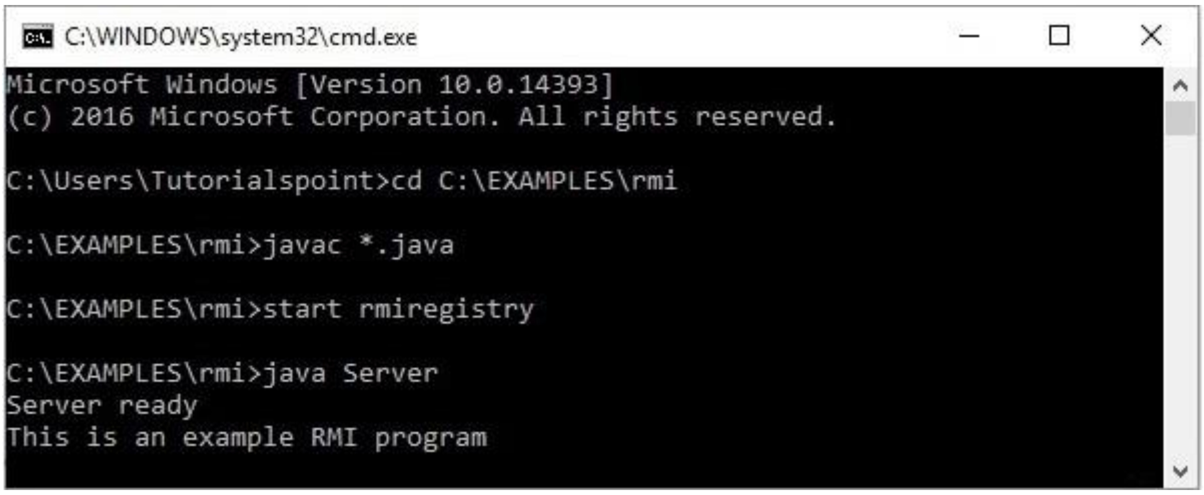
Step 2 – Run the server class file as shown below.

Java Server

Step 3 – Run the client class file as shown below.

java Client

Verification – **As soon you start the client, you would see the following output in the server.**



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi

C:\EXAMPLES\rmi>javac *.java

C:\EXAMPLES\rmi>start rmiregistry

C:\EXAMPLES\rmi>java Server
Server ready
This is an example RMI program
```

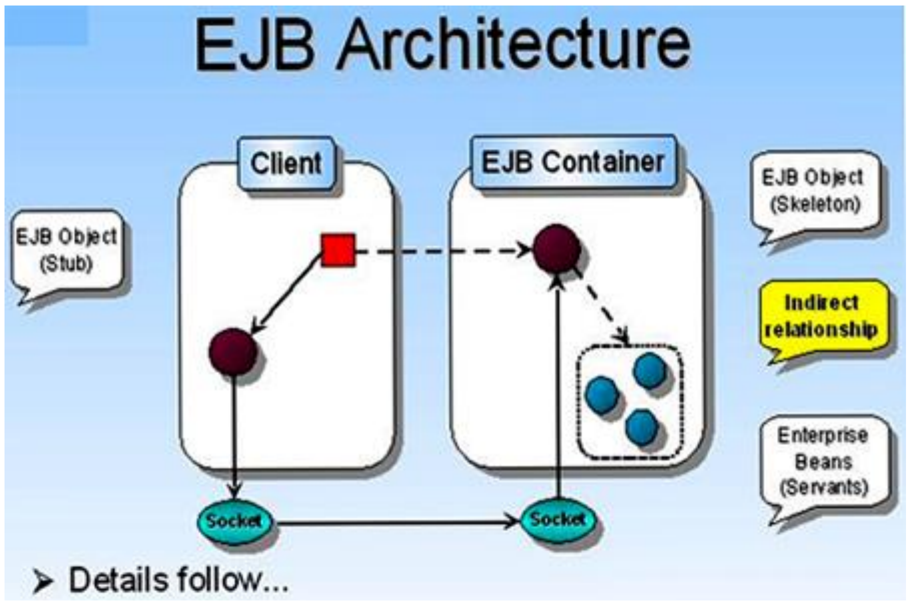
Unit-V

EJB Architecture

Enterprise JavaBeans (EJB) is the server-side and platform-independent Java application programming interface (API) for Java Platform, Enterprise Edition (Java EE). EJB is used to simplify the development of large distributed applications.
Enterprise JavaBeans (EJB) are reusable Java components that implement business logic and enable you to develop component-based distributed businessapplications.

EJBs reside in an EJB container, which provides a standard set of services such as
Persistence,
Security,
Transactions, and
Concurrency.
Enterprise JavaBeans are the standard for defining server-side components. WebLogic Server's implementation of the Enterprise JavaBeans component architecture is based on Sun Microsystems EJB specification.

Simple architecture of EJB



The EJB specification defines the following **four** types of Enterprise JavaBeans:

Stateless session. These non-persistent EJBs provide a service without storing an interaction or conversation state between methods.
Stateful session. These non-persistent EJBs maintain state across methods and transactions. Each instance is associated with a particular client.
Entity. These persistent EJBs represent an object view of the data, usually rows in a database. An entity bean has a primary key as a unique identifier.
Message-driven. These EJBs are integrated with the Java Message Service (JMS) to enable message-driven beans to act as a standard JMS message consumer and perform asynchronous processing between the server and the JMS message producer.

From the EJB sub-node, of the Deployments node of the Administration Console, you can update or configure EJBs deployed on WebLogic Server or monitor the performance.

Configuring EJBs

To configure an EJB for deployment to the WebLogic Server Administration Console:

Start the WebLogic Server Administration Console.
Select the Domain in which you will be working.
In the left pane of the Console, click Deployments.
In the left pane of the Console, click EJB. A table is displayed in the right pane of the Console showing all the deployed EJBs.
Select the Configure a new EJB option.
Locate the EAR, WAR or JAR file you would like to configure. You can also configure an exploded application or component directory. Note that WebLogic Server will deploy all components it finds in and below the specified directory.
Creating the application in EJB:
Click [select] to the left of a directory or file to choose it and proceed to the next step.
Select a Target Server from among Available Servers.
Enter a name for the EJB or application in the provided field.
Click Configure and Deploy. The Console will display the Deploy panel, which lists deployment status and deploymen tactivities for the EJB.

Enterprisebeans

Enterprise beans are typically deployed in EJB containers and run on EJB servers.
An enterprise bean is a non-visual component of a distributed, transaction-oriented enterprise application. You can customize them by changing their deployment descriptors and you can assemble them with other beans to create new applications.

There are three types of enterprise beans:

Session beans
Entity beans
Message-driven beans.

Session beans: Session beans are non-persistent enterprise beans. They can be stateful or stateless.

A stateful session bean acts on behalf of a single client and maintains client-specific session information (called conversational state) across multiple method calls and transactions. It exists for the duration of a single client/server session.
A stateless session bean, by comparison, does not maintain any conversational state. Stateless session beans are pooled by their container to handle multiple requests from multiple clients.
Entity beans: Entity beans are enterprise beans that contain persistent data and that can be saved in various persistent data stores. Each entity bean carries its own identity.

Bean-managed persistence (BMP) Entity beans that manage their own persistence are called bean-managed persistence (BMP) entity beans.
Container-managed persistence (CMP) Entity beans that delegate their persistence to their EJB

container are called container-managed persistence (CMP) entity beans.
Message-driven beans: Message-driven beans are enterprise beans that receive and process JMS messages. Unlike session or entity beans, message-driven beans have no interfaces. They can be accessed only through messaging and they do not maintain any conversational state. Message-driven beans allow asynchronous communication between the queue and the listener, and provide separation between message processing and business logic.

Overview of EJB

Enterprise Java Beans (EJB) is one of the several Java APIs for standard manufacture of enterprise software. EJB is a server-side software element that summarizes business logic of an application. Enterprise Java Beans web repository yields a runtime domain for web related software elements including computer reliability, Java Servlet Lifecycle (JSL) management, transaction procedure and other web services. The EJB enumeration is a subset of the Java EE enumeration.

The EJB enumeration was originally developed by IBM in 1997 and later adopted by Sun Microsystems in 1999 and enhanced under the Java Community Process.

The EJB enumeration aims to provide a standard way to implement the server-side business software typically found in enterprise applications. Such machine code addresses the same types of problems, and solutions to these problems are often repeatedly re-implemented by programmers. Enterprise Java Beans is assumed to manage such common concerns as endurance, transactional probity and security in a standard way that leavs programmers free to focus on the particular parts of the enterprise software at hand.

To run EJB application we need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

1. Life cycle management
2. Security
3. Transaction management
4. Object pooling

Types of Enterprise Java Beans

There are **three** types of EJB:

1. ***Session Bean:*** Session bean contains business logic that can be invoked by local, remote or webservice client. There are two types of session beans: (i) Stateful session bean and (ii) Stateless session bean.

(i) Stateful Session bean :

Stateful session bean performs business task with the help of a state. Stateful session bean can be used to access various method calls by storing the information in an instance variable. Some of the applications require information to be stored across separate method calls. In a shopping site, the items chosen by a customer must be stored as data is an example of stateful session bean.

(ii) Stateless Session bean :

Stateless session bean implement business logic without having a persistent storage mechanism, such as a state or database and can used shared data. Stateless session bean can be used in situations where information is not required to used across call methods.

2. ***Message Driven Bean:*** Like Session Bean, it contains the business logic but it is invoked by passing message.

3. ***Entity Bean:*** It summarizes the state that can be remained in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API). There are two types of entity bean:

(i) Bean Managed Persistence :

In a bean managed persistence type of entity bean, the programmer has to write the code for database calls. It persists across multiple sessions and multiple clients.

(ii) Container Managed Persistence :

Container managed persistence are enterprise bean that persists across database. In container managed persistence the container take care of database calls.

When to use Enterprise Java Beans

1.Application needs Remote Access. In other words, it is distributed.
2.Application needs to be scalable. EJB applications supports load balancing, clustering and fail-over.
3.Application needs encapsulated business logic. EJB application is differentiated from demonstration and persistent layer.

Advantages of Enterprise Java Beans

1. EJB repository yields system-level services to enterprise beans, the bean developer can focus on solving business problems. Rather than the bean developer, the EJB repository is responsible for system-level services such as transaction management and security authorization.
2. The beans rather than the clients contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the pattern that execute business rules or access databases. Due to this the clients are thinner which is a benefit that is particularly important for clients that run on small devices.
3. Enterprise Java Beans are portable elements, the application assembler can build new applications from the beans that already exists.

Disadvantages of Enterprise Java Beans

1. Requires application server
2. Requires only java client. For other language client, you need to go for webservice.
3. Complex to understand and develop EJB applications.

Building and Deploying EJBs

Writing the EJB:

EJB must provide three required classes:

EJB implementation Class

Home Interface

Remote Interface

Each EJB must provided with a deployment descriptor

The deployment descriptor is serialized instance of a Java class that contains information about how to deploy the bean.

Two flavors of deployment descriptors:

Session Descriptors – apply to Session EJBs

Entity Descriptors – apply to Entity EJBs

A Deployment Descriptor contains information such as the following:

The name of the EJB Class

The name of the EJB Home Interface

The name of the EJB Remote Interface

ACLs of entities authorized to use each class or method.

For Entity Beans, a list of container – managed fields.

For Session Beans, a value denoting whether the bean is stateful or stateless.

Properties File may be generated, which contains environment properties that will be activated during runtime.

Manifest File is needed to create the ejb-jar file, which is the medium used to distribute EJBs.

Name : <filename>

Enterprise-Bean:True

Deploying the EJBs:

At the deployment time, the EJB container must read the ejb-jar file, create implementations for the home and remote interfaces.

Reads the deployment-descriptor and ensures that the bean has what it wants and ad the bean's property settings to the environment so that they're available to the bean at runtime.

Connecting to the EJB

The client can use either RMI or CORBA to connect to the EJB

The client looks up the EJB's home interface using a naming service (JNDI or COS)

Invokes a find() or create() method on the home interface to get a reference to an EJB object.

And then uses the EJB object as if it were an ordinary object.

Roles in EJB

Each or all roles can be performed at once or at different levels by an individual or an organization.

Bean Provider:

A bean provider is an EJB developer (provides beans by developing) who has to be familiar with the business logic for an enterprise application.

Application Assembler:

In an EJB application, several beans are developed by the bean developers. The role of an application assembler is to assemble all the beans that are developed by the application assemblers. He also writes the deployment descriptor.

Deployer:

The deployer is one whose responsibility is to deploy the EJB application in a particular EJB container(s). His inputs are enterprise beans and deployment descriptors from bean developers and application assemblers. He needs to be expertise the functionality of the EJB container.

System Administrator:

The system administrator is responsible to maintain the deployed EJB applications very frequently. He has to see that the EJBs run 24x7 environment without any interruption including the restart of a server in case of its crash. In addition to these, he is also responsible to maintain the security of the users.

EJB Server Provider:

The EJB Server Provider provides the EJB server that hosts the EJB container. Most of the EJB containers are packaged with the EJB Servers.

EJB Container Provider:

The EJB container provider provides the resources to write an EJB container and conforms that the software is on par to the EJB specifications. He should also provide the required softwares and tools to the administrator in order to administer the EJB applications.

EJB Design and Implementation

Design and develop EJBs

EJB architecture

Enterprise beans

An *enterprise bean* is a non-visual component of a distributed, transaction-oriented enterprise application. Enterprise beans are typically deployed in EJB containers and run on EJB servers. You can customize them by changing their deployment descriptors and you can assemble them with other beans to create new applications. There are three types of enterprise beans: *session beans*, *entity beans*, and *message-driven beans*.

Session beans: *Session beans* are non-persistent enterprise beans. They can be stateful or stateless. A *stateful session bean* acts on behalf of a single client and maintains client-specific session information (called conversational state) across multiple method calls and transactions. It exists for the duration of a single client/server session. A *stateless session bean*, by comparison, does not maintain any conversational state. Stateless session beans are pooled by their container to handle multiple requests from multiple clients.
Entity beans: *Entity beans* are enterprise beans that contain persistent data and that can be saved in various persistent data stores. Each entity bean carries its own identity. Entity beans that manage their own persistence are called *bean-managed persistence (BMP)* entity beans. Entity beans that delegate their persistence to their EJB container are called *container-managed persistence (CMP)* entity beans.
Message-driven beans: *Message-driven beans* are enterprise beans that receive and process JMS messages. Unlike session or entity beans, message-driven beans have no interfaces. They can be accessed only through messaging and they do not maintain any conversational state. Message-driven beans allow asynchronous communication between the queue and the listener, and provide separation between message processing and business logic.
Remote client view

The *remote client view* specification became available beginning with EJB 1.1. The remote client view of an enterprise bean is location independent. A client running in the same JVM as a bean instance uses the same API to access the bean as a client running in a different JVM on the same or different machine.

Remote interface: The *remote interface* specifies the remote business methods that a client can call on an enterprise bean.
Remote home interface: The *remote home interface* specifies the methods used by remote clients for locating, creating, and removing instances of enterprise bean classes.
Local client view

The *local client view* specification is available in EJB 2.0 or later. Unlike the remote client view, the local client view of a bean is location dependent. Local client view access to an enterprise bean requires both the local client and the enterprise bean that provides the local client view to be in the same JVM. The local client view therefore does not provide the location transparency provided by the remote client view. Local interfaces and local home interfaces provide support for lightweight access from enterprise bean that are local clients. Session and entity beans can be tightly couple with their clients, allowing access without the overhead typically associated with remote method calls.

Local interface: The *local interface* is a lightweight version of the remote interface, but for local clients. It includes business logic methods that can be called by a local client.
Local home interface: The *local home interface* specifies the methods used by local clients for locating, creating, and removing instances of enterprise bean classes.

Web service client view

In the EJB 2.1 specification, the EJB architecture introduced the support for Web services. A client for a session bean can be a Web service client. A Web service client can make use of the Web service client view of a stateless session bean, which has a corresponding service endpoint interface.

Service endpoint interface

The *service endpoint interface* for a stateless session bean exposes the functionality of the session bean as a Web service endpoint. The Web Service Description Language (WSDL) document for a Web service describes the Web service as a set of endpoints operating on messages. A WSDL document can include the service endpoint interface of a stateless session bean as one of its endpoints. An existing stateless session bean can be modified to include a Web service client view, or a service endpoint interface can be mapped from an existing WSDL to provide the correct interface.

A Web service client view is independent of location and can be accessed through remote calls.

EJB client JAR file

An *EJB client JAR* file is an optional JAR file that can contain the client interfaces that a client program needs to use the client views of the enterprise beans that are contained in the EJB JAR file. If you decide not to create an EJB client JAR file for an EJB module, all of the client interface classes will be in the EJB JAR file. By default, the workbench creates EJB client JAR projects for each corresponding EJB project.

EJB container

An *EJB container* is a runtime environment that manages one or more enterprise beans. The EJB container manages the life cycles of enterprise bean objects, coordinates distributed transactions, and implements object security. Generally, each EJB container is provided by an EJB server and contains a set of enterprise beans that run on the server.

Deployment descriptor

A *deployment descriptor* is an XML file packaged with the enterprise beans in an EJB JAR file or an EAR file. It contains metadata describing the contents and structure of the enterprise beans, and runtime transaction and security information for the EJB container.

EJB server

An *EJB server* is a high-level process or application that provides a runtime environment to support the execution of server applications that use enterprise beans. An EJB server provides a JNDI-accessible naming service, manages and coordinates the allocation of resources to client applications, provides access to system resources, and provides a transaction service. An EJB server could be provided by, for example, a database or application server.

EJB development resources

EJB modules

EJB modules are displayed in the **Project Explorer** view of the **J2EE** perspective, and they correspond to EJB projects.

An EJB module is used to assemble one or more enterprise beans into a single deployable unit. An EJB module is developed in an EJB project, and it can be exported as either a standalone EJB JAR file, or it can be combined with other EJB or Web modules within an enterprise application. An EJB JAR file uses the format of a standard Java archive file. An EJB module contains the following:

One or more enterprise beans and their associated .class and .java files.
Graphics and other files depending on the need of the enterprise bean.
A deployment descriptor. The file type for the deployment descriptor is Extensible Markup Language (XML). This file declares the contents of the EJB module, defines the structure of the beans in the module, and provides a description of how the beans are to be used at run time.
a MANIFEST.MF file in the META-INF directory. The manifest file can contain a class path entry, with references to other JAR files or EJB modules in a J2EE enterprise application. It defines the module's external dependencies.
IBM extensions to the standard deployment descriptor, if the target runtime container is WebSphere Application Server.

An EJB module is installed and runs in an EJB container.

An enterprise bean is a Java component that can be combined with other resources to create distributed client/server applications.

Note: If you choose to create an EJB client JAR file for your EJB module, the client interface classes for the enterprise beans will not be included in the EJB JAR file, but are included in the EJB client JAR file.

EJB projects

In the workbench, you create and maintain resources for enterprise applications in projects. An EJB project is a logical module that allows you to organize your enterprise beans. In the **Project Explorer** view, an EJB project is shown as an EJB module.

The workbench supports EJB 1.1, EJB 2.0, and EJB 2.1 projects. The J2EE specification level of a containing EAR project must be set to J2EE 1.3 or higher for EJB 2.0 projects, and J2EE 1.4 for EJB 2.1 projects. In an EJB 1.1 project, you will only be able to create EJB 1.1 beans.

An EJB project is a specialized Java project. The source and the output files of the project are located in the ejbModule folder. As you make changes and generate deployment code, the Java classes are compiled into the ejbModule folder. You cannot use the EJB project as the source folder; doing so will cause errors.

The imported classes are located in the imported_classes folder. The folder is only created when no source exists for any .class file in an imported JAR file, and only .class files for which no source exists are added to the folder. Whenever the project is built, the classes from the imported_classes folder are copied to the ejbModule folder. If you later add source for an imported class, the .class file in the imported_classes folder will be ignored; however, you should delete the .class file when the .java file is added.

Note: If you choose to create an EJB client JAR file for your EJB module, the client interface classes for the enterprise beans will not be included in the EJB project, but in separate EJB client JAR project. EJB client JAR projects are displayed in the **Project Explorer** as Java projects under the **Other projects** node.

EJB client projects

The EJB tooling supports the creation of EJB client JAR projects for EJB modules. An EJB client JAR project contains all the interface classes that a client program needs to use the client views of the enterprise beans that are contained in the EJB project. When you create an EJB client project for an EJB project, a new Java project is created and added to your workspace. The EJB client project is added as a project utility JAR file to each module that the EJB project belongs to.

By default, when you use the wizard to create an EJB project, an EJB client JAR project is also created. However, you can clear this option in the wizard.

Tip: You can also add the EJB client project to another enterprise application that does not include the EJB project as a module. This will ensure that the EJB client JAR file is exported and packaged with the EAR file when the application is exported.

Enterprise beans

An enterprise bean is a Java component that can be combined with other resources to create distributed client/server applications.

There are three types of enterprise beans: entity beans, session beans, and message-driven beans. Typically, all types of beans are used together within an enterprise application.

Entity beans

Entity beans store permanent data. Entity beans with container-managed persistence (CMP) require database connections. Entity beans with bean-managed persistence manage permanent data in whichever manner is defined in the bean code. This can include writing to databases or XML files, for example.

Session beans

Session beans *do not require* database access, though they can obtain it indirectly (as needed) by accessing entity beans. Session beans can also obtain direct access to databases (and other resources) through the use of resource references.

Message-driven beans

Message-driven beans are a special kind of enterprise bean that acts as a message consumer in the JMS messaging system. As with standard JMS message consumers, message-driven beans perform business logic based on the message contents. In several ways, the dynamic creation and allocation of message-driven bean instances mimics the behavior of stateless session enterprise beans. However, message-driven beans are different from stateless session enterprise beans (and other types of enterprise beans) in a couple of ways:

Message-driven beans process multiple JMS messages asynchronously, rather than processing a serialized sequence of method calls.
Message-driven beans have no home or remote interface, and therefore cannot be directly accessed by internal or external clients.

Beans requiring data access use *data sources*, administrative resources defining pools of database connections.

- **Deployment descriptors**

A deployment descriptor contains configuration data that the runtime environment uses for an application. A deployment descriptor can include information about the following:

The structure and content (enterprise beans, for example) of the application.
References to internal and external dependencies. For example, an enterprise bean in an EJB module can require another enterprise bean that is not bundled in the same module.
References to resource factory objects, such as URLs or JDBC data sources.
Security roles that the container uses when implementing the required access control for the application.
Transactional information about how (and whether) the container is to manage transactions for the application.

Deployment descriptors are XML files packaged with the application's files in a Java archive file. An EJB deployment descriptor is called ejb-jar.xml and is located in the META-INF folder of an EJB project. A J2EE application contains one application-level deployment descriptor file, governing the application as a whole. It also contains several component-level deployment descriptors, one for each module in the application.

In addition to the standard deployment descriptor, the workbench also includes information on WebSphere Application Server bindings and extensions. The bindings and extensions documents are specific to IBM. Both binding and extension descriptors are stored in XMI files, ibm-ejb-jar-bnd.xmi and ibm-ejb-jar-ext.xmi, respectively. Binding information maps a logical name of an external dependency or resource to an actual JNDI name. For example, the container uses binding information to locate a remote bean at installation. Extensions are additions to the standard descriptors. The extensions enable older (legacy) systems to work in the WebSphere Application Server environment. They are also used to specify application behavior that is vendor-specific, undefined in a current specification, or expected to be included in a future specification.

Mapping documents (map.mapxmi)

The Mapping editor helps you to map enterprise beans to databases. The map.mapxmi file holds this mapping information.

Developing enterprise beans

Create an EJB project.
Create enterprise beans or import beans to your EJB project.
Add methods to the home and remote interfaces.
Add custom finders or add EJBQL queries where necessary.

Add and define additional CMP fields when necessary.
Define relationships between beans as needed.
Define other EJB deployment descriptor elements (i.e. references).
Map enterprise beans to relational database tables:
Generate a **top-down** mapping (create database tables from existing beans and map them to each other)
Generate a **bottom-up** mapping (create beans from a database schema and map them to each other)
Generate a **meet-in-the-middle** mapping (use existing beans and tables and map them to each other)
Update maps with the **Mapping editor**.
Create session bean facades or create EJB access beans for client applications to use.
Generate deployment code for your enterprise beans (optionally, you can generate deployment code from the command line.
Test the enterprise beans.
Export your EJB project to an EJB JAR file.

Generating a bottom-up mapping

You can use the bottom-up mapping approach to generate CMP enterprise beans and mappings from an existing database schema.

You can use the EJB mapping wizard in the EJB tools and use the following development approaches for mapping enterprise beans to database tables. If you already have entity beans in a project, the mapping gives unique names to generated beans to avoid collision of new table mappings to existing CMP entity beans. By default, relationships are generated where foreign-keys exist.

Container-Managed Transactions

In an enterprise bean with container-managed transactions, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session, entity, or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction.

Typically, the container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When deploying a bean, you specify which of the bean's methods are associated with transactions by setting the transaction attributes.

A transaction attribute controls the scope of a transaction. A transaction attribute may have one of the following values:

Required

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The Required attribute will work for most transactions. Therefore, you may want to use it as a default, at least in the early phases of development. Because transaction attributes are declarative, you can easily change them at a later time.

RequiresNew

If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:

Suspends the client's transaction
Starts a new transaction
Delegates the call to the method
Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the RequiresNew attribute when you want to ensure that the method always runs within a new transaction.

Mandatory

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws the TransactionRequiredException.

Use the Mandatory attribute if the enterprise bean's method must use the transaction of the client.

NotSupported

If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the NotSupported attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

Supports

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the Supports attribute with caution.

Never

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a RemoteException. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

You can specify the transaction attributes for the entire enterprise bean or for individual methods. If you've specified one attribute for a method and another for the bean, the attribute for the method takes precedence. When specifying attributes for individual methods, the requirements differ with the type of bean.

Session beans need the attributes defined for business methods, but do not allow them for the create methods.

Entity beans require transaction attributes for the business, create, remove, and finder methods.

Message-driven beans require transaction attributes (may use either Required or NotSupported) for the onMessage method.

Session Beans Constraints

Session beans disappear from the server, if

The session bean’s timeout value expires

The server crashes and is restarted

The server is shutdown and restarted

If any of these event occurs, state information of the bean is lost.

Solutions:

Session timeout value is configurable so the timeout value can be increased.

Server crashes and restarts are fairly rare events.

The user can be notified that non-recoverable error has occurred and ask the user to try again later.

A copy of the state of the bean can be cached at the client side and a new bean can be recreated in the event of original bean dies.

Use a database on the server side to hold the state of the client’s conversation. If the session bean dies during the conversation, the client can reconnect and pass in a unique customer identifier to query the database to retrieve whatever state information was saved prior to the failure of the session beans.

A session bean's state is not transactional. If the internal state of the bean is modified, the rollback will not affect the session bean's internal state.

Solution:

The session bean's state should be reset manually to whatever it was before the transaction began.

A stateless session bean cannot implement the SessionSynchronization interface. Because stateless session beans lack a conversational state and cannot hold a transaction across method invocations.

The final constraint on session beans is that they are not reentrant. If one thread is currently using a session bean, the container will throw a java.rmi.RemoteException in response to any other requests to connect to the session bean.

Life Cycle of a Session Bean

This topic discusses the life cycle methods of session beans and contains the following sections:

Life Cycle of a Stateless Session Bean

Life Cycle of a Stateful Session Bean

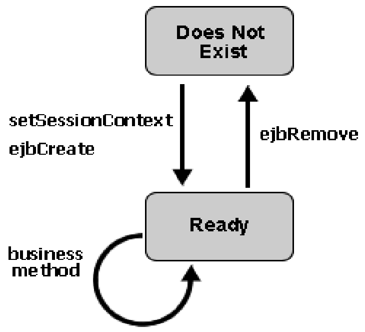
Life Cycle of a Stateless Session Bean

The following figure shows the life cycle of a stateless session bean. A stateless session bean has two states:

Does not exist. In this state, the bean instance simply does not exist.

Ready state. When WebLogic Server is first started, several bean instances are created and placed in the Ready pool. More instances might be created by the container as needed by the EJB container.

The various state transitions as well as the methods available during the various states are discussed below.



Moving from the Does Not Exist to the Ready State

When the EJB container creates a stateless session bean instance to be placed in the ready pool, it calls the callback method public void setSessionContext(SessionContext ctx). This method has the parameter javax.ejb.SessionContext, which contains a reference to the session context, that is, the interface to the EJB container, and can be used to self-reference the session bean object. Complete details about the javax.ejb.SessionContext can be found in your favorite J2EE documentation and the API reference at http://java.sun.com.

After the callback method setSessionContext is called, the EJB container calls the callback method ejbCreate. You can implement this callback method to, for instance, obtain the home interfaces of other EJBs invoked by the session bean, as shown in Defining a Session Bean. The ejbCreate method is only called once during the lifetime of a session bean, and is not tied to the calling of the create method by a client application. For a stateless session bean, calling the create method returns a reference to a bean instance already in the ready pool; it does not create a new bean instance. The management of stateless session bean instances is fully done by the EJB container.

Ready State

When a bean instance is in the ready state, it can service client requests; that is, execute component methods. When a client invokes a business method, the EJB container assigns an available bean instance to execute the business method. Once execution has finished, the session bean instance is ready to execute another business method.

Moving from the Ready to the Does Not Exist State

When the EJB container decides to reduce the number of session bean instances in the ready pool, it makes the bean instance ready for garbage collection. Just prior to doing this, it calls the callback method ejbRemove. If your session bean needs to execute some cleanup action prior to garbage collection, you can implement it using this callback method. The callback method is not tied to the remove method invoked by a client. For a stateless session bean, calling the remove method invalidates the reference to the bean instance already in the ready pool, but it does not move a bean instance from the ready to the does not exist state, as the management of stateless session bean instances is fully done by the EJB container.

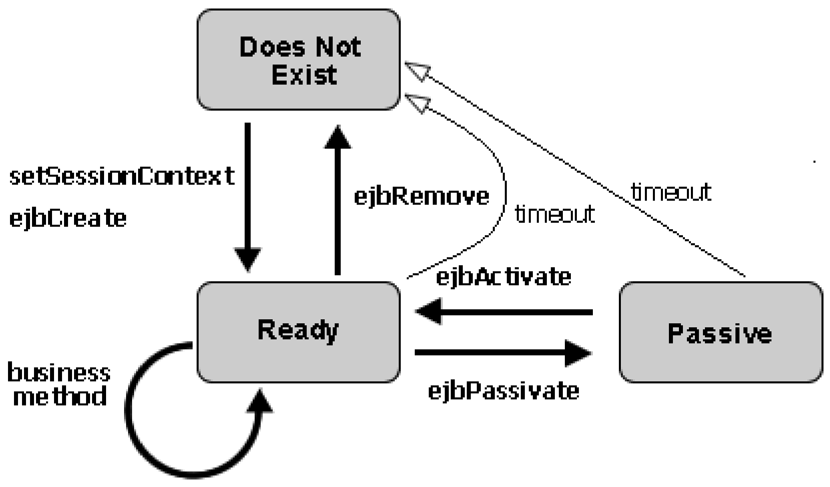
Life Cycle of a Stateful Session Bean

The following figure shows the life cycle of a stateful session bean. It has the following states:

Does not exist. In this state, the bean instance simply does not exist.

Ready state. A bean instance in the ready state is tied to particular client and engaged in a conversation.

Passive state. A bean instance in the passive state is passivated to conserve resource.



Moving from the Does Not Exist to the Ready State

When a client invokes a create method on a stateful session bean, the EJB container creates a new instance and invokes the callback method public void setSessionContext(SessionContext ctx). This method has the parameter javax.ejb.SessionContext, which contains a reference to the session context, that is, the interface to the EJB container, and can be used to self-reference the session bean object. Complete details about the javax.ejb.SessionContext can be found in your favorite J2EE documentation and the API reference at http://java.sun.com. After the callback method setSessionContext is called, the EJB container calls the callback method ejbCreate that matches the signature of the create method.

The Ready State

A stateful bean instance in the ready state is tied to a particular client for the duration of their conversation. During this conversation the instance can the execute component methods invoked by the client.

Activation and Passivation

To more optimally manage resources, the EJB container might passivate an inactive stateful session bean instance by moving it from the ready state to the passive state. When a session bean instance is passivated, its (non-transient) data is serialized and written to disk, after which the bean instance is purged from memory. Just prior to serialization, the callback method ejbPassivate is invoked. If your session bean needs to execute some custom logic prior to passivation, you can implement it using this callback method.

If after passivation a client application continues the conversation by invoking a business method, the passivated bean instance is reactivated; its data stored on disk is used to restore the bean instance state. Right after the state has been restored, the callback method ejbActivate is invoked. If your session bean needs to execute some custom logic after activation, you can implement it using this callback method. The caller (a client application or another EJB) of the session bean instance will be unaware of passivation (and reactivation) having taken place.

If a stateful session bean is set up to use the NRU (not recently used) cache-type algorithm, the session bean can time out while in passivated state. When this happens, it moves to the does not exist state; that is, it is removed. Prior to removal the EJB container will call the callback method ejbRemove. If a stateful session bean is set up to use the LRU (least recently used) algorithm, it cannot time out while in passivated state. Instead this session bean is always moved from the ready state to the passivated state when it times out.

The exact timeout can be set using the idleTimeoutSeconds attribute on the @Session annotation. The cache-type algorithm can be set using the cacheType attribute on the same annotation.

Moving from the Ready to the Does Not Exist State

When a client application invokes a remove method on the stateful session bean, it terminates the conversation and tells the EJB container to remove the instance. Just prior to deleting the instance, the EJB container will call the callback method ejbRemove. If your session bean needs to execute some custom logic prior to deletion, you can implement it using this callback method.

An inactive stateful session bean that is set up to use the NRU (not recently used) cache-type algorithm can time out, which moves it to the does not exist state, that is, it is removed. Prior to removal the EJB container will call the callback method ejbRemove. If a stateful session bean set up to use the LRU (least recently used) algorithm times out, it always moves to the passivated state, and is not removed.

The exact timeout can be set using the idleTimeoutSeconds attribute on the @Session annotation. The cache-type algorithm can be set using the cacheType attribute on the same annotation.

Stateless Session Bean

Stateless Session bean *is a business object that represents business logic* only. It doesn't have state (data).

In other words, *conversational state* between multiple method calls is not maintained by the container in case of stateless session bean.

The stateless bean objects are pooled by the EJB container to service the request on demand.

It can be accessed by one client at a time. In case of concurrent access, EJB container routes each request to different instance.

Annotations used in Stateless Session Bean

There are 3 important annotations used in stateless session bean:

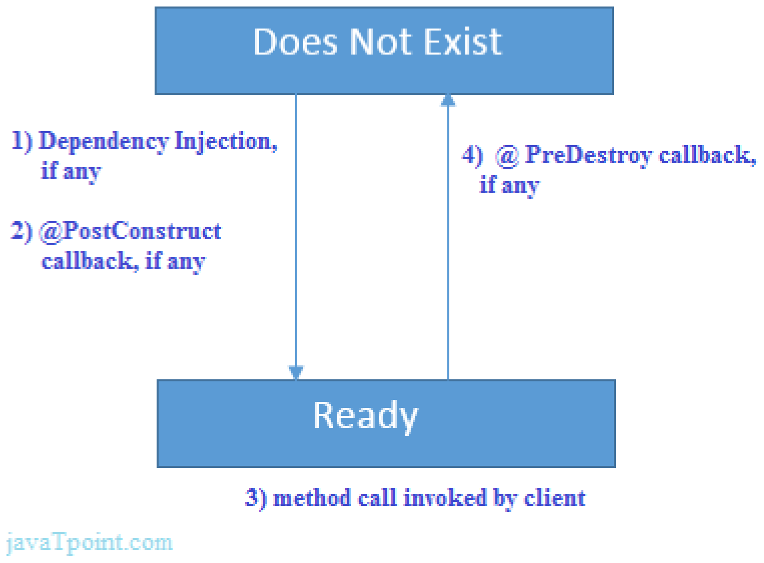
@Stateless

@PostConstruct

@PreDestroy

Life cycle of Stateless Session Bean

There is only two states of stateless session bean: does not exist and ready. It is explained by the figure given below.


Does Not Exist
1) Dependency Injection, if any
2) @PostConstruct callback, if any
4) @ PreDestroy callback, if any
Ready
3) method call invoked by client
javaTpoint.com

EJB Container creates and maintains a pool of session bean first. It injects the dependency if then calls the @PostConstruct method if any. Now actual business logic method is invoked by the client. Then, container calls @PreDestory method if any. Now bean is ready for garbage collection.

Example of Stateless Session Bean

To develop stateless bean application, we are going to use **Eclipse IDE** and **glassfish 3** server.

To create EJB application, you need to create bean component and bean client.

1) Create stateless bean component

To create the stateless bean component, you need to create a remote interface and a bean class.

File: AdderImplRemote.java

```
package com.javatpoint;
import javax.ejb.Remote;

@Remote
public interface AdderImplRemote {
int add(int a,int b);
}
```
File: AdderImpl.java

```
package com.javatpoint;
import javax.ejb.Stateless;

@Stateless(mappedName="st1")
public class AdderImpl implements AdderImplRemote {
  public int add(int a,int b){
      return a+b;
  }
}
```

Stateful Session Bean

Stateful Session bean *is a business object that represents business logic* like stateless session bean. But, it maintains state (data).

In other words, *conversational state* between multiple method calls is maintained by the container in stateful session bean.

Annotations used in Stateful Session Bean

There are 5 important annotations used in stateful session bean:

@Stateful

@PostConstruct

@PreDestroy

@PrePassivate

@PostActivate

Example of Stateful Session Bean

To develop stateful session bean application, we are going to use **Eclipse IDE** and **glassfish 3** server

1) Create stateful bean component

Let's create a remote interface and a bean class for developing stateful bean component.

File: BankRemote.java

```
package com.javatpoint;
import javax.ejb.Remote;
@Remote
public interface BankRemote {
    boolean withdraw(int amount);
    void deposit(int amount);
    int getBalance();
}
```

File: Bank.java

```
package com.javatpoint;
import javax.ejb.Stateful;
@Stateful(mappedName = "stateful123")
public class Bank implements BankRemote {
    private int amount=0;
    public boolean withdraw(int amount){
        if(amount<=this.amount){
            this.amount-=amount;
            return true;
        }else{
            return false;
        }
    }
    public void deposit(int amount){
        this.amount+=amount;
    }
    public int getBalance(){
        return amount;
    }
}
```

Entity Bean

Entity beans are objects that represents a persistence storage mechanism. The objects can be a customer, product, account etc. Each entity bean has underlying table in a relational database and each row in the table represents the instance of the bean.

Need of entity beans:

1. When it is needed by multiple clients.

2. When any action from the database side.

3. When a transaction is to be performed from the client.

4. When the enforcement of accuracy and integrity is needed.

Explain the types of Entity beans. Explain their uses.

There are two types of entity beans.:

1) Container Managed Persistence (CMP)
2) Bean Managed Persistence (BMP)

CMP can be used when:

- No code for database code is needed(by placing the specific accessibility is provided by XML file).

- The code reusability is needed.

- The speed of development by creation of the database access code by the developers.

BMP can be used when:

- Synchronizing the state of the database directly by the container.

- To provide the flexibility for the bean developer to perform the persistence operations.

- The needs of the query exceeds the capabilities of EJB QL.

- The persistent storage is a non database system or is a legacy database system that does not supports for CMP.

Explain the steps involved in developing Entity beans.

1. Select which bean is to be used – BMP or CMP entity bean

2. Create the required entity bean.

3. Author the source code for the entity bean, which involves developing the following:

a. The component interface(s)
b. The home interface(s)
c. The bean class
d. Any helper classes required by the entity bean

4. Develop the deployment descriptors for the beans.

5. Create the database tables. (If your database tables already exist, you can omit this step)

6. Establish the Object Relation mapping.

7. Compile and archive all the EJB components into a JAR file.

8. Archive the complete application into an EAR file.

9. Make the deployment of the application onto the J2EE engine.

10. Start the application.

What are the steps involved in deployment of CMP and BMP entity beans.

The purpose of deployment of entity bean is to make them available for the client applications.

Requirements before deploying:

1. Author and compile the various entity beans that are to be deployed. This includes bean class, remote interfaces, home interfaces, and the primary key class, if required.

2. Get the mapping done for the entity beans for the appropriate database tables and save the mapping information that is a deployable XML file.

3. Assemble the entity beans into a .jar or .ear file

Steps involved in deployment of entity beans:

There are three steps in deploying the entity beans.

1. Configuration :

- Specifying the number of properties for the bean.
- The usage of persistence mechanism and the needed information to do so.

2. Code generation :

- The information from the configuration phase is used by the different tools to generate the classes for the required beans.
- This process may include helper classes related to transactions, persistence, and security.
- The implementations of EJBHome and EJBObject, and the stubs and skeletons required for RMI-IIOP.

3. Installation :

- The EJB server is started and make the beans available to the clients applications.

Lifecycle of the entity beans.

The time when the entity bean starts instantiating and till it takes the time for garbage collection is the life cycle of an entity bean.

The life cycle has three states, namely

1. Does not exist state
2. Pooled state
3. Ready state

1. Does not Exist State :

- The life of a bean's instance starts with a set of files. These files include remote interface, interface, deployment descriptor, primary key and the needed files at the time of deployment.

- In this state, no instances of the bean exists.

2. Pooled State :

- At the time of starting the EJB server,with the help of the method Class.newInstance() , all the instances are created and placed in a pool. The default values of the persistent fields are set. The bean class should never contain the constructor. Therefore the default no-argument must be available in the container.

- Before placing the instances in the pool, an instance EntityContext is assigned by the container which assigns and implements the setEntityContext() method by the bean class, followed by entering into the instance pool.

- Now the bean instance is available for the client requests and become active soon after receiving the requests. The bean container provides different levels of access at every stage of the bean's life cycle.

3. The Ready State:

- Once the bean reaches this state, it can accept the requests from the client. The state of a beans instance reaches to ready state once the container assigns it to an EJB object.

- This assignment occurs in two different circumstances:

1. When the new entity bean is being created.
2. When the container is activating an entity bean.

4. End of the Life Cycle :

- The life of a bean’s instance comes to an end when the container decides to remove it from the pool and allows it to be garbage collected.

- This may happen when the container decides to reduce the number of instances from the pool in order to conserve the resources. So that the highest performance can be achieved.