

Module IV Multithreading

Dr. Zahid Ansari

Multi-Tasking

- Two kinds of multi-tasking:
 - 1) process-based multi-tasking
 - 2) thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.
- Processes are heavyweight tasks:
 - 1) that require their own address space
 - 2) inter-process communication is expensive and limited
 - 3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:
 - 1) they share the same address space
 - 2) they cooperatively share the same process
 - 3) inter-thread communication is inexpensive
 - 4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:
 - 1) the transmission rate of data over a network is much slower than the rate at which the computer can process it
 - 2) local file system resources can be read and written at a much slower rate than can be processed by the CPU
 - 3) of course, user input is much slower than the computer

Threads: Model

- Thread exist in several states:
 - 1) ready to run
 - 2) running
 - 3) a running thread can be suspended
 - 4) a suspended thread can be resumed
 - 5) a thread can be blocked when waiting for a resource
 - 6) a thread can be terminated, Once terminated, a thread cannot be resumed.

Threads: Priorities

- Every thread is assigned priority – an integer number to decide when to switch from one running thread to the next (context-switching).
- Rules for context switching:
 - 1) a thread can voluntarily relinquish control (sleeping, blocking on I/O, etc.), then the highest-priority ready to run thread is given the CPU.
 - 2) a thread can be preempted by a higher-priority thread – a lower-priority thread is suspended
- When two equal-priority threads are competing for CPU time, which one is chosen depends on the operating system.

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program. How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:
 - 1) classes can define so-called synchronized methods
 - 2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
 - 3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Class

- To create a new thread a program will:
 - 1) extend the Thread class, or
 - 2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

<code>getName</code>	obtain a thread's name
<code>getPriority</code>	obtain a thread's priority
<code>isAlive</code>	determine if a thread is still running
<code>join</code>	wait for a thread to terminate
<code>run</code>	entry-point for a thread
<code>sleep</code>	suspend a thread for a period of time
<code>start</code>	start a thread by calling its run method

The Main Thread

- The main thread is a thread that begins as soon as a program starts.
- The main thread:
 - 1) is invoked automatically
 - 2) is the first to start and the last to finish
 - 3) is the thread from which other “child” threads will be spawned
- It can be obtained through the public static `Thread currentThread()` method of `Thread`.

Example: Main Thread

```
class CurrentThreadDemo {
    public static void main(String args[]) {
        // The main thread is obtained, displayed, its name changed & redisplayed:
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        // the sleep method may throw InterruptedException if some other thread
        // wanted to interrupt:
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Example: Thread Methods

- Thread methods used by the example:
 - 1) static void sleep(long milliseconds) throws InterruptedException
Causes the thread from which it is executed to suspend execution for the specified number of milliseconds.
 - 2) final String getName()
Allows to obtain the name of the current thread.
 - 3) final void setName(String threadName)
Sets the name of the current thread.

Creating a Thread

- Two methods to create a new thread:
 - 1) by implementing the Runnable interface
 - 2) by extending the Thread class

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:
 - 1) create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

```
public void run()
```
 - 2) instantiate a Thread object within that class, a possible constructor is:

```
Thread(Runnable threadOb, String threadName)
```
 - 3) call the start method on this object (start calls run):

```
void start()
```

Example: New Thread 1

```
class NewThread implements Runnable {
    Thread t;
    // Creating and starting a new thread. Passing this to the Thread constructor –
    // the new thread will call this object's run method:
    NewThread() {
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start();
    }
    //This is the entry point for the newly created thread, a five-iterations loop with
    a half-sec pause
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

Example: New Thread 2

```
class ThreadDemo {
    public static void main(String args[]) {
        // A new thread is created as an object NewThread:
        new NewThread();
        // After calling the NewThread start method, control returns here.
        // Both threads (new and main) continue concurrently.
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```


New Thread: Extend Thread

- The second way to create a new thread:
 - 1) create a new class that extends Thread
 - 2) create an instance of that classThread provides both run and start methods:
 - 1) the extending class must override run
 - 2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:

```
class NewThread extends Thread {
    // Create a new thread by calling the Thread's constructor and start():
    NewThread() {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
    }
    // NewThread overrides the Thread's run method:
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

Example: New Thread 2

```
class ExtendThread {
    public static void main(String args[]) {
        new NewThread();
        // After a new thread is created the new and
        // main threads continue concurrently...
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

New Thread: Which Approach?

- The Thread class defines several methods that can be overridden.
- Of these methods, only run must be overridden.
- Creating a new thread:
 - 1) implement Runnable if only run is overridden
 - 2) extend Thread if other methods are also overridden

Example: Multiple Threads

- A program may spawn as many threads as it needs.

```
class NewThread implements Runnable {
    String name;
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
```

Example: Multiple Threads

- The demonstration class creates three threads then waits until they all finish:

```
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One");
        new NewThread("Two");
        new NewThread("Three");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Using isAlive and join Methods

- How can one thread know when another thread has ended?
- Two methods are useful:
 - 1) final boolean isAlive() - returns true if the thread upon which it is called is still running and false otherwise
 - 2) final void join() throws InterruptedException – waits until the thread on which it is called terminates

Example: isAlive and join 1

- **NewThread implements the Runnable interface:**

```
class NewThread implements Runnable {
    String name;
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```


Example: isAlive and join 2

```
class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        //Checking if those threads are still alive:
        System.out.println(ob1.t.isAlive());
        System.out.println(ob2.t.isAlive());
        System.out.println(ob3.t.isAlive());
        // Waiting until all three threads have finished:
        try {
            System.out.println("Waiting to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

Example: isAlive and join 3

- Testing again if the new threads are still alive:

```
        System.out.println(ob1.t.isAlive());  
        System.out.println(ob2.t.isAlive());  
        System.out.println(ob3.t.isAlive());  
        System.out.println("Main thread exiting.");  
    }  
}
```

Thread Priorities

- Priority is used by the scheduler to decide when each thread should run.
- In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU.
- In practice, the amount of CPU time that a thread gets depends on several factors besides its priority.

Setting and Checking Priorities

- Setting thread's priority:
final void setPriority(int level)
- where level specifies the new priority setting between:
 - 1) MIN_PRIORITY (1)
 - 2) MAX_PRIORITY (10)
 - 3) NORM_PRIORITY (5)
- Obtain the current priority setting:
final int getPriority()

Synchronization

- When several threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. This way is called synchronization.
- Synchronization uses the concept of monitors:
 - 1) only one thread can enter a monitor at any one time
 - 2) other threads have to wait until the thread exits the monitor
- Java implements synchronization in two ways:
 - through the **synchronized methods** and
 - through the **synchronized statement**.

Synchronized Method

- All objects have their own implicit monitor associated with them.
- To enter an object's monitor, call this object's synchronized method.
- While a thread is inside a monitor, all threads that try to call this or any other synchronized method on this object have to wait.
- To exit the monitor, it is enough to return from the synchronized method.
- Consider first an example without synchronization...

Example: No Synchronization 1

- The call method tries to print the message string inside brackets, pausing the current thread for one second in the middle:

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

Example: No Synchronization 2

- Caller constructor obtains references to the Callme object and String, stores them in the target and msg variables, then creates a new thread:

```
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    // The Caller's run method calls the call method on the target
    // instance // of Callme, passing in the msg string:
    public void run() {
        target.call(msg);
    }
}
```


Example: No Synchronization 3

- Synch class creates a single instance of Callme and three of Caller, each with a message. The Callme instance is passed to each Caller:

```
class Synch {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        // Waiting for all three threads to finish:  
        try {  
            ob1.t.join(); ob2.t.join(); ob3.t.join();  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

No Synchronization

- Output from the earlier program:

```
[Hello[Synchronized[World]
```

```
]
```

```
]
```

- By pausing for one second, the call method allows execution to switch to another thread. A mix-up of the outputs from of the three message strings.
- In this program, nothing exists to stop all three threads from calling the same method on the same object at the same time.

Synchronized Method

- To fix the earlier program, we must serialize the access to call:

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

- This prevents other threads from entering call while another thread is using it. The output result of the program is now as follows:

```
[Hello]  
[Synchronized]  
[World]
```

Synchronized Statement

- How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?
- Put calls to the methods of this class inside the synchronized block:

```
synchronized(object) {  
    ...  
}
```

- This ensures that a call to a method that is a member of the object occurs only after the current thread has successfully entered the object's monitor.

Example: Synchronized 1

- Now the call method is not modified by synchronized:

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

Example: Synchronized 2

```
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    // The Caller's run method uses the synchronized statement to include
    // the call the target's call method:
    public void run() {
        synchronized(target) {
            target.call(msg);
        }
    }
}
```

Example: Synchronized 3

```
class Synch1 {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

Inter-Thread Communication

- Inter-thread communication relies on three methods in the Object class:
 - 1) final void wait() throws InterruptedException
tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
 - 2) final void notify()
wakes up the first thread that called wait() on the same object
 - 3) final void notifyAll()
wakes up all the threads that called wait() on the same object; the highest-priority thread will run first.
- All three must be called from within a synchronized context.

Queuing Problem

- Consider the classic queuing problem where one thread (producer) is producing some data and another (consumer) is consuming this data:
 - 1) producer should not overrun the consumer with data
 - 2) consumer should not consume the same data many times
- We consider two solutions:
 - 1) incorrect with synchronized only
 - 2) correct with synchronized and wait/notify

Example: Incorrect Queue 1

- The one-place queue class Q, with the variable n and methods get and put. Both methods are synchronized:

```
class Q {  
    int n;  
    synchronized int get() {  
        System.out.println("Got: " + n);  
        return n;  
    }  
    synchronized void put(int n) {  
        this.n = n;  
        System.out.println("Put: " + n);  
    }  
}
```

Example: Incorrect Queue 2

- Producer creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

Example: Incorrect Queue 3

- Consumer creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

Example: Incorrect Queue 4

- The PC class first creates a single Queue instance q, then creates a

Producer and Consumer that share this q:

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

Why Incorrect?

- Here is the output:

Put: 1

Got: 1

Got: 1

Put: 2

Put: 3

Get: 3

...

- Nothing stops the producer from overrunning the consumer, nor the
- consumer from consuming the same data twice.

Example: Corrected Queue 1

- The correct producer-consumer system uses wait and notify to synchronize the behavior of the producer and consumer.
- The queue class introduces the additional boolean variable valueSet used by the get and put methods:

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

Example: Corrected Queue 2

- Inside get, wait is called to suspend the execution of Consumer until Producer notifies that some data is ready:

```
synchronized int get() {  
    if (!valueSet)  
        try {  
            wait();  
        }  
    catch (InterruptedException e) {  
        System.out.println("InterruptedException");  
    }  
    // After the data has been obtained, get calls notify to tell Producer  
    // that it can put more data on the queue:  
    System.out.println("Got: " + n);  
    valueSet = false;  
    notify();  
    return n;  
}
```


Example: Corrected Queue 3

- Inside put, wait is called to suspend the execution of Producer until Consumer has removed the item from the queue:

```
synchronized void put(int n) {
    if (valueSet)
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException");
        }
    // After the next item of data is put in the queue, put calls notify to
    // tell Consumer that it can remove this item:
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
}
```

Example: Corrected Queue 6

- Producer creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    Public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

Example: Corrected Queue 7

- Consumer creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

Example: Corrected Queue 8

- The PCFixed class first creates a single Queue instance q, then creates a Producer and Consumer that share this q:

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

Deadlock

- Multi-threading and synchronization create the danger of deadlock.
- Deadlock: a circular dependency on a pair of synchronized objects.
- Suppose that:
 - 1) one thread enters the monitor on object X
 - 2) another thread enters the monitor on object Y
 - 3) the first thread tries to call a synchronized method on object Y
 - 4) the second thread tries to call a synchronized method on object X
- The result: the threads wait forever – deadlock.

Example: Deadlock 1

- Class A contains the foo method which takes an instance b of class B as a parameter. It pauses briefly before trying to call the b's last method:

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying B.last()");
        b.last();
    }
    // Class A also contains the synchronized method last:
    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
```

Example: Deadlock 2

- Class B contains the bar method which takes an instance a of class A as a parameter. It pauses briefly before trying to call the a's last method:

```
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying A.last()");
        a.last();
    }
    // Class B also contains the synchronized method last:
    synchronized void last() {
        System.out.println("Inside B.last");
    }
}
```

Example: Deadlock 3

- The main Deadlock class creates the instances a of A and b of B:

```
class Deadlock implements Runnable {
```

```
    A a = new A();
```

```
    B b = new B();
```

```
    // The constructor creates and starts a new thread, and creates a lock on the  
    // object in the main thread (running foo on a) with b passed as a parameter:
```

```
    Deadlock() {
```

```
        Thread.currentThread().setName("MainThread");
```

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

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

```
        a.foo(b);
```

```
        System.out.println("Back in main thread");
```

```
    }
```

```
    // The run method creates a lock on the b object in the new thread  
    // (running bar on b) with a passed as a parameter:
```

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

```
        b.bar(a);
```

```
        System.out.println("Back in other thread");
```

```
    }
```

```
    // Create a new Deadlock instance:
```

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

```
        new Deadlock();
```

```
    }
```

```
}
```


Deadlock Reached

- Program output:

MainThread entered A.foo

RacingThread entered B.bar

MainThread trying to call B.last()

RacingThread trying to call A.last()

- RacingThread owns the monitor on b while waiting for the monitor on a.
- MainThread owns the monitor on a while it is waiting for the monitor on b.
- The program deadlocks!

Suspending/Resuming Threads

- Thread management should use the run method to check periodically whether the thread should suspend, resume or stop its own execution.
- This is usually accomplished through a flag variable that indicates the execution state of a thread, e.g.
 - 1) running – the thread should continue executing
 - 2) suspend – the thread must pause
 - 3) stop – the thread must terminate

Example: Suspending/Resuming 1

- boolean variable `suspendFlag` to control the execution of a thread, initialized to `false`:

```
class NewThread implements Runnable {
    String name;
    Thread t;
    boolean suspendFlag;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start();
    }
}
```

Example: Suspending/Resuming 2

// The run method contains the synchronized statement that checks suspendFlag. If true, the wait method is called.

```
public void run() {
    try {
        for (int i = 15; 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.");
}
```

Example: Suspending/Resuming 3

```
// The mysuspend method sets suspendFlag to true:
void mysuspend() {
    suspendFlag = true;
}
// The myresume method sets suspendFlag to false and invokes notify
// to wake up the thread:
synchronized void myresume() {
    suspendFlag = false;
    notify();
}
}
```

Example: Suspending/Resuming 4

- SuspendResume class creates two instances ob1 and ob2 of NewThread, therefore two new threads, through its main method:

```
class SuspendResume {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
    }  
}
```

Example: Suspending/Resuming 5

- **The two threads are kept running, then suspended, then resumed from the main thread:**

```
try {  
    Thread.sleep(1000);  
    ob1.mysuspend();  
    System.out.println("Suspending thread One");  
    Thread.sleep(1000);  
    ob1.myresume();  
    System.out.println("Resuming thread One");  
    ob2.mysuspend();  
    System.out.println("Suspending thread Two");  
    Thread.sleep(1000);  
    ob2.myresume();  
    System.out.println("Resuming thread Two");  
} catch (InterruptedException e) {  
    System.out.println("Main thread Interrupted");  
}
```

Example: Suspending/Resuming 6

- The main thread waits for the two child threads to finish, then finishes itself:

```
try {
    System.out.println("Waiting to finish.");
    ob1.t.join();
    ob2.t.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
```





The Last Word on Multi-Threading

- Multi-threading is a powerful tool to writing efficient programs.
- When you have two subsystems within a program that can execute concurrently, make them individual threads.
- However, creating too many threads can actually degrade the performance of your program because of the cost of context switching.

- Exercise: Multi-Threading 1
- 1) Create a new main class called MultiThread
- 2) Create a new class called MemoryThread . This class should implement the interface Runnable so that it can be run as a thread.
- This Thread will monitor the memory usage on the system.
- a) Add void start(), stop() and run() methods.
- b) The run() method should print to the screen every 5 seconds the amount of memory presently being used. This can be done using these two lines of code:
- `Runtime r = Runtime.getRuntime();`
- `long memoryUsed = r.totalMemory()- r.freeMemory();`
- 3) Create a new class called ClockThread. This class should implement the interface Runnable so that it can be run as a thread. This Thread will monitor what the time is.
- a) The constructor for ClockThread should take an argument that sets the time (in seconds) that this thread will run for.
- b) Add void start(), stop() and run() methods.

- Exercise: Multi-Threading 2
- c) The run() method should print the current time to the screen every 5 seconds. This can be done using the built in Date class.
- `import java.util.Date;`
- `Date timeNow = new Date();`
- `System.out.println(timeNow.toString());`
- The run() method should stop when the elapsed time set in the constructor has been reached.
- 4) In the main() method of MultiThread create the two thread objects and start them; using the *start()* method.
- 5) Add a while loop, that will print the number of active Threads every one second. A one second pause can be implemented as follows:
 - `while (true) {`
 - `Thread.sleep(1000);`
 - `}`
- 6) The number of threads can be monitored using the ThreadGroup class as follows:
 - `int numThreads =`
 - `Thread.currentThread().getThreadGroup().activeCount();`

- 
- Exercise: Multi-Threading 3
 - 7) This while loop should terminate after one minute and the two threads
 - stopped by invoking their stop() methods.
 - 8) When printing to the screen - the Threads names should be appended so
 - that it is clear from which process the data is originating.
 - 9) The MemoryThread thread should be assigned a maximum priority and
 - the ClockThread thread a minimum priority.

Example: Priorities 1

```
class Clicker implements Runnable {
    int click = 0;
    Thread t;
    private volatile boolean running = true;
    // A new thread is created, its priority initialized:
    public Clicker(int p) {
        t = new Thread(this);
        t.setPriority(p);
    }
    // When running, click is incremented. When stopped, running is false:
    public void run() {
        while (running) {
            click++;
        }
    }
    public void stop() {
        running = false;
    }
    public void start() {
        t.start();
    }
}
```

Example: Priorities 3

```
class HiLoPri {
    public static void main(String args[]) {
        // The main thread is set at the highest priority, the new threads at
        // two above and two below the normal priority:
        Thread.currentThread().
        setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
        // The threads are started and allowed to run for 10 seconds:
        lo.start(); hi.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        // After 10 seconds, both threads are stopped and click variables printed:
        lo.stop(); hi.stop();
        try {
            hi.t.join(); lo.t.join();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException");
        }
        System.out.println("Low-priority: " + lo.click);
        System.out.println("High-priority: " + hi.click);
    }
}
```

Volatile Variable

- The volatile keyword is used to declare the running variable:

```
private volatile boolean running = true;
```

- This is to ensure that the value of running is examined at each iteration of:

```
while (running) {  
    click++;  
}
```

- Otherwise, Java is free to optimize the loop in such a way that a local copy of running is created. The use of volatile prevents this optimization.