Chp-33
Java Multithreading

Java is a *multithreaded programming language* which means we can develop multithreaded program using

Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.

CHAPTER

Above-mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
Threads with higher priority are more important to a program and should be allocated processor time before lowerpriority
threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing**Runnable** interface. You will need to follow three basic steps:

STEP 1:

As a first step you need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

STEP 2:

At second step you will instantiate a **Thread** object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

STEP 3

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
private Thread t;
private String threadName;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
```

```
}
}
public class TestThread {
public static void main(String args[]) {
RunnableDemo R1 = new RunnableDemo( "Thread-1");
```

```
R1.start();
RunnableDemo R2 = new RunnableDemo( "Thread-2");
R2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

STEP 1

You will need to override **run( )** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of **run()** method:

```
public void run( )
```

STEP 2

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of **start()** method:

```
void start( );
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
ThreadDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
}
public class TestThread {
public static void main(String args[]) {
ThreadDemo T1 = new ThreadDemo( "Thread-1");
T1.start();
ThreadDemo T2 = new ThreadDemo( "Thread-2");
T2.start();
}
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
```

```
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods:

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

SN Methods with Description

1 **public void start()**

Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.

2

public void run()

If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

3 **public final void setName(String name)**

Changes the name of the Thread object. There is also a getName() method for retrieving the name.

4 **public final void setPriority(int priority)**

Sets the priority of this Thread object. The possible values are between 1 and 10.

5 **public final void setDaemon(boolean on)**

A parameter of true denotes this Thread as a daemon thread.

6

public final void join(long millisec)

The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

7 **public void interrupt()**

Interrupts this thread, causing it to continue execution if it was blocked for any reason.

8

public final boolean isAlive()

Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are

static. Invoking one of the static methods performs the operation on the currently running thread.
SN Methods with Description
1
public static void yield()
Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2 **public static void sleep(long millisec)**
Causes the currently running thread to block for at least the specified number of milliseconds.
3 **public static boolean holdsLock(Object x)**
Returns true if the current thread holds the lock on the given Object.
4 **public static Thread currentThread()**
Returns a reference to the currently running thread, which is the thread that invokes this method.
5
public static void dumpStack()
Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
private String message;
public DisplayMessage(String message)
{
this.message = message;
}
public void run()
{
while(true)
{
System.out.println(message);
}
}
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
private int number;
public GuessANumber(int number)
{
this.number = number;
}
public void run()
{
int counter = 0;
int guess = 0;
do
{
guess = (int) (Math.random() * 100 + 1);
System.out.println(this.getName()
+ " guesses " + guess);
counter++;
}while(guess != number);
System.out.println("** Correct! " + this.getName()
+ " in " + counter + " guesses.**");
}
}
```
Following is the main program which makes use of above defined classes:
```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
public static void main(String [] args)
{
Runnable hello = new DisplayMessage("Hello");
Thread thread1 = new Thread(hello);
thread1.setDaemon(true);
```

```
thread1.setName("hello");
System.out.println("Starting hello thread...");
thread1.start();
Runnable bye = new DisplayMessage("Goodbye");
Thread thread2 = new Thread(bye);
thread2.setPriority(Thread.MIN_PRIORITY);
thread2.setDaemon(true);
System.out.println("Starting goodbye thread...");
thread2.start();
System.out.println("Starting thread3...");
Thread thread3 = new GuessANumber(27);
thread3.start();
try
{
thread3.join();
}catch(InterruptedException e)
{
System.out.println("Thread interrupted.");
}
System.out.println("Starting thread4...");
Thread thread4 = new GuessANumber(75);
thread4.start();
System.out.println("main() is ending...");
}
}
```
This would produce the following result. You can try this example again and again and you would get different result every time.
```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```
Major Java Multithreading Concepts:

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

- What is thread synchronization?
- Handling threads inter communication
- Handling thread deadlock
- Major thread operations

What is Thread synchronization?

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.
So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.
Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

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

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Multithreading example without Synchronization:

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces different result based on CPU availability to a thread.

```
class PrintDemo {
public void printCount(){
try {
for(int i = 5; i > 0; i--) {
System.out.println("Counter --- " + i );
}
} catch (Exception e) {
System.out.println("Thread interrupted.");
}
}
}
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
PrintDemo PD;
ThreadDemo( String name, PrintDemo pd){
threadName = name;
PD = pd;
}
public void run() {
PD.printCount();
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
```

```
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
}
public class TestThread {
public static void main(String args[]) {
PrintDemo PD = new PrintDemo();
ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
T1.start();
T2.start();
// wait for threads to end
try {
T1.join();
T2.join();
} catch( Exception e) {
System.out.println("Interrupted");
}
}
}
```

This produces different result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter --- 5
Counter --- 4
Counter --- 3
Counter --- 5
Counter --- 2
Counter --- 1
Counter --- 4
Thread Thread - 1 exiting.
Counter --- 3
Counter --- 2
Counter --- 1
Thread Thread - 2 exiting.
```

Multithreading example with Synchronization:

Here is the same example which prints counter value in sequence and every time we run it, it produces same result.

```
class PrintDemo {
public void printCount(){
try {
for(int i = 5; i > 0; i--) {
System.out.println("Counter --- " + i );
}
} catch (Exception e) {
System.out.println("Thread interrupted.");
```

```
}
}
}
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
PrintDemo PD;
ThreadDemo( String name, PrintDemo pd){
threadName = name;
PD = pd;
}
public void run() {
synchronized(PD) {
PD.printCount();
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
```

```
}
}
}
public class TestThread {
public static void main(String args[]) {
PrintDemo PD = new PrintDemo();
ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
T1.start();
T2.start();
// wait for threads to end
try {
T1.join();
T2.join();
} catch( Exception e) {
System.out.println("Interrupted");
}
}
}
```

This produces same result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter --- 5
Counter --- 4
Counter --- 3
```

```
Counter --- 2
Counter --- 1
Thread Thread - 1 exiting.
Counter --- 5
Counter --- 4
Counter --- 3
Counter --- 2
Counter --- 1
Thread Thread - 2 exiting.
```

Handling threads inter communication

If you are aware of interprocess communication then it will be easy for you to understand inter thread communication. Inter thread communication is important when you develop an application where two or more threads exchange some information.

There are simply three methods and a little trick which makes thread communication possible. First let's see all the three methods listed below:

SN Methods with Description

1 **public void wait()**

Causes the current thread to wait until another thread invokes the notify().

2 **public void notify()**

Wakes up a single thread that is waiting on this object's monitor.

3 **public void notifyAll()**

Wakes up all the threads that called wait( ) on the same object.

These methods have been implemented as **final** methods in Object, so they are available in all the classes. All three methods can be called only from within a **synchronized** context.

Example:

This examples shows how two thread can communicate using **wait()** and **notify()** method. You can create a complex system using the same concept.

```
class Chat {
boolean flag = false;
public synchronized void Question(String msg) {
if (flag) {
try {
wait();
} catch (InterruptedException e) {
```

```
e.printStackTrace();
}
}
System.out.println(msg);
flag = true;
notify();
}
public synchronized void Answer(String msg) {
if (!flag) {
try {
wait();
```

```
} catch (InterruptedException e) {
e.printStackTrace();
}
}
System.out.println(msg);
flag = false;
notify();
}
}
class T1 implements Runnable {
Chat m;
String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };
public T1(Chat m1) {
this.m = m1;
new Thread(this, "Question").start();
}
public void run() {
for (int i = 0; i < s1.length; i++) {
m.Question(s1[i]);
}
}
}
class T2 implements Runnable {
Chat m;
String[] s2 = { "Hi", "I am good, what about you?", "Great!" };
public T2(Chat m2) {
this.m = m2;
new Thread(this, "Answer").start();
}
public void run() {
for (int i = 0; i < s2.length; i++) {
m.Answer(s2[i]);
}
}
}
public class TestThread {
public static void main(String[] args) {
Chat m = new Chat();
new T1(m);
new T2(m);
}
}
```

When above program is complied and executed, it produces following result:

```
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```

Above example has been taken and then modified from [http://stackoverflow.com/questions/2170520/inter-threadcommunication-
in-java]

Handling threads deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example:

Example:

```
public class TestThread {
public static Object Lock1 = new Object();
public static Object Lock2 = new Object();
public static void main(String args[]) {
ThreadDemo1 T1 = new ThreadDemo1();
ThreadDemo2 T2 = new ThreadDemo2();
T1.start();
T2.start();
}
private static class ThreadDemo1 extends Thread {
public void run() {
synchronized (Lock1) {
System.out.println("Thread 1: Holding lock 1...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 1: Waiting for lock 2...");
synchronized (Lock2) {
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}
}
private static class ThreadDemo2 extends Thread {
public void run() {
synchronized (Lock2) {
System.out.println("Thread 2: Holding lock 2...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 2: Waiting for lock 1...");
synchronized (Lock1) {
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
}
}
```

When you compile and execute above program, you find a deadlock situation and below is the output produced by the program:

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
```

Above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL-C.

Deadlock Solution Example:

Let's change the order of the lock and run the same program to see if still both the threads waits for each other:

```
public class TestThread {
public static Object Lock1 = new Object();
public static Object Lock2 = new Object();
public static void main(String args[]) {
ThreadDemo1 T1 = new ThreadDemo1();
ThreadDemo2 T2 = new ThreadDemo2();
T1.start();
T2.start();
}
private static class ThreadDemo1 extends Thread {
public void run() {
synchronized (Lock1) {
System.out.println("Thread 1: Holding lock 1...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 1: Waiting for lock 2...");
synchronized (Lock2) {
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}
}
private static class ThreadDemo2 extends Thread {
public void run() {
synchronized (Lock1) {
System.out.println("Thread 2: Holding lock 1...");
try { Thread.sleep(10); }
catch (InterruptedException e) {}
System.out.println("Thread 2: Waiting for lock 2...");
synchronized (Lock2) {
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
}
}
```

So just changing the order of the locks prevent the program in going deadlock situation and completes with the following result:

```
Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...
```

Above example has been shown just for making you the concept clear, but its a more complex concept and you should deep dive into it before you develop your applications to deal with deadlock situations.

Major thread operatios

Core Java provides a complete control over multithreaded program. You can develop a multithreaded program which can be suspended, resumed or stopped completely based on your requirements. There are various static methods which you can use on thread objects to control their behavior. Following table lists down those methods:

SN Methods with Description

1 **public void suspend()**

This method puts a thread in suspended state and can be resumed using resume() method.

2 **public void stop()**

This method stops a thread completely.

3 **public void resume()**

This method resumes a thread which was suspended using suspend() method.

4 **public void wait()**

Causes the current thread to wait until another thread invokes the notify().

5 **public void notify()**

Wakes up a single thread that is waiting on this object's monitor.

Be aware that latest versions of Java has deprecated the usage of suspend( ), resume( ), and stop( ) methods and so you need to use available alternatives.

Example:

```
class RunnableDemo implements Runnable {
public Thread t;
private String threadName;
boolean suspended = false;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 10; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(300);
synchronized(this) {
while(suspended) {
wait();
}
}
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
```

```
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
void suspend() {
suspended = true;
}
synchronized void resume() {
suspended = false;
notify();
}
}
public class TestThread {
public static void main(String args[]) {
RunnableDemo R1 = new RunnableDemo( "Thread-1");
R1.start();
RunnableDemo R2 = new RunnableDemo( "Thread-2");
R2.start();
try {
Thread.sleep(1000);
R1.suspend();
System.out.println("Suspending First Thread");
Thread.sleep(1000);
R1.resume();
System.out.println("Resuming First Thread");
R2.suspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
R2.resume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
try {
System.out.println("Waiting for threads to finish.");
R1.t.join();
R2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Here is the output produced by the above program:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 10
Running Thread-2
Thread: Thread-2, 10
Thread: Thread-1, 9
```

```
Thread: Thread-2, 9
Thread: Thread-1, 8
Thread: Thread-2, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Suspending thread Two
Thread: Thread-1, 6
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```

Chp-34
Java Applet Basics

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other ages.

CHAPTER

- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that

contains the applet.

 paint: Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet:

The following is a simple applet named HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet
{
public void paint (Graphics g)
{
g.drawString ("Hello World",25,50);
}
}
```

These import statements bring the classes into the scope of our applet class:

 java.applet.Applet.

 java.awt.Graphics.

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet CLASS:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following:

 Get applet parameters

 Get the network location of the HTML file that contains the applet

 Get the network location of the applet class directory

 Print a status message in the browser

 Fetch an image

 Fetch an audio clip

 Play an audio clip

 Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may:

 request information about the author, version and copyright of the applet

 request a description of the parameters the applet recognizes

 initialize the applet

 destroy the applet

 start the applet's execution

 stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet:

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Below is an example that invokes the "Hello, World" applet:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<appletcode="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
```

```
message would appear here.
</applet>
<hr>
</html>
```

Based on the above examples, here is the live applet example: Applet Example.

Note: You can refer to HTML Applet Tag to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with a </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown:

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class"width="320"height="120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example:

```
<applet code="mypackage.subpackage.TestApplet.class"
width="320" height="120">
```

Getting Applet Parameters:

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.

The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of CheckerApplet.java:

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
{
int squareSize =50;// initialized to default size
public void init (){}
private void parseSquareSize (String param){}
private Color parseColor (String param){}
public void paint (Graphics g){}
}
```

Here are CheckerApplet's init() and private parseSquareSize() methods:

```
public void init ()
{
String squareSizeParam = getParameter ("squareSize");
parseSquareSize (squareSizeParam);
String colorParam = getParameter ("color");
Color fg = parseColor (colorParam);
setBackground (Color.black);
setForeground (fg);
}
private void parseSquareSize (String param)
{
if(param ==null) return;
try{
squareSize =Integer.parseInt (param);
}
catch(Exception e){
// Let default value remain
}
}
```

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.
Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.
The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet works.

Specifying Applet Parameters:

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

```
<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

Note: Parameter names are not case sensitive.

Application Conversion to Applets:

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the java program launcher) into an applet that you can embed in a web page.
Here are the specific steps for converting an application to an applet.

- Make an HTML page with the appropriate tag to load the applet code.
- Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object.the browser instantiates it for you and calls the init method.
- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
- If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
- Don't call setVisible(true). The applet is displayed automatically.

Event Handling:

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.
Inorder to react an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;
public class ExampleEventHandling extends Applet implements MouseListener{
StringBuffer strBuffer;
public void init(){
addMouseListener(this);
strBuffer =new StringBuffer();
addItem("initializing the apple ");
}
public void start(){
addItem("starting the applet ");
}
public void stop(){
addItem("stopping the applet ");
}
public void destroy(){
addItem("unloading the applet");
}
void addItem(String word){
System.out.println(word);
strBuffer.append(word);
repaint();
}
public void paint(Graphics g){
//Draw a Rectangle around the applet's display area.
g.drawRect(0,0,
getWidth()-1,
getHeight()-1);
//display the string inside the rectangle.
g.drawString(strBuffer.toString(),10,20);
}
public void mouseEntered(MouseEvent event){
}
public void mouseExited(MouseEvent event){
}
public void mousePressed(MouseEvent event){
}
public void mouseReleased(MouseEvent event){
}
```

```
publicvoid mouseClicked(MouseEventevent){
addItem("mouse clicked! ");
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>Event Handling</title>
<hr>
<appletcode="ExampleEventHandling.class" width="300" height="300">
</applet>
<hr>
</html>
```

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle "mouse clicked" will be displayed as well.

Based on the above examples, here is the live applet example: Applet Example.

Displaying Images:

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.

Following is the example showing all the steps to show images:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
private Image image;
private AppletContext context;
public void init()
```

```
{
context =this.getAppletContext();
String imageURL =this.getParameter("image");
if(imageURL ==null)
{
imageURL ="java.jpg";
}
try
{
URL url =new URL(this.getDocumentBase(), imageURL);
image = context.getImage(url);
}catch(MalformedURLException e)
{
e.printStackTrace();
// Display in browser status bar
context.showStatus("Could not load image!");
}
}
public void paint(Graphics g)
{
context.showStatus("Displaying image");
g.drawImage(image,0,0,200,84,null);
g.drawString("www.javalicense.com",35,100);
```

```
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<appletcode="ImageDemo.class"width="300"height="200">
<paramname="image"value="java.jpg">
</applet>
<hr>
</html>
```

Based on the above examples, here is the live applet example: Applet Example.

Playing Audio:

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including:

- **public void play():** Plays the audio clip one time, from the beginning.
- **public void loop():** Causes the audio clip to replay continually.
- **public void stop():** Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is the example showing all the steps to play an audio:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
private AudioClip clip;
private AppletContext context;
public void init()
{
context =this.getAppletContext();
String audioURL =this.getParameter("audio");
if(audioURL ==null)
{
audioURL ="default.au";
```

```
}
try
{
URL url =new URL(this.getDocumentBase(), audioURL);
clip = context.getAudioClip(url);
}catch(MalformedURLException e)
{
e.printStackTrace();
context.showStatus("Could not load audio file!");
}
}
public void start()
{
if(clip !=null)
{
```

```
clip.loop();
}
}
publicvoid stop()
{
if(clip !=null)
{
clip.stop();
}
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<appletcode="ImageDemo.class"width="0"height="0">
<paramname="audio"value="test.wav">
</applet>
<hr>
</html>
```

You can use your test.wav at your PC to test the above example.