**APPLETS & THREADS**

**APPLETS**

**APPLET**

* An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server.
* Applets are used to make the web site more dynamic and entertaining.

**Life cycle of an applet :**

When an applet begins, the following methods are called, in this sequence:

* Init()
* Start()
* Paint()

When an applet is terminated, the following sequence of method calls takes place:

* Stop()
* Destroy()

Let’s look more closely at these methods:

1. **init( ) :**The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.

**2. start( ) :**The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )**is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

1. **paint( ) :**The **paint( )** method is called each time an AWT-based applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

**paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )**is called.

1. **stop( ) :**The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don’t need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.
2. **destroy( ) :** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

**Sample program:**

**package** com.anna.msccs;

**import** java.applet.Applet;

**import** java.awt.Graphics;

**import** java.awt.\*;

/\*<applet code="AppletLifeCycle.class" width="350" height="150"> </applet>\*/

**public** **class** AppletLifeCycle **extends** Applet

{

**public** **void** init()

{

setBackground(Color.***PINK***);

System.***out***.println("init() called");

}

**public** **void** start(){ System.***out***.println("Start() called"); }

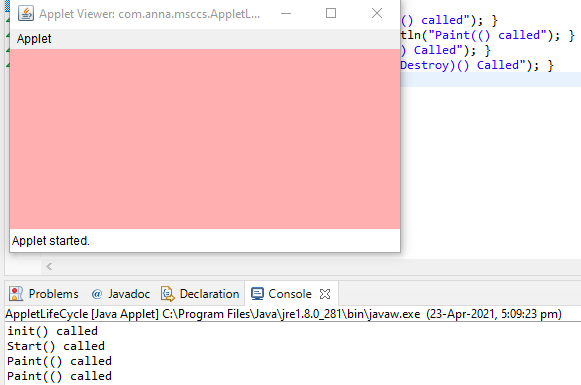
**public** **void** paint(Graphics g){ System.***out***.println("Paint(() called"); }

**public** **void** stop() { System.***out***.println("Stop() Called"); }

**public** **void** destroy() { System.***out***.println("Destroy)() Called"); }

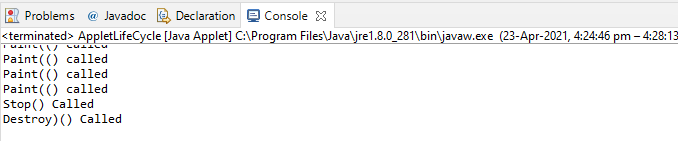
}

**Output:**



**Explanation:**

When the user opens Applet Life, an HTML page containing the applet life cycle. java in an app1etviewer, an instance for each of the applet classes is created using the no-argument constructor, and its init() method called. As a result, the message” init() called” is displayed in the command window. After execution of the code in the init() method, the program control returns to the applet viewer which then calls the applet’s start () method. As a result, the message “start() called” is displayed. Next, the appletviewer calls the applet’s paint() method as a result of which the message “paint() called” displayed.



Now when the user minimizes the applet window, the stop() method is called, and on restoring the window the start() method is called again. When the user exits the appletviewer, the destroy() method called that displays the message “destroy() called” in the command window.

**Threads**

**Thread**

* A Thread is a single sequential flow of control within a program.
* Facilityto allow multiple activites when within a single process.
* Referred as light weight process.
* Each thread has it own program counter, stack and local variables.
* Threads share the memory, heap area, files.,

**How to create thread?**

There are two ways to create a thread:

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

**Thread Class**

A **Thread class** has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the **Object** class. The **Object** class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

* Thread()
* Thread(Runnable, String name)
* Thread(Runnable target)
* Thread(ThreadGroup group, Runnable target, String name)
* Thread(ThreadGroup group, Runnable target)
* Thread(ThreadGroup group, String name)
* Thread(ThreadGroup group, Runnable target, String name, long stackSize)

### **Runnable Interface(run() method)**

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the **run()** method to perform an action for the thread.

### **start() method**

The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the **start()** method, the thread changes the state from New to Runnable. It executes the **run() method** when the thread gets the correct time to execute it.

Let's take an example to understand how we can create a [Java](https://www.javatpoint.com/java-tutorial) thread by extending the Thread class:

**package** com.anna.msccs;

//Implementing runnable interface by extending Thread class

**public** **class** ThreadExample1 **extends** Thread {

// run() method to perform action for thread.

**public** **void** run()

{

**int** a= 10;

**int** b=10;

**int** result = a+b;

System.***out***.println("Thread started running..");

System.***out***.println("Sum of two numbers is: "+ result);

}

**public** **static** **void** main( String args[] )

{

// Creating instance of the class extend Thread class

ThreadExample1 t1 = **new** ThreadExample1();

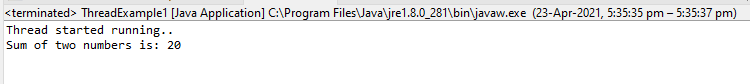
//calling start method to execute the run() method of the Thread class

t1.start();

}

}

**Output:**



**Creating thread by implementing the runnable interface**

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

Let's takes an example to understand how we can create, start and run the thread using the runnable interface.

**package** com.anna.msccs;

**class** NewThread **implements** Runnable {

String name;

Thread thread;

NewThread (String name){

**this**.name = name;

thread = **new** Thread(**this**, name);

System.***out***.println( "A New thread: " + thread+ "is created\n" );

thread.start();

}

**public** **void** run() {

**try** {

**for**(**int** j = 5; j > 0; j--) {

System.***out***.println(name + ": " + j);

Thread.*sleep*(1000);

}

}**catch** (InterruptedException e) {

System.***out***.println(name + " thread Interrupted");

}

System.***out***.println(name + " thread exiting.");

}

}

**class** ThreadExample2 {

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

**new** NewThread("1st");

**new** NewThread("2nd");

**new** NewThread("3rd");

**try** {

Thread.*sleep*(8000);

} **catch** (InterruptedException excetion) {

System.***out***.println("Inturruption occurs in Main Thread");

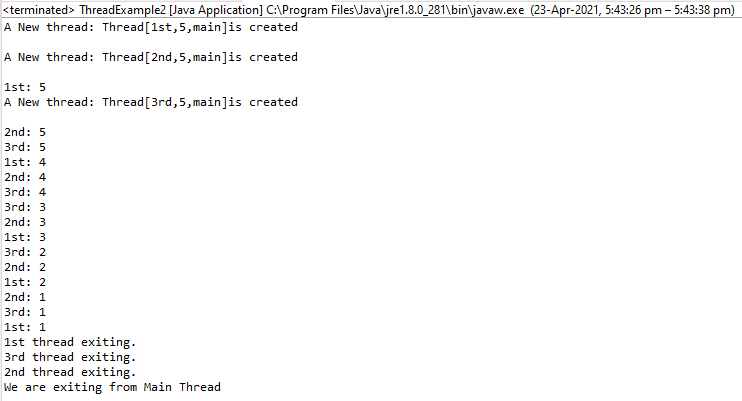
}

System.***out***.println("We are exiting from Main Thread");

}

}

**Output:**



# Multithreading

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a 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. Multi-threading 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.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

# Thread Synchronization

# Multithreading Example without Synchronization:

**package** com.anna.msccs;

**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** SyncExample {

**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");

}

}

# }

# Output:

# 

## **Multithreading Example with Synchronization**

**package** com.anna.msccs;

**class** PrintDemo1 {

**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** ThreadDemo1 **extends** Thread {

**private** Thread t;

**private** String threadName;

PrintDemo PD;

ThreadDemo1( String name, PrintDemo1 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** SyncExample1 {

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

PrintDemo1 PD = **new** PrintDemo1();

ThreadDemo1 T1 = **new** ThreadDemo1( "Thread - 1 ", PD );

ThreadDemo1 T2 = **new** ThreadDemo1( "Thread - 2 ", PD );

T1.start();

T2.start();

// wait for threads to end

**try** {

T1.join();

T2.join();

} **catch** ( Exception e) {

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

}

}

}

**Output:**

