6SENG006W Concurrent Programming

Week 7

States & “Life-Cycle” of a Java Thread

States & “Life-Cycle” of a Java Thread

Aim of the lecture is to describe the *life-cycle* of a Java thread by describing:

- The *states* of a Java thread:
  - **`NEW`**
  - **`RUNNABLE`**
  - **`BLOCKED`**
  - **`WAITING`**
  - **`TIMED_WAITING`**
  - **`TERMINATED`**
- Thread *“life-cycle”* — *when* & *how* threads *transition* between these states, e.g. the **`RUNNABLE`** & the **non-runnable** states.
- Selected features of the `Thread` class, e.g. methods & exceptions.
- An FSP process that models Java *thread states* & *transitions*.
- Why certain `Thread` class methods have been **“deprecated”**.

PART I

Thread States & Life-Cycle

Java Thread States & Life-Cycle

Fig. 7.1 shows the states that a Java thread can be in during its life-cycle & what conditions & method calls cause a transition between states.

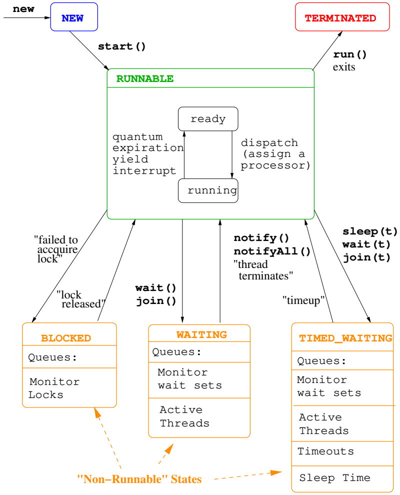

Figure : 7.1 Java Thread States

Description of Thread States

A thread can *only be in one state at a given point in time*.

The following values of the `Thread.State` (enumeration) type represent the *"life-cycle"* states of a thread:

State	Description
`NEW`	a *created* thread, but which has **not** yet started executing.
`RUNNABLE`	a *"runnable"* thread, i.e. is *"executing"*.
`BLOCKED`	a thread **blocked waiting** for a *"synchronisation lock"*.
`WAITING`	a waiting thread, i.e. waiting for *"something"* to happen.
`TIMED_WAITING`	a waiting thread with a *specified waiting time*.
`TERMINATED`	a **terminated** thread, i.e. completed execution.

They are the possible *virtual machine states* which do not reflect any operating system thread states.

We shall now describe these thread states & how threads switch between them in more detail.

The **NEW** Thread State

Given the following definition of `MyThreadClass`:

```
class MyThreadClass extends Thread
{
   public void run() { ... }
}
```

The following statement *creates a new thread* but **does not start it**, thereby leaving the thread in the **NEW** Thread state.

```
Thread myThread = new MyThreadClass() ;
```

When a thread is in the **NEW** Thread state, it is merely an empty `Thread` object.

No system resources have been allocated for it yet.

When a thread is in this state, you can only start the thread by calling the `start()` method.

Calling any other method besides `start()` when a thread is in this state makes no sense & causes an `IllegalThreadStateException`.

The **RUNNABLE** Thread State

Consider the code:

```
Thread myThread = new MyThreadClass() ;
myThread.start() ;
```

The effect of calling the `start()` method is that the JVM:

1. *creates the system resources* necessary to run the thread,
2. *schedules the thread*, so its ready to run, i.e. begin executing,
3. *calls* the thread's `run()` method.

At this point the thread is in the **RUNNABLE** state.

This state is called **RUNNABLE** rather than "**RUNNING**" because the thread **may not actually be running** when it is in this state.

A thread in the **RUNNABLE** state is executing in the JVM but it may be waiting for other resources from the operating system such as processor, or waiting for I/O to complete.

Note that you **do not** start a thread executing by calling a thread's `run()` method, this is **never called** by a user program.

A Running Thread

Many computers have a single processor making it impossible to run all the threads that are in the **`RUNNABLE`** state at the same time.

So, the Java run-time system must implement a *scheduling scheme* that shares the processor between all the threads in the **`RUNNABLE`** state.

(This is covered in the *Thread Scheduling* lecture.)

However, it is simpler & safer to assume that all threads that are in the **`RUNNABLE`** state are concurrently executing.

By making this assumption programmers are less likely to make incorrect assumptions about what threads are executing & in what order, etc.

When a thread is actually being executed (i.e. running) it is in the **`RUNNABLE`** state & it is the current thread, i.e. its `run()` method is being executed (sequentially).

The "Not Runnable" States

An important aspect of the *thread life-cycle* is how threads switch between the **`RUNNABLE`** state & the **"not runnable"** states.

The following states are the **not runnable** states:

- **`BLOCKED`**,
- **`WAITING`** &
- **`TIMED_WAITING`**.

If a thread is in one of these states it is not available to be scheduled, i.e. it cannot be executed because it is waiting for something to happen.

The *not runnable* states are (mainly) related to how a thread interacts with a monitor, & in particular the *lock* associated with the monitor.

Thus when a thread "is waiting for something to happen", the "something" is usually a thread's attempt to perform some action on a monitor &/or its *lock*.

(We shall cover monitors & locks in the *Thread Synchronization & Java Monitors* lecture.)

We shall now examine how & why a thread switches between these states & the states themselves in more detail.

Examples of transitions between the **RUNNABLE** state & the "**Not Runnable**" states

The following diagram illustrates some of the the possible state transitions for a thread between the **RUNNABLE** state & the three "**not runnable**" states.

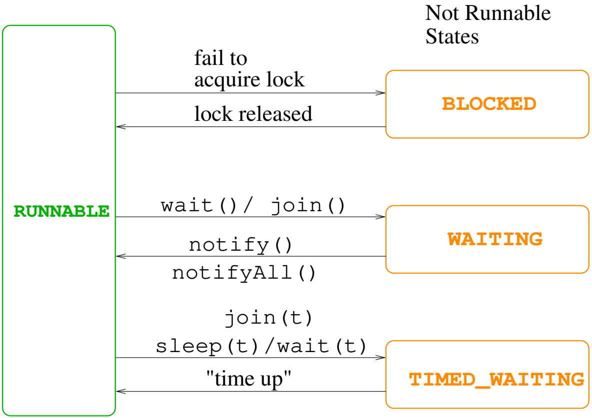

Figure : 7.2 Thread State Transitions: Runnable & Non-Runnable

Explanation of the Transitions

For each of the different transitions that a thread can make from the **RUNNABLE** state to one of the **not runnable** states, there is an *opposite transition* that returns the thread to the **RUNNABLE** state.

In effect, for each type of “entry route” into a **not runnable** state, there is a specific & distinct “exit route” that returns the thread to the **RUNNABLE** state.

That is, each *“exit route”* only works for its corresponding *“entry route”*.

Transitions between: **RUNNABLE** & **Not Runnable** states

Some examples of transitions that a thread can make between the **RUNNABLE** state & one of the **not runnable** states:

- **RUNNABLE** $\rightleftarrows$ **BLOCKED**
  - the thread fails to *acquire* a monitor lock.
  - the monitor lock is *released* by some other thread & the thread can now attempt to acquire it.
- **RUNNABLE** $\rightleftarrows$ **WAITING**
  - The thread calls the `wait()` method, (since it requires the monitor to be in a different state before it can proceed).
  - Another thread has changed the state of the resource & calls either the `notify()` or `notifyAll()` methods.
- **RUNNABLE** $\rightleftarrows$ **TIMED_WAITING**
  - The `sleep(t)` method is called on the thread.
  - The specified number of milliseconds must have elapsed before the thread becomes **RUNNABLE** again.

Example: **BLOCKED** Thread States

Assume a Java multi-threaded program consists of 10 threads `T1`, ..., `T10`, & uses 5 synchronisation locks: `Lock 1`, `Lock 2`, ..., `Lock 5`.

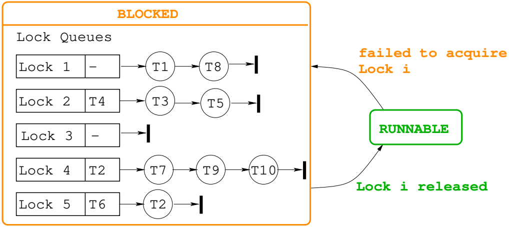

Figure : 7.3 Example **BLOCKED** Thread States

Figure 7.3 is an example of a possible state of the threads in the program:

`Lock 1` – just unlocked, before `T1` & `T8` moved back to **RUNNABLE**.

`Lock 2`, `Lock 4` & `Lock 5` – locked & several threads blocked waiting for them to be released.

`Lock 3` – unlocked & no threads are blocked on it.

The **`BLOCKED`** Thread State

Thread state for a thread blocked waiting for a monitor lock.

This means that another thread currently has *"acquired"* or *"holds"* the lock & therefore the thread cannot proceed & is blocked.

A thread in the **`BLOCKED`** state is waiting for a monitor lock so that it can either:

- *enter* a `synchronized` block or method; e.g.

```
// thread attempts to enter block
synchronized ( object_lock )
{
          // critical section
}
```

- or *reenter* a `synchronized` block or method after calling `Object.wait`, e.g.

```
public synchronized void useSharedResource()
{
    wait() ;
   // thread attempts to reenter method
}
```

The **`WAITING`** Thread State

Thread state for a waiting thread.

A thread in the **`WAITING`** state is waiting for another thread to perform a particular action.

A thread is in the **`WAITING`** state due to calling one of the following methods:

- `Object.wait` with *no timeout*.
  A thread that has called `Object.wait()` on an object & is waiting for another thread to call `Object.notify()` or `Object.notifyAll()` on that object.

- `Thread.join` with *no timeout*.
  A thread that has called `Thread.join()` is waiting for a specified thread to terminate.

The `TIMED_WAITING` Thread State

Thread state for a waiting thread with a specified waiting time.

A thread is in the `TIMED_WAITING` state due to calling one of the following methods with a specified positive waiting time:

- `Thread.sleep`
  A thread calls `Thread.sleep(t)` & is waiting for the timeout `t`.
- `Object.wait` with timeout
  A thread calls `Object.wait(t)` on an object & is waiting for:
  - the timeout `t` to have elapsed
  - another thread to call `Object.notify()` or `Object.notifyAll()` on that object.
- `Thread.join` with timeout;
  A thread calls `Thread.join(t)` & is waiting for:
  - the timeout `t` to have elapsed; or
  - for the specified thread to terminate.

Example: Thread in the **TIMED_WAITING** State

Moving a thread from the **RUNNABLE** state to the **TIMED_WAITING** state:

In `main()` method:

```
Thread myThread = new MyThreadClass() ;
myThread.start() ;
```

In `run()` method of `myThread`:

```
try   {  myThread.sleep(10000) ;  }
catch (InterruptedaException e){ }
```

The result of executing "`myThread.sleep(10000)`" is that:

- it puts `myThread` to sleep for 10 seconds (10,000 milliseconds); &
- `myThread` is moved form the **RUNNABLE** to **TIMED_WAITING** state.

During those 10 seconds, even if the processor became available `myThread` would **not run**.

After the 10 seconds are up, `myThread` returns to the **RUNNABLE** state again & now if the processor became available `myThread` would run.

The **TERMINATED** Thread State

A thread can only *terminate* when its `run()` method *exits normally*.

For example, a thread with the following `run()` method will *terminate* after the loop & the `run()` method completes.

```
public void run()
{
   int i = 0 ;
   while ( i < 100 )
   {
      i++ ;
      System.out.println("i = " + i) ;
   }
   // i = 100
}
// run() exits & Thread terminates
```

The `isAlive()` & `getState()` Methods

The `Thread` class includes a method called `isAlive()`.

`isAlive()` returns:

- `true` – if the thread has been *started* but **not terminated**, i.e. it is either in the **RUNNABLE** or one of the **not runnable** states.
- `false` – if it is either in the **NEW** or **TERMINATED** states.

WARNING: you **cannot** differentiate between a thread in:

- the **NEW** state & one in the **TERMINATED** state.
- the **RUNNABLE** state & one that is in a **not runnable** states.

However, it is possible to determine the state of a thread by calling the `Thread.getState()` method on the thread.

The `getState()` method returns the state of the thread, that is a value of type `Thread.State`, i.e. **NEW**, **RUNNABLE**, etc.

Thread Exceptions

The run-time system *throws an exception* called: `IllegalThreadStateException`.

This is generated when a *method* is called on a thread whose state does not allow for that method call.

For example, `IllegalThreadStateException` is thrown when you call `start()` on a thread that has already been started.

Note when you call a thread method that *can throw an exception* you must either:

- *catch & handle the exception*; or
- *declare* that the *calling method throws the uncaught exception*.

(For more details see **ORACLE**'s online *Java Tutorial*.)

PART II

Modelling Thread States using an FSP Process

Modelling Thread States using an FSP Process

The following FSP process `JavaThreadStates` models the possible Java *thread states* & *transitions* between states.

It is defined as a collection of *local FSP processes*, where each Java thread state is represented by at least one FSP process.

It uses *indexed* processes, e.g. "`[ t : TIME]`", to model the *waiting time* counting down for a thread while it is in the **TIMED_WAITING** state.

This occurs when one of Java's *timed commands*: `wait(t)`, `join(t)` or `sleep(t)`, is used.

Java States FSP Process: `JavaThreadStates`

The **NEW** & **TERMINATED** states are modelled by the following *local* FSP processes:

```
const MAX_TIME = 3

range TIME = 0..MAX_TIME

JavaThreadStates = ( new -> NEW ) ,

NEW = ( start -> RUNNABLE ) ,

TERMINATED = ( terminated -> END ) ,
```

FSP Process `JavaThreadStates`: **RUNNABLE** State

The **RUNNABLE** state (including “**RUNNING**”) are modelled by the following two *local* FSP processes:

```
RUNNABLE = (   ready    -> RUNNABLE
             | dispatch -> RUNNING   ) ,

RUNNING
 = (  execute_run_method   -> RUNNING
    | quantum_expires      -> RUNNABLE
    | yield                -> RUNNABLE
    | interrupt            -> RUNNABLE
    | failed_acquire_lock  -> BLOCKED
    | wait                 -> WAITING_wait
    | join                 -> WAITING_join
    | wait_t[ t : TIME ]   -> TIMED_WAITING_wait[ t ]
    | join_t[ t : TIME ]   -> TIMED_WAITING_join[ t ]
    | sleep_t[ t : TIME ]  -> TIMED_WAITING_sleep[ t ]
    | exit_run_method      -> TERMINATED
   ) ,
```

The Non-Runnable States: **BLOCKED** & **WAITING**

```
BLOCKED = (   blocked          -> BLOCKED
            | lock_released    -> RUNNABLE    ) ,
```

The **WAITING** state is represented by two states: **WAITING**`_wait` & **WAITING**`_join`.

This is because we need to be able to match up its mode of *"entry into"* & *"exit from"* the **WAITING** state.

```
WAITING_wait
   = (   wait_waiting   -> WAITING_wait
       | notify         -> RUNNABLE
       | notifyAll      -> RUNNABLE
       | interrupt      -> RUNNABLE     ) ,

WAITING_join
   = (  join_waiting               -> WAITING_join
      | join_thread_terminates     -> RUNNABLE
      | interrupt                  -> RUNNABLE     ) ,
```

The Non-Runnable State: TIMED_WAITING (1)

The **TIMED_WAITING** state is represented by three states: **TIMED_WAITING**_wait, **TIMED_WAITING**_join & **TIMED_WAITING**_sleep.

Again this is because we need to be able to match up its mode of *"entry into"* & *"exit from"* the **TIMED_WAITING** state.

```
TIMED_WAITING_wait[ t : TIME ]
= (  when (t > 0) timed_waiting[ t ] -> TIMED_WAITING_wait[t - 1]
   | notify                          -> RUNNABLE
   | notifyAll                       -> RUNNABLE
   | when (t == 0) wait_time_up      -> RUNNABLE
   | interrupt                       -> RUNNABLE
  ) ,

TIMED_WAITING_join[ t : TIME ]
= (  when (t > 0) timed_joining[ t ]  -> TIMED_WAITING_join[t - 1]
   | join_thread_terminates           -> RUNNABLE
   | when (t == 0) join_time_up       -> RUNNABLE
   | interrupt                        -> RUNNABLE
  ) ,
```

The Non-Runnable State: **TIMED_WAITING** (2)

The **TIMED_WAITING**_sleep state is represented by:

```
TIMED_WAITING_sleep[ t : TIME ]
 = (  when ( t > 0)  sleeping[ t ]  -> TIMED_WAITING_sleep[t - 1]
    | when ( t == 0) sleep_time_up  -> RUNNABLE
    | interrupt                     -> RUNNABLE
   ) .
```

Extending the JavaThreadStates FSP Process

See Tutorial 5, for exercises that involve extending the `JavaThreadStates` FSP process, e.g.

- Add the `IllegalThreadStateException` action.
- Run it in parallel with FSP models of simple Java threads.

PART III

Deprecated `Thread` Class Methods

Deprecated `Thread` Class Methods

From the JDK 1.2 version of Java on wards several methods which formed part of the `Thread` class in earlier versions (JDK 1.0, 1.1.x) have been **"deprecated"**, i.e. no longer part of the `Thread` class.

Each of these methods *affected the life-cycle of a thread*, i.e. the possible transitions between states & the states it could be in.

The **deprecated** methods are:

- `stop()` – *terminated* the execution of a thread.
- `suspend()` – *suspended* the execution of a thread by moving it from the **RUNNABLE** state to the **not runnable** state.
- `resume()` – *resumed* a suspended thread by returning it back to the **RUNNABLE** state, i.e. reversed the `suspend()` method.
- `destroy()` – *suspended* the execution of a thread **permanently** by moving it from the **RUNNABLE** state to the **not runnable** state.

Investigation of Why They Were Deprecated

To understand why these methods have been **deprecated** we shall look at:

- How the **`stop()`** method *worked* & why it was *deprecated*.
- Examples of how to *stop a thread* without using the **`stop()`** method.
- Why the other methods were *deprecated*.

Example of using the stop() Method

Consider the following example of how the **stop()** method was usually used:

```
public void run()              // Master thread
{
   // create & start the worker thread

   Thread  worker = new WorkerThread() ;

   worker.start() ;

   try {
         // put self to sleep for 10 seconds

         Thread.currentThread().sleep(10000) ;
   }
   catch (InterruptedException e){ }

   worker.stop() ;              // kill the worker thread
}
```

Stopping a Thread Using `stop()`

How a thread was **stopped** in early versions of Java, e.g. JDK 1.1, using `stop()`.

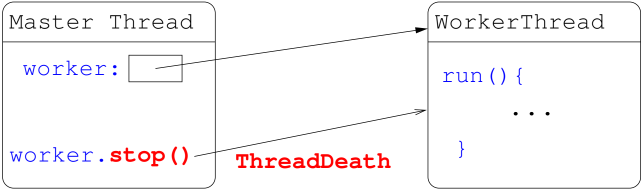

Figure : 7.4 Stopping a Thread using `stop()` in JDK 1.1

How The `stop()` Method Worked

- The `Master` thread *creates* & *starts* the `worker` thread.
- Then puts itself (i.e. the *"current thread"*) to *sleep* for 10 seconds.
- "`currentThread()`" is a `static Thread` class method & returns a *reference* to the *currently executing thread* object.
- After the 10 seconds have elapsed, the `Master` thread *wakes up*.
- It then *calls* the **`stop()`** method which *throws* a **`ThreadDeath`** object at the `worker` thread to **kill it**.
- When a thread is **killed** in this manner it dies *asynchronously*, i.e. the thread will die when it *actually receives* the **`ThreadDeath`** exception.
- An important fact about the **`ThreadDeath`** object is that it **cannot** be caught by the usual `try{..} catch(E e){..}` block.

Why the `stop()` Method was **Deprecated**

A thread could be killed at *any time* by having its `stop()` method called.

A consequence is that the thread has **no warning** that it is going to be *killed*.

In particular, a thread could be in the middle of performing some *critical operation* when it is killed.

If this occurred when a critical operation had **not been completed** then whatever *data* & *resources* it was using at the time it was killed, *could be* left in an **inconsistent state**.

Such *objects* are said to be **"damaged"**.

When threads operate on **damaged** objects, *arbitrary behaviour* can result that may be subtle & difficult to detect, or it may be obvious.

The **corruption** can manifest itself at any time after the actual damage occurs, even hours or days in the future.

So because of the way `stop()` – **killed threads** by using `ThreadDeath`, the user has **no warning** that his program may be **corrupted**.

Obviously, this is **"inherently unsafe"** & why `stop()` was **deprecated**.

Example of Problems Caused by the `stop()` Method

The most important example of the problems caused by the use of the **`stop()`** method is when a thread is *"accessing a protected shared resource"*, i.e. executing a *critical section* of code that uses the resource.

This is achieved by using a *"monitor"* to protect the shared resource. (Monitors will be covered in a future lecture.)

For a thread to use/access the shared resource it must:

1. **lock** the monitor;
2. *execute the critical section* accessing the resource;
3. **unlock** the monitor.

Now if the thread is **killed** by **`stop()`** when it is *"inside"* the monitor, i.e. doing (2), then the monitor was automatically **unlocked**.

This happens *irrespective* of the state of the *shared resource*, so the shared resource could be left in an **inconsistent state**, i.e. it is a **damaged** object.

So any other thread that subsequently accesses the shared resource will be using a **damaged** object.

How to Stop a Thread Without Using `stop()`

From JDK 1.2 inwards a Java program **cannot stop a thread** like it stops an applet, by calling the `Thread.`**`stop()`** method.

Rather, a thread should *"arrange"* for its *own death* by having a `run()` method that **terminates "naturally"**.

Most uses of **`stop()`** should be replaced by code that simply:

- Modifies a *"terminate flag"* variable, usually it is just a boolean.
- This is used to *indicate* that the target thread should **stop running & terminate**, i.e. **exit** its `run()` method.
- The target thread should *check this variable regularly*.
- If the variable is *true* then it should *return* from its `run()` method in an orderly fashion, i.e. *terminate in a controlled way*.

Problem with the "Terminate Flag" Variable Approach

This all seems very straightforward, but there is a **"hidden trap"** with this approach that must be avoided if it is to work properly.

The key issue is that the terminating thread (normally) *only reads* the variable.

This means that the Java compiler is *very likely* to *"optimise"* the code & the terminating thread would only read the variable **once** at the *start of* the `run()` method.

So *any changes made to its value* by another thread after the terminating thread has started executing its `run()` method, would **NOT be picked up by the thread**.

The effect of this is that even though another thread has made the *"terminate flag"* variable false, the `run()` method would **NOT TERMINATE**.

The key fact is that:

> *the approach **cannot be guaranteed** to always work across every OS & every Java system.*

So as Clint Eastwood would say *"Do you feel lucky punk?"*

Ensuring the “Terminate Flag” Variable Works

The key is that when one thread makes a **stop-request** to another thread it is *communicated* to that thread & the thread terminates.

For this to happen *successfully*, the setting of the *“terminate flag”* variable to false **MUST** be *“observed”* by the terminating thread.

To ensure that this always happens the *variable* **MUST** either:

- be declared as “`volatile`”
  (The existence of `volatile` is Java’s acknowledgement the problem exists.)
- or access to the variable must be via a “`synchronized`” method.

Either of these approaches *ensures* that the *“freshest”* value of the variable is read whenever the thread references it.

This is because to access it the terminating thread must *acquire its lock* & because of how the JVM works (see previous lecture) it guarantees that it gets the *“freshest”* value of the variable.

Note: use either the `volatile` or `synchronized` approach but **not both**.

Stopping a Thread Without Using `stop()`

Fig. 7.5 illustrates how to **stop** a thread without using `stop()`, but using **`volatile`** or **`synchronized`**.

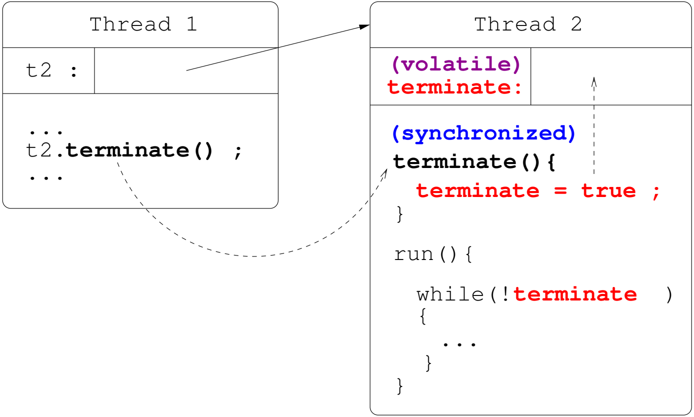

Figure : 7.5 Stopping a Thread Without using the `stop()` Method (JDK 1.2+)

Example 1: Stopping an Applet Thread using **stop()**

For example, suppose your (JDK 1.1) applet contains the following `start()`, `stop()` & `run()` methods:

```
private final int ONESECOND = 1000 ;
private Thread oneSecondTimer ;

public void start() {
    oneSecondTimer = new Thread( this );
    oneSecondTimer.start();
}

public void stop() {        // Applet ``stop'' method
  oneSecondTimer.stop() ;   // Thread ``stop'' method
}

public void run() {
    Thread thisThread = Thread.currentThread();

    while ( true )
    {
        try   {  thisThread.sleep( ONESECOND ) ; }
        catch (InterruptedException e){ }
        repaint();
    }
}
```

Solution using `volatile`

You can avoid the use of `Thread`.**`stop()`** by keeping the same applet `start()` method, but by using **`volatile`** & replacing the applet's `stop()` & `run()` methods as follows:

```
private volatile Thread oneSecondTimer ;

public void stop()          // Applet's ``stop''
{
    oneSecondTimer = null ;
}

public void run()
{
    Thread thisThread = Thread.currentThread() ;

    while ( oneSecondTimer == thisThread )
    {
        try {
                thisThread.sleep( ONESECOND );
          }
        catch (InterruptedException e){}
        repaint();
    }
}
```

How `volatile` Solution Works

- Declare in the `Thread` reference variable **`oneSecondTimer`** as **`volatile`**.
- Initially **`oneSecondTimer`** & **`thisThread`** will *both contain a reference* to the thread executing the `run()` method.
- So the `while`-loop continues executing until the Applet's `stop()` method is called & sets **`oneSecondTimer`** to **`null`**.
- The use of **`volatile`** in the declaration of **`oneSecondTimer`** ensures that *each time it is referenced* in the `while` loop condition the *most recent value is used*.
- At some point after **`oneSecondTimer`** equals **`null`**, it will result in the `while` loop condition being **false**, & hence the loop & `run()` method terminating.
- Consequently, the thread has **terminate**.

Solution using **synchronized** Methods

Modify the applet's `stop()` & `run()` methods, & add a termination variable & **synchronized** set & get methods:

```
private Thread oneSecondTimer;
private boolean terminateThread = false;

public void stop() {
    terminate() ;
}

private synchronized void terminate() {
    terminateThread = true ;
}

private synchronized boolean doNotTerminate() {
    return ( ! terminateThread ) ;
}

public void run() {
   Thread thisThread = Thread.currentThread() ;
   while ( doNotTerminate() ) {
      try   { thisThread.sleep( ONESECOND ) ; }
      catch (InterruptedException e){} ;
      repaint() ;
    }
}
```

Example 2: Lecture 5's `Clock` Applet's Termination

Consider the `Clock` applet thread again & how its termination was arranged.

```
private Thread clockThread = null ;

public void start(){                   // Applet's "start()"
  // create & start the clockThread
}
public void stop(){                  // Applet's "stop()"
      clockThread = null ;
}
public void run(){
   while ( clockThread != null ){
         // redraw clock & sleep for 1 second
   }
}
```

The exit condition for this `run()` method is basically the exit condition for the `while` loop because there is no code after the `while` loop.

The loop exit condition indicates that the loop will exit when **`clockThread`** is equal to **`null`**.

Setting `clockThread` to `null`

This happens when you leave the web page, then the application in which the applet is running calls the applet's `stop()` method.

This method sets **`clockThread`** to **`null`**, indicating the `while` loop in the `run()` method should terminate.

However, as we have seen above, this does **not always guarantee** that `clockThread`'s `run()` method will *"notice"* this change in value of the `clockThread` variable.

Therefore, to guarantee that this approach works it is necessary to apply one of the two methods outlined above.

For example, if we adopt the **`volatile`** method, we just need to make one modification to the above code:

```
private volatile Thread clockThread = null ;
```

Now the clock thread will be terminated & if the web page is revisited, the `start()` method is called again & a new clock thread is created & started.

*So ensures that any previously created clock threads will terminate, thus stopping multiple ones from being created & started, **but not terminated.***

Why `suspend()`, `resume()` & `destroy()` were deprecated

- `suspend()` is **inherently deadlock-prone**.
  If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, **no thread can access this resource** until the target thread is resumed.
  If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume()`, **deadlock results**.
  Such deadlocks typically manifest themselves as *"frozen" processes*.

- `resume()` has also been **deprecated** because it exists solely for use with `suspend()`.

- `destroy()` has never been implemented.
  If it were implemented, it would be **deadlock-prone** in the manner of `suspend`.
  It is roughly equivalent to `suspend()` without the possibility of a subsequent `resume()`.