UNIT - IV

Exception-Handling

&

Multithreading

Exception-Handling and Multithreading

Exception-Handling: Exception handling basics, different types of exception classes, use of try & catch with throw, throws & finally, creation of user defined exception classes.

Multithreading: Basics of multithreading, main thread, thread life cycle, creation of multiple threads, thread priorities, thread synchronization, inter-thread communication, deadlocks for threads, suspending & resuming threads.

Multithreading

- Basics of Multithreading
- Main thread
- Thread life cycle
- Creation of multiple threads
- Thread priorities
- Thread synchronization
- Inter-thread communication
- Deadlocks for threads
- Suspending & Resuming threads

Multithreading

Basics:

- Unlike many other computer languages, Java provides built-in support for *multithreaded programming*.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: ***process based*** and ***thread-based***.

Multithreading

Basics: …

- ***PROCESS-BASED* MULTITASKING**
- A *process* is a program that is executing. Thus, *process-based* multitasking is the feature that allows our computer to run two or more programs concurrently.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
- In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Multithreading

Basics: …

- ***THREAD-BASED*** **MULTITASKING**
- In a *thread-based multitasking environment, the thread is the smallest unit of dispatchable* code. This means that a single program can perform two or more tasks simultaneously.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multithreading

Basics: …

***THREAD-BASED vs.PROCESS-BASED* MULTITASKING**

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java.

Multithreading

Basics: …

***THREAD-BASED vs.PROCESS-BASED* MULTITASKING…**

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
- For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU.
- And, of course, user input is much slower than the computer. In a single-threaded environment, our program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

Multithreading

- The Java Thread Model
- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.

Multithreading

- The Java Thread Model ...
- **Eg:** The idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Advantage of Java Multithreading:

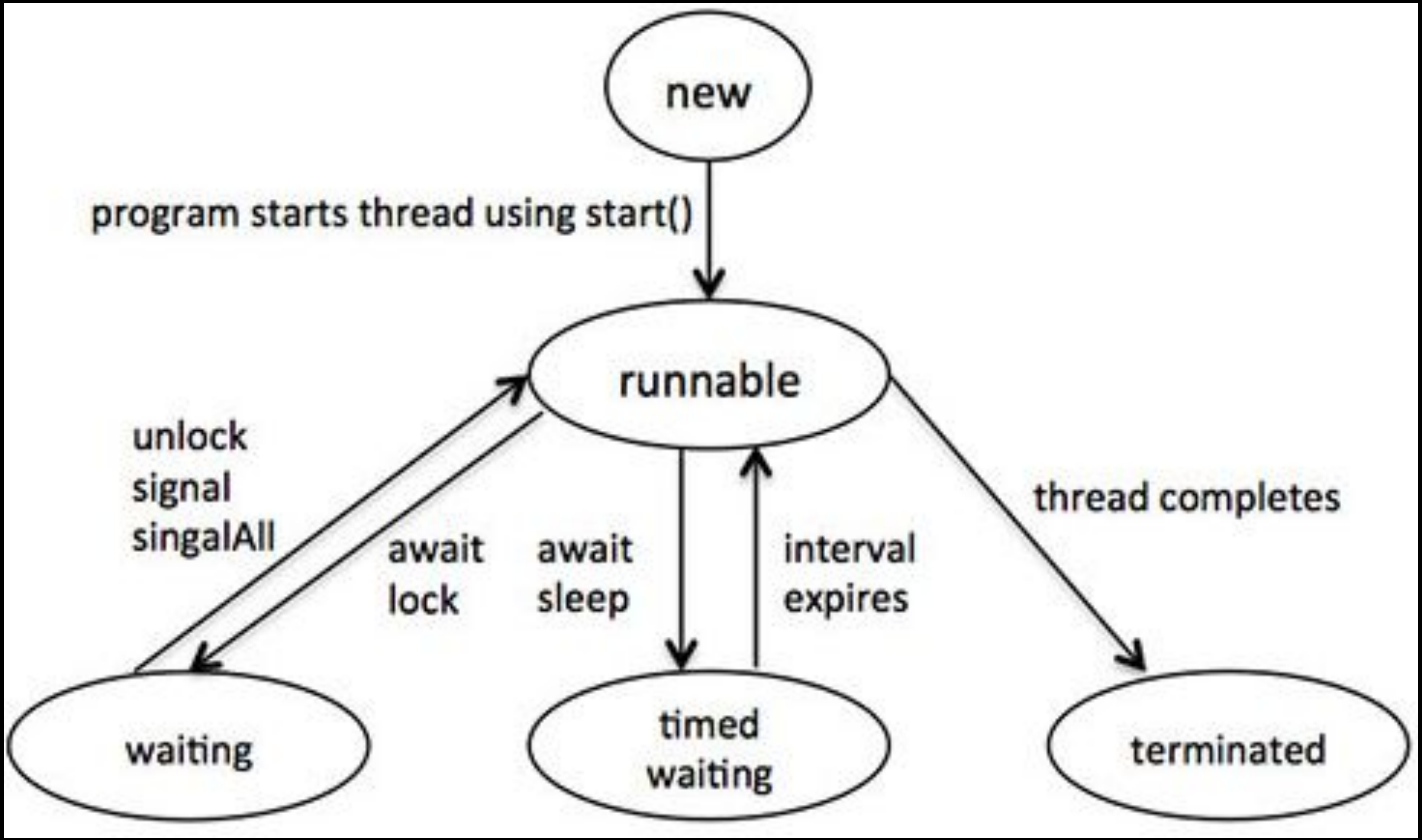
- It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- You **can perform many operations together so it saves time**.
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multithreading

- The Java Thread Model ...
- **Life Cycle of a Thread:**
  - Threads exist in several states. A thread can be ***running***.
  - It can be ***ready to run*** as soon as it gets CPU time.
  - A running thread can be ***suspended***, which temporarily suspends its activity.
  - A suspended thread can then be ***resumed***, allowing it to pick up where it left off.
  - A thread can be ***blocked*** when waiting for a resource.
  - At any time, a thread can be **terminated,** which halts its execution immediately.
  - Once terminated, a thread cannot be resumed.

Multithreading

- The Java Thread Model ...
- **Life Cycle of a Thread: ...**



Multithreading

- The Java Thread Model ...

Thread Priorities

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

Multithreading

- The Java Thread Model ...

Thread Priorities ...

- The rules that determine when a context switch takes place are simple:

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

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

Multithreading

- The Java Thread Model ...

Thread Priorities ...

- *In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.*

- 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).

Multithreading

- The Java Thread Model ...

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it.
- For this purpose, Java implements an elegant twist on an age-old model of inter-process synchronization: ***the monitor***.
- **The monitor is a control mechanism first defined by C.A.R. Hoare.**

Multithreading

- The Java Thread Model ...

Synchronization ...

- A monitor is a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
- Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate.
- **The synchronization is mainly used to,**

To prevent thread interference.

To prevent consistency problem.

Multithreading

- The Java Thread Model ...

Synchronization ...

- Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when **one of the object's synchronized methods** is called.
- **Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.**

Multithreading

- The Java Thread Model ...

Messaging

- After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead.
- By contrast, Java provides a clean, low-cost way for two or more ***threads to talk to each other, via calls to predefined methods that all objects have.***
- Java's messaging system ***allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to***

Multithreading

- The Java Thread Model ...

The Thread Class and the Runnable Interface

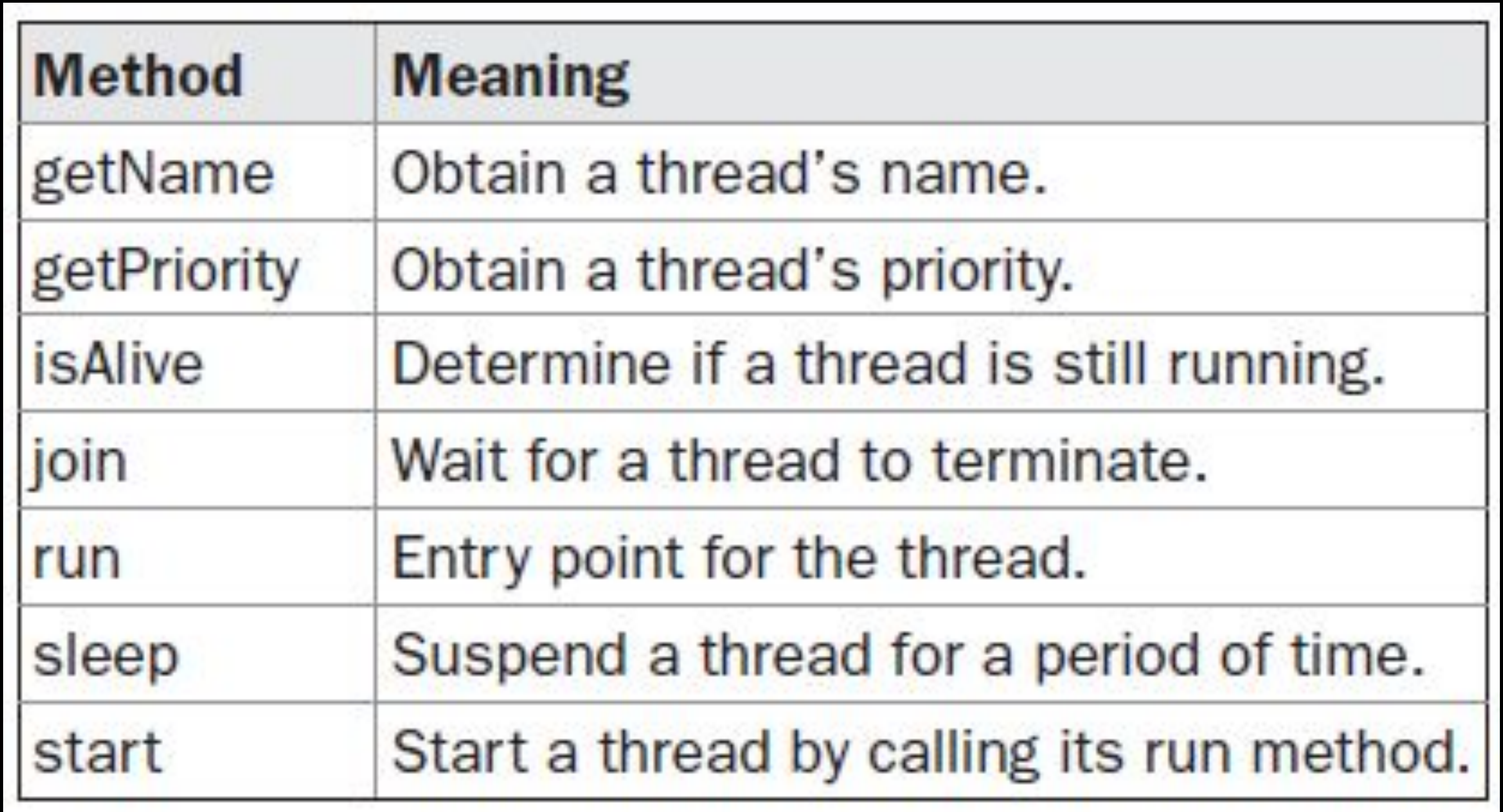
- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable interface.**

Multithreading

- The Java Thread Model ...

The Thread Class and the Runnable Interface ...

- The **Thread** class defines several methods that help manage



Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Multithreading

- The Main thread
- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:

✔ It is the thread from which other “child” threads will be spawned.

✔ Often, it must be the last thread to finish execution because it performs various shutdown actions.

Multithreading

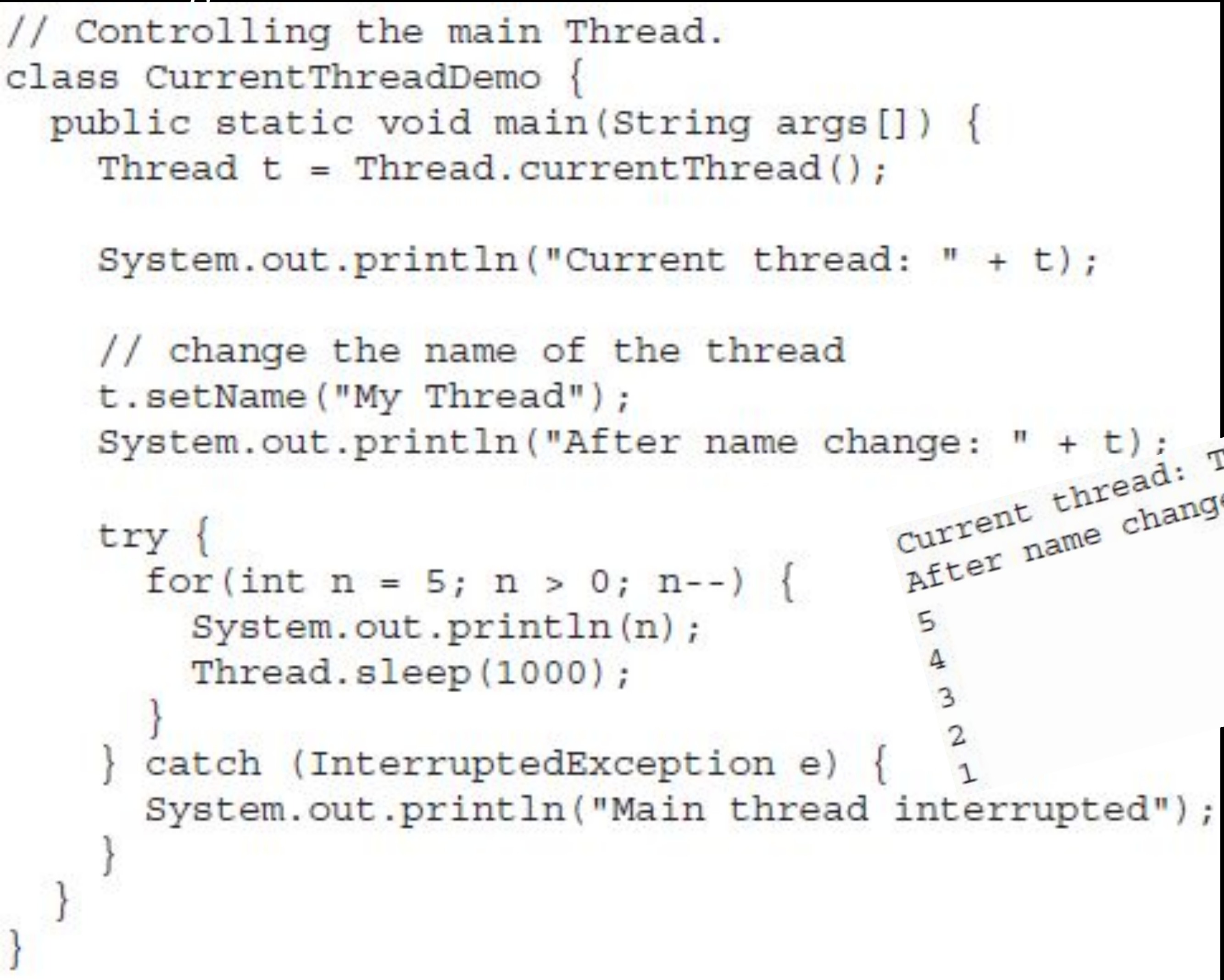
- The Main thread ...
- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread.**
- The General form of the method currentThread( )

static Thread currentThread( )

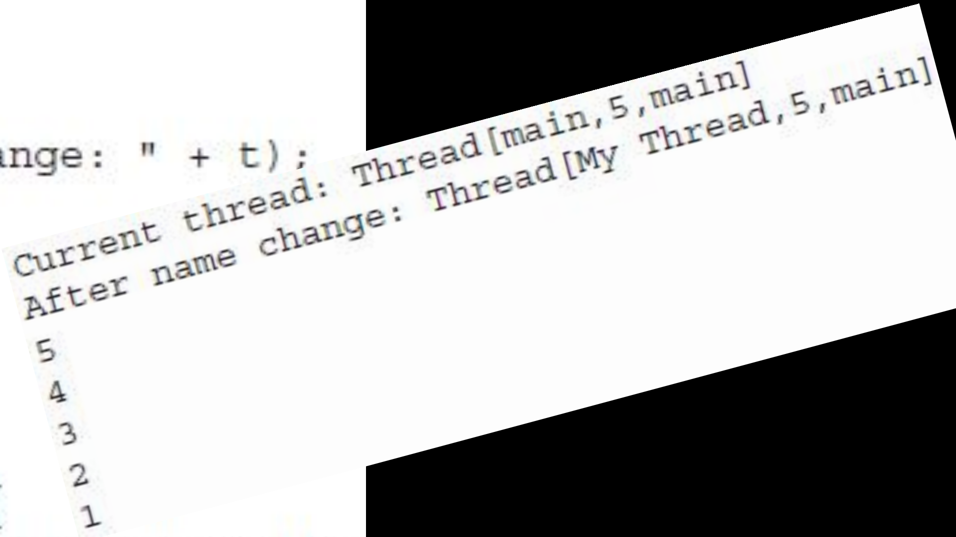
- This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Multithreading

- The Main thread ...
- **Eg:**



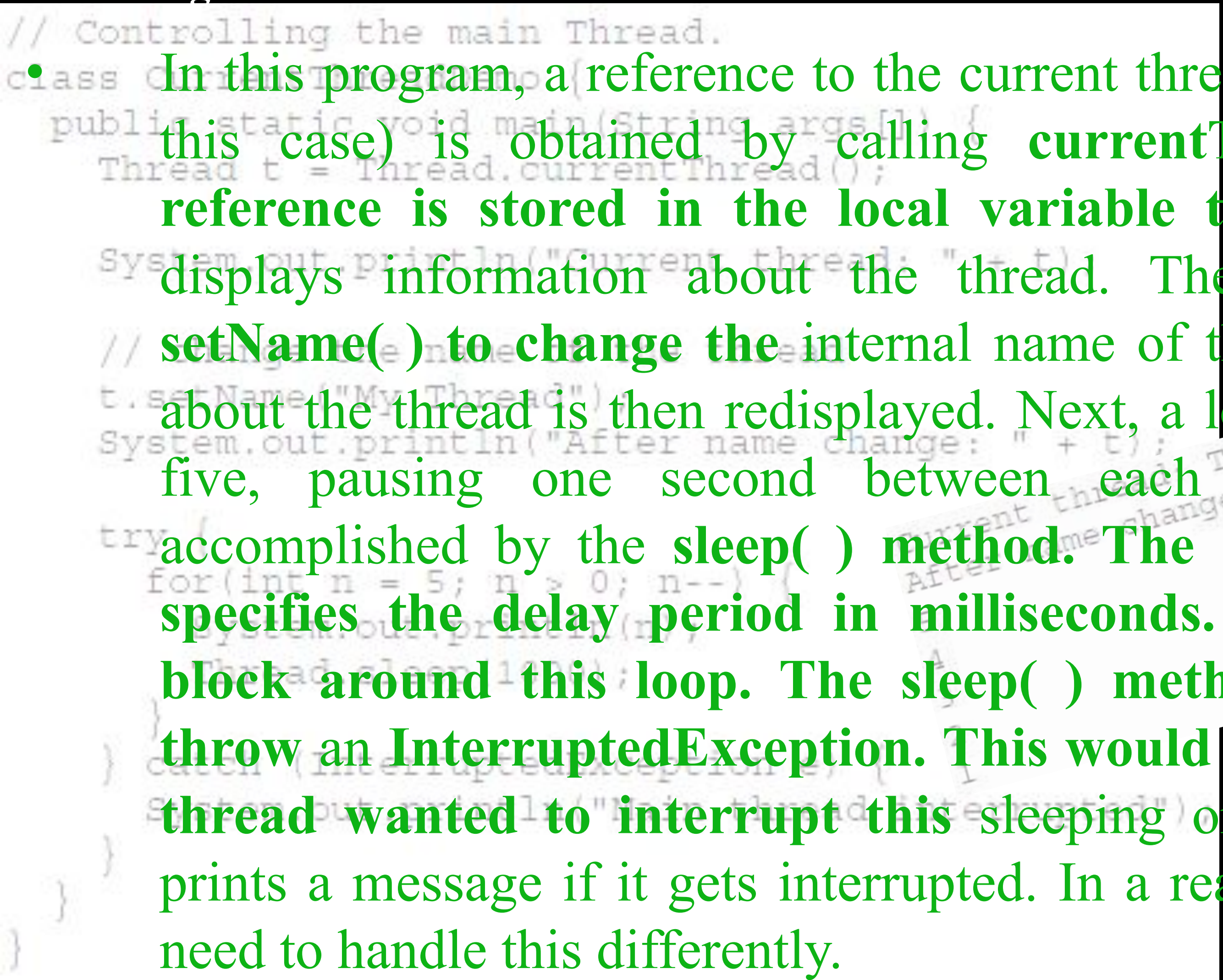
```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

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

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```



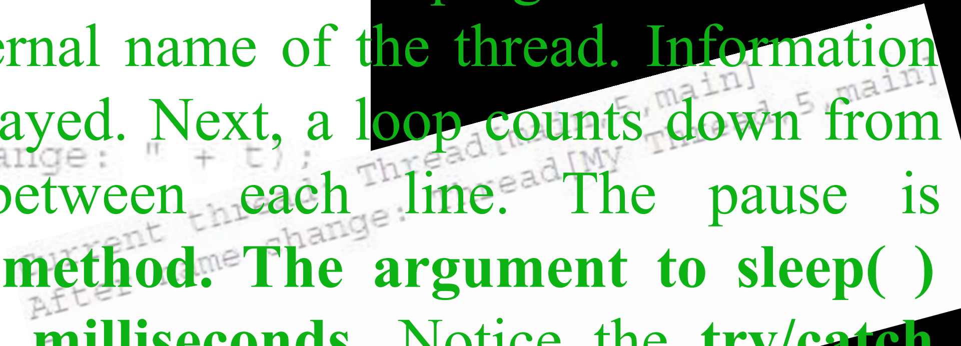
```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Multithreading

- The Main thread ...
- **Eg:**



```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

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

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

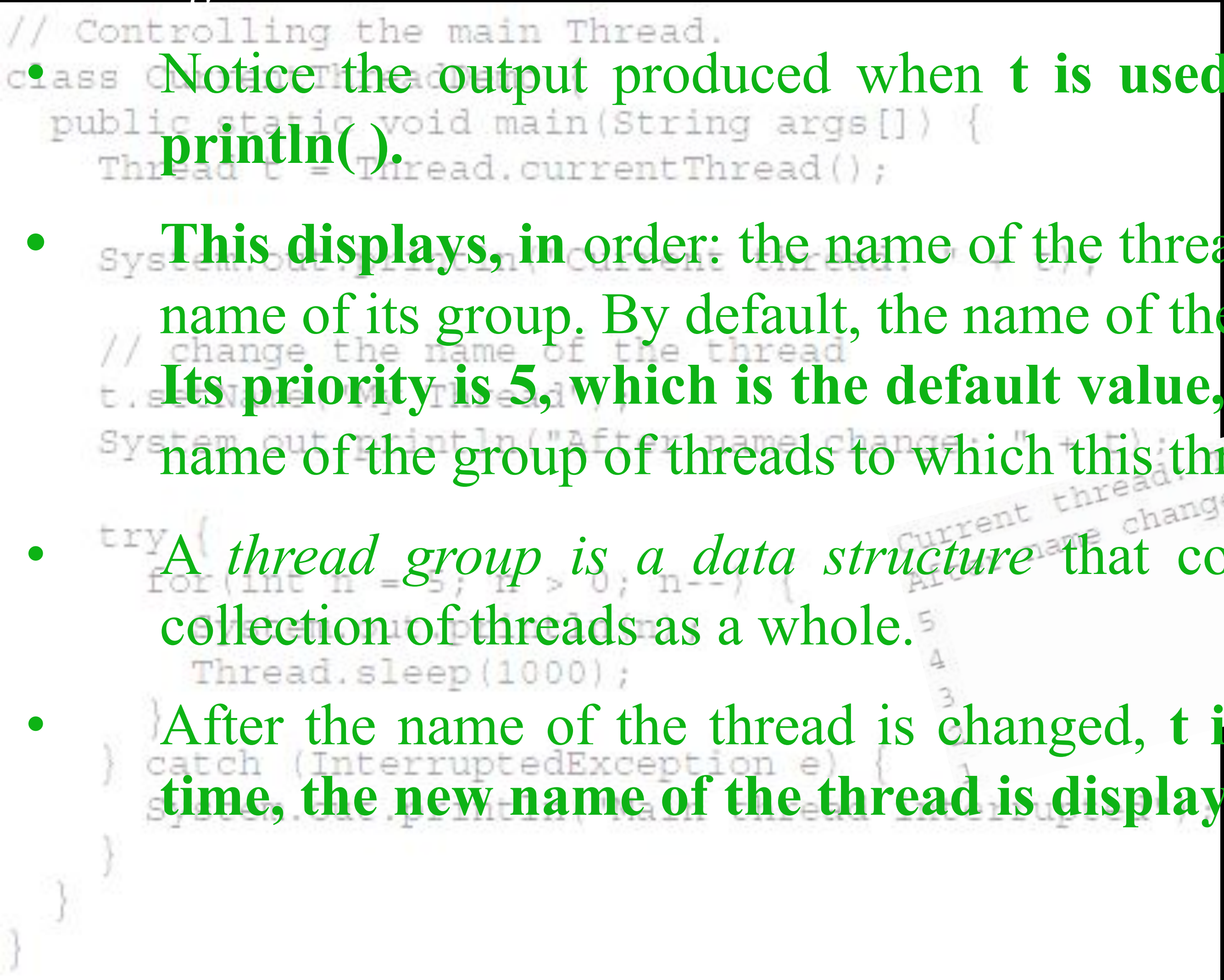


```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
```

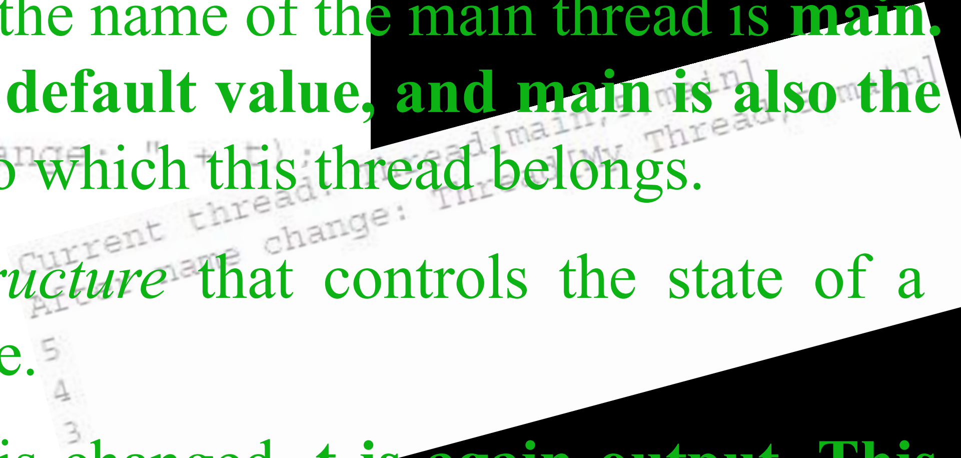
- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( ), and this reference is stored in the local variable t. Next, the program** displays information about the thread. The program then calls **setName( ) to change the** internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( ) method. The argument to sleep( ) specifies the delay period in milliseconds.** Notice the **try/catch block around this loop. The sleep( ) method in Thread might throw** an **InterruptedException. This would happen if some other thread wanted to interrupt this** sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently.

Multithreading

- The Main thread ...
- **Eg:**



```
// Controlling the main Thread.
class C[illegible]
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

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

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        [illegible]
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```



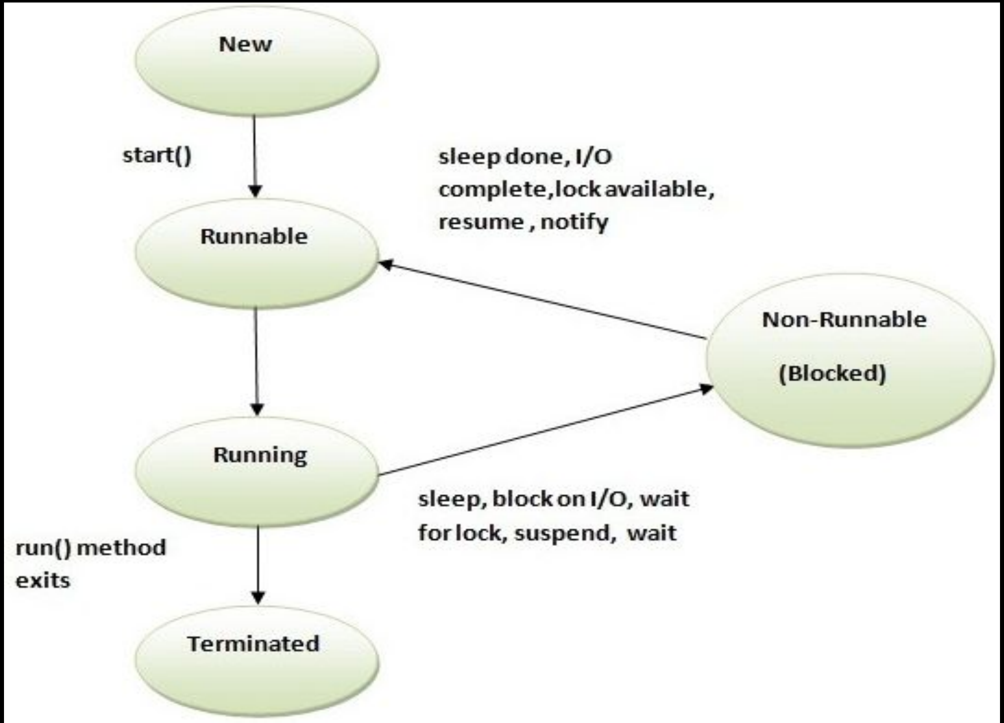
```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
```

- Notice the output produced when **t is used as an argument to println( ).**
- **This displays, in** order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main. Its priority is 5, which is the default value, and main is also the** name of the group of threads to which this thread belongs.
- A *thread group is a data structure* that controls the state of a collection of threads as a whole.
- After the name of the thread is changed, **t is again output. This time, the new name of the thread is displayed.**

Multithreading

- Thread life cycle

- As the process has several states, similarly a thread exists in several states. A thread can be in the following states:
  - ✔ ***Ready to run (New):*** First time as soon as it gets CPU time.
  - ✔ ***Running:*** Under execution.
  - ✔ ***Suspended:*** Temporarily not active or under execution.
  - ✔ ***Blocked:*** Waiting for resources.
  - ✔ ***Resumed:*** Suspended thread resumed, and start from where it left off.
  - ✔ ***Terminated:*** Halts the execution immediately and never resumes.


New
start()
sleep done, I/O complete, lock available, resume, notify
Runnable
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

Multithreading

- Creating a Thread
  - Implementing Runnable
  - Extending Thread
  - Choosing an Approach

Creating a Thread

- One can create a thread by instantiating an object of type **Thread.**
- Java defines two ways in which thread can be accomplished: by implementing the **Runnable interface** and by extending the **Thread class.**

Multithreading

- Creating a Thread ...

Implementing Runnable:

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable.**
- To implement **Runnable,** a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

Multithreading

- Creating a Thread ...

Implementing Runnable: ...

- Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.
- After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here: Thread(Runnable *threadOb*, *String*

Multithreading

- Creating a Thread ...

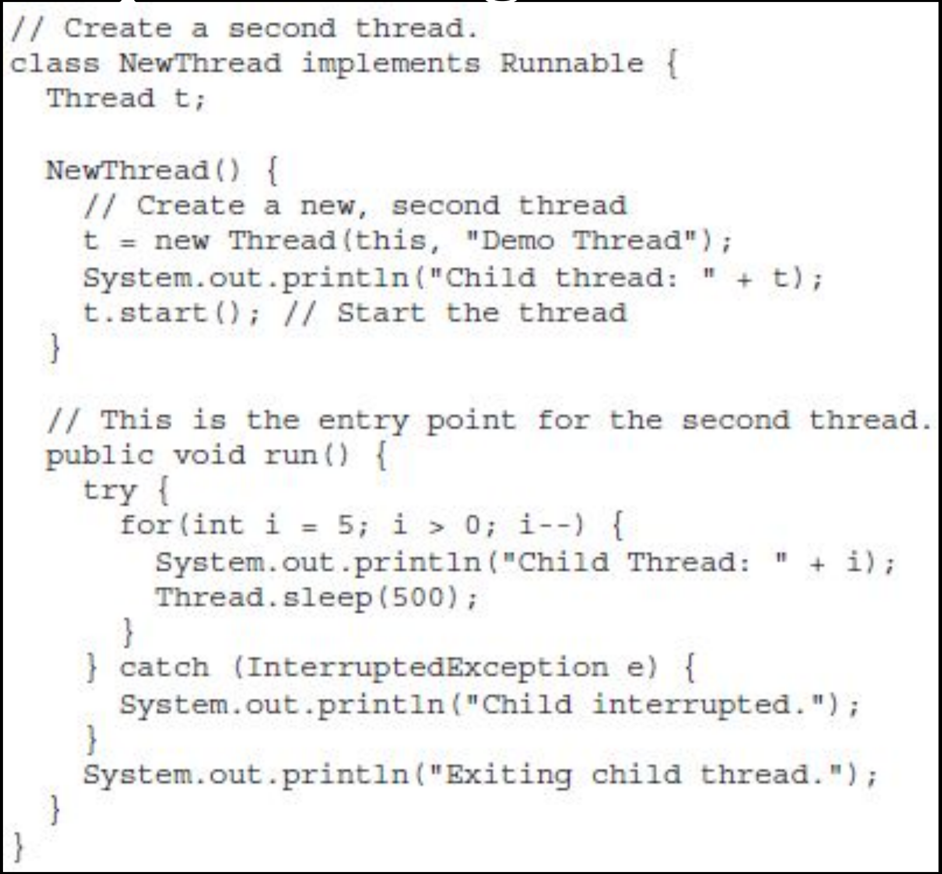
Implementing Runnable: ...

- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( ).**
- **The start( )** method is shown here: **void start( )**

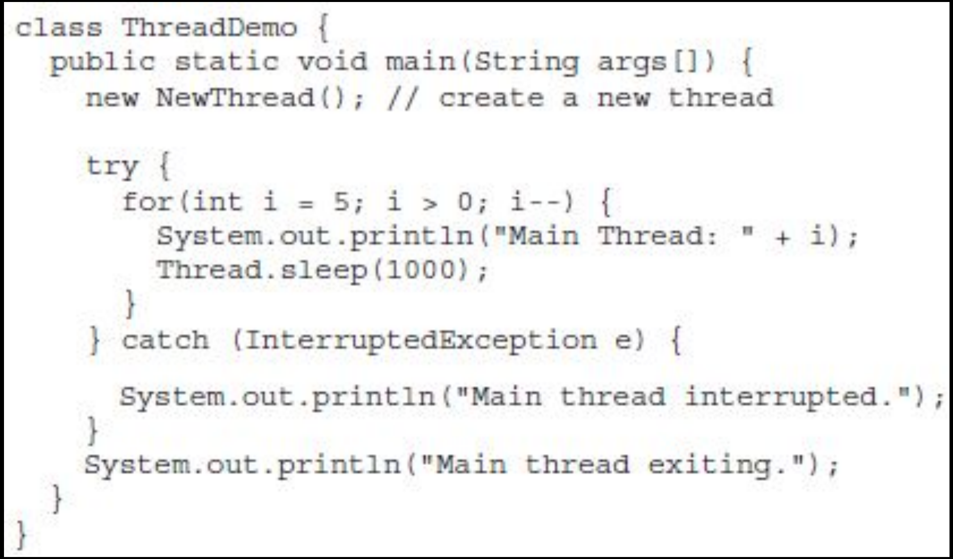
Multithreading

- Creating a Thread ...

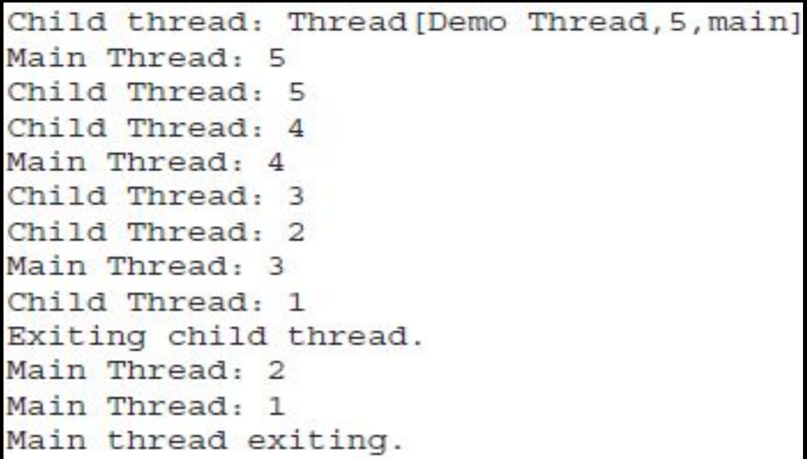
Implementing Runnable: ...



```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  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.");
  }
}
```



```
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    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.");
  }
}
```



```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Multithreading

- Creating a Thread ...

Extending Thread:

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

Multithreading

- Creating a Thread ...

Extending Thread: ...

```java
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  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.");
  }
}
```

```java
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    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.");
  }
}
```

Multithreading

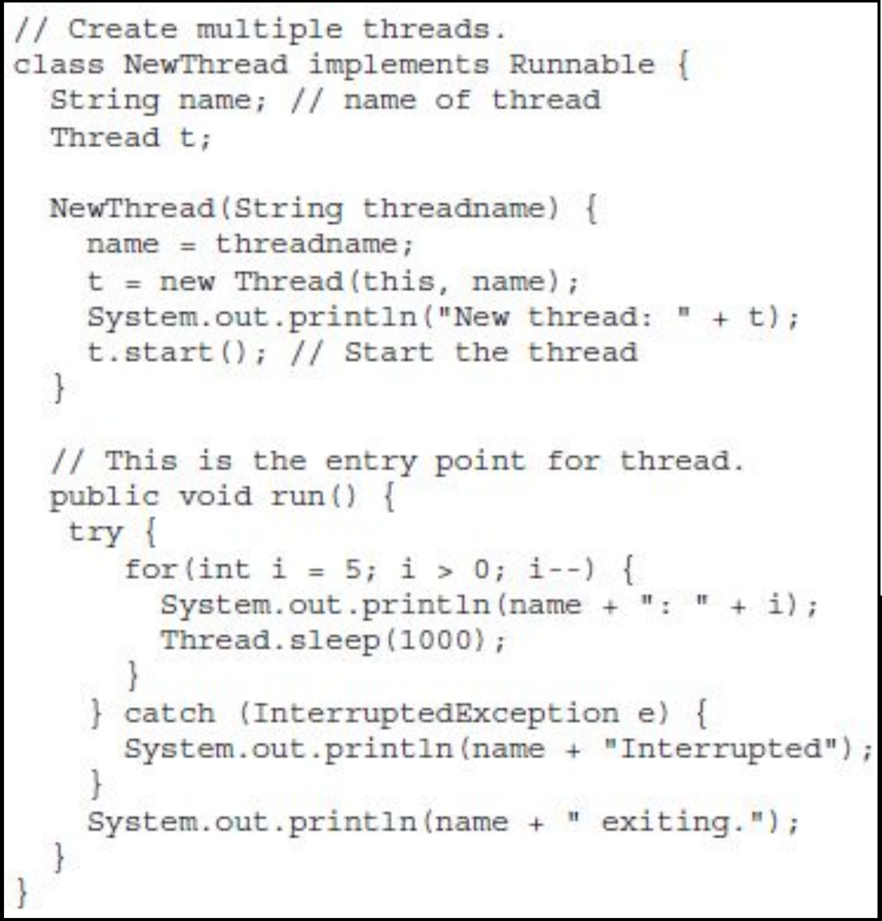
- Creating a Thread ...

Choosing an Approach:

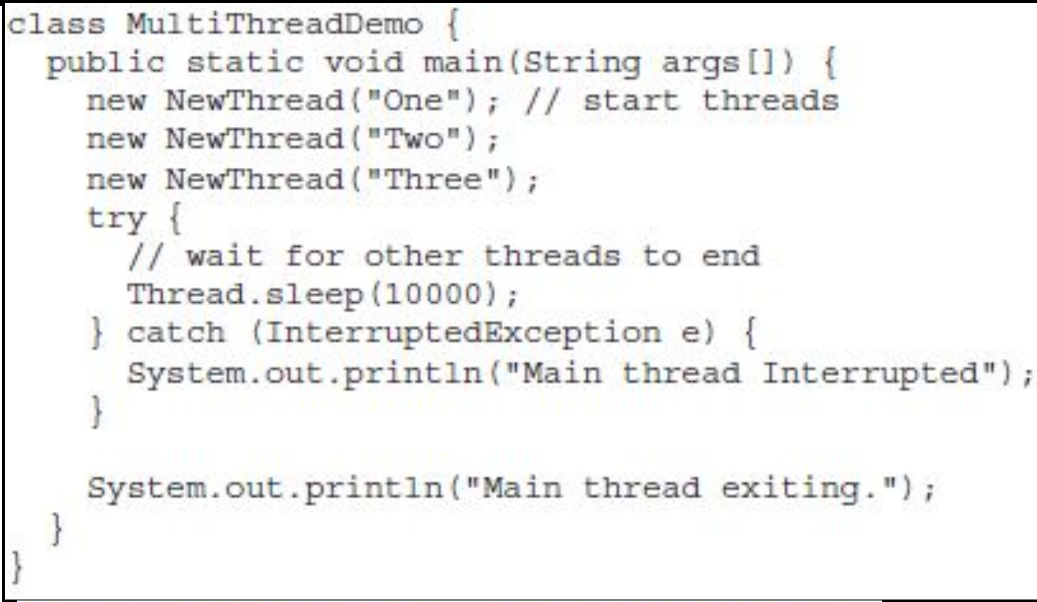
- At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point.
- The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must be overridden is* ***run( )***. This is, of course, the same method required when you implement **Runnable.**
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread's** other methods, it is probably best simply to implement **Runnable**.

Multithreading

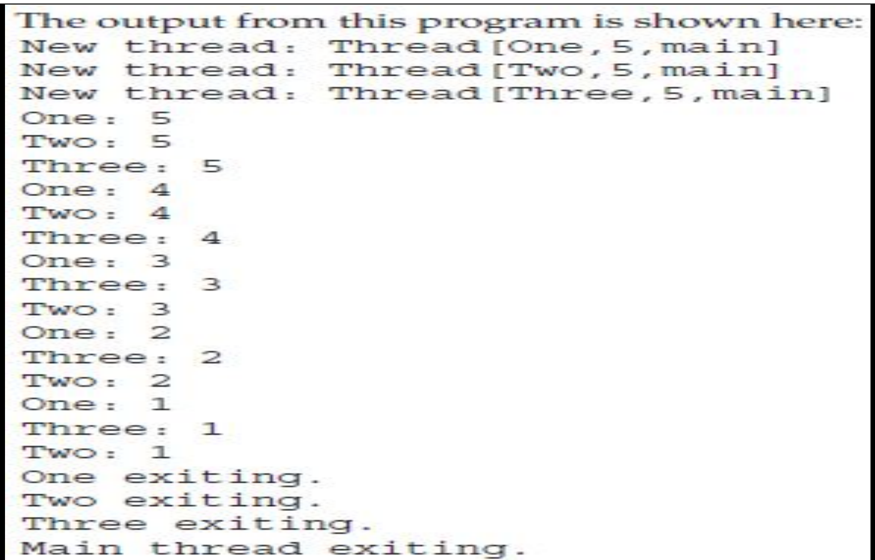
- Creation of multiple threads



```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

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

  // This is the entry point for thread.
  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.");
  }
}
```



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

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



The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Multithreading

- Using isAlive( ) and join( )
- How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished.
- First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

Multithreading

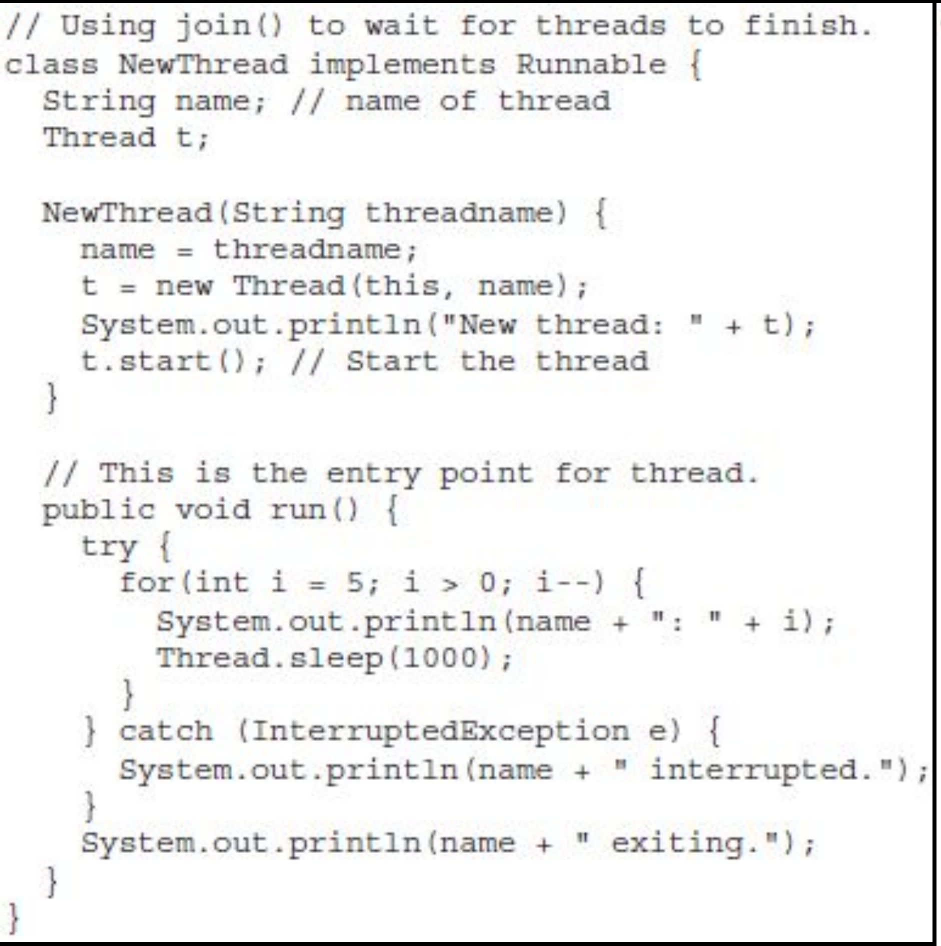
- Using isAlive( ) and join( ) ...
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

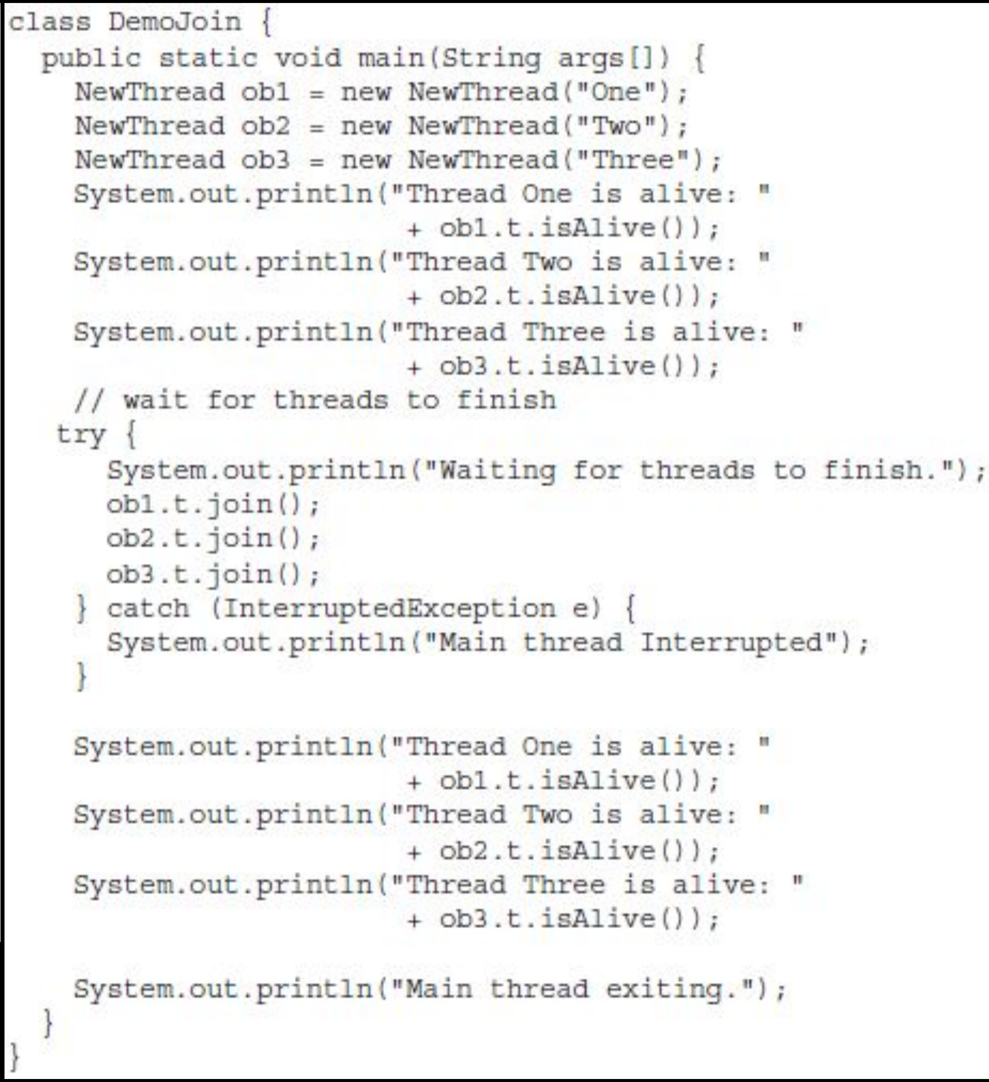
- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Multithreading

- Using isAlive( ) and join( ) ...



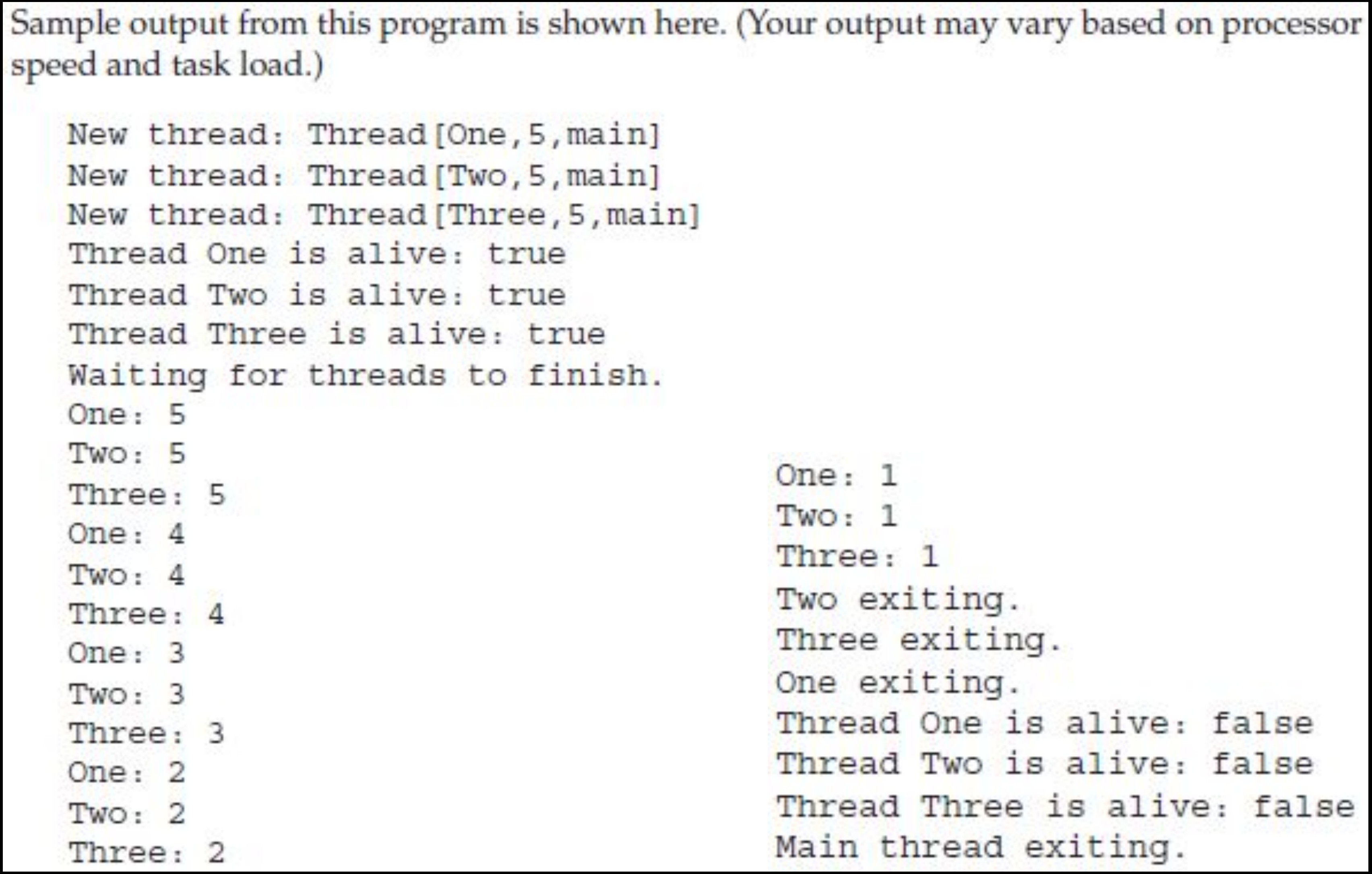
```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

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

  // This is the entry point for thread.
  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.");
  }
}
```



```
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
    // wait for threads to finish
   try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());

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

Multithreading

- Using isAlive( ) and join( ) ...



Sample output from this program is shown here. (Your output may vary based on processor speed and task load.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

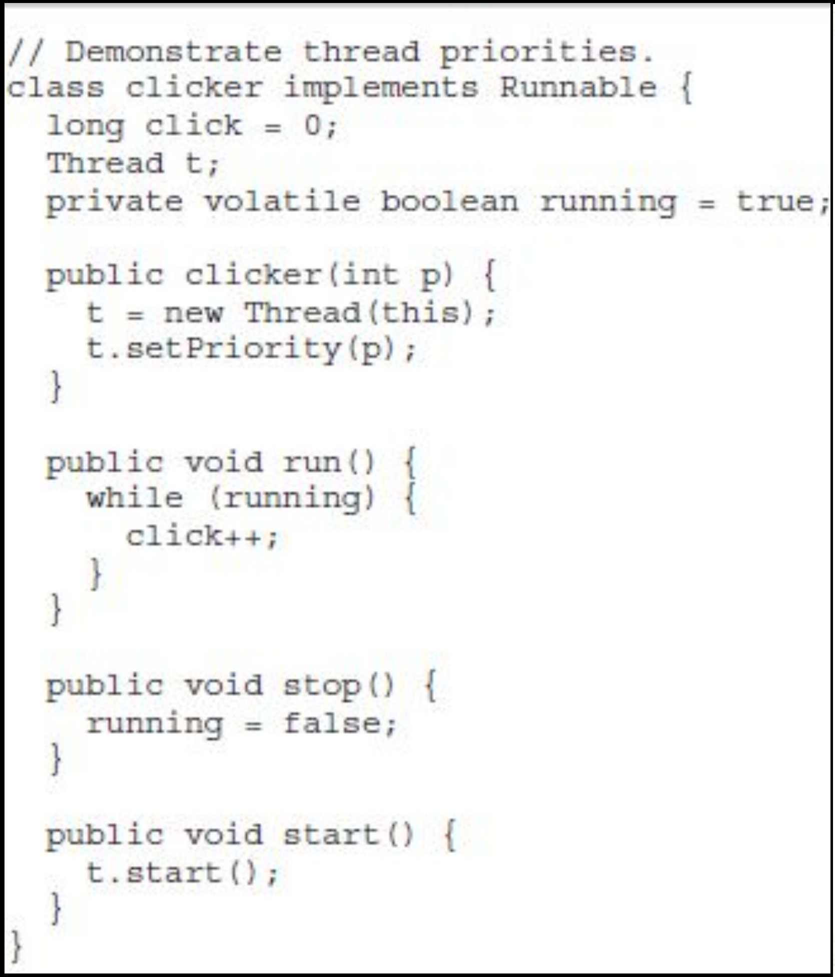
Multithreading

- Thread priorities
- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread.** This is its general form:

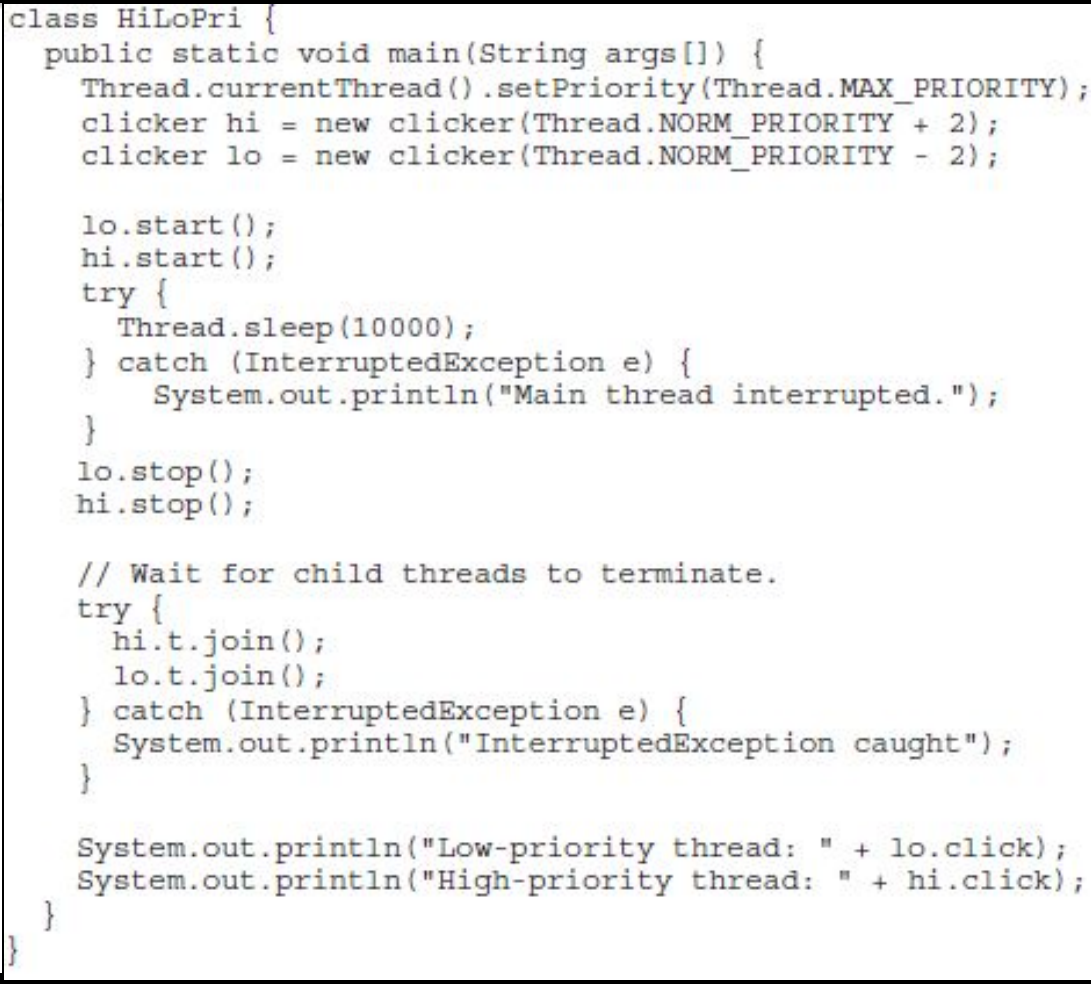
  final void setPriority(int *level)*
- Here, *level* specifies the new priority setting for the calling thread.
- The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY. Currently**, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY,** which is currently 5. These priorities are defined as **static final** ariables within **Thread.**
- To obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here: ***final int getPriority( )***

Multithreading

- Thread priorities ...



```
// Demonstrate thread priorities.
class clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run() {
    while (running) {
      click++;
    }
  }

  public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}
```



```
class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
    }
    lo.stop();
    hi.stop();

    // Wait for child threads to terminate.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

Multithreading

- Thread synchronization
- When two or more 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. The process by which this is achieved is called *synchronization*.
- If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because these languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives.
- Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword.

Multithreading

- Thread synchronization ...

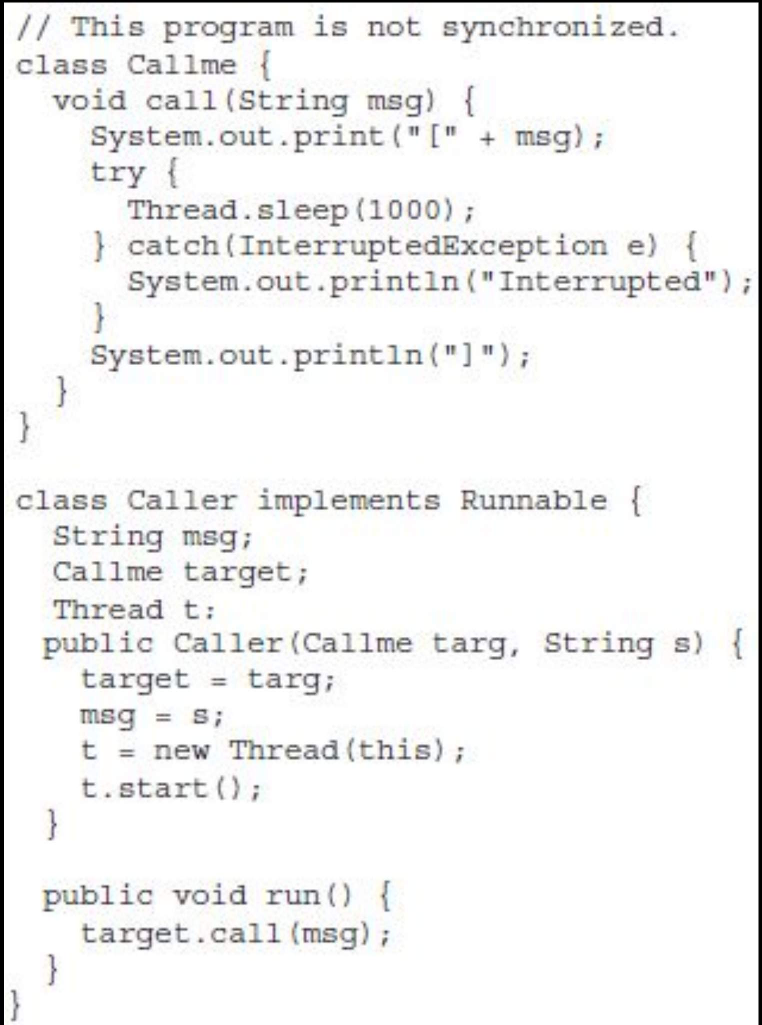
Using Synchronized Methods:

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

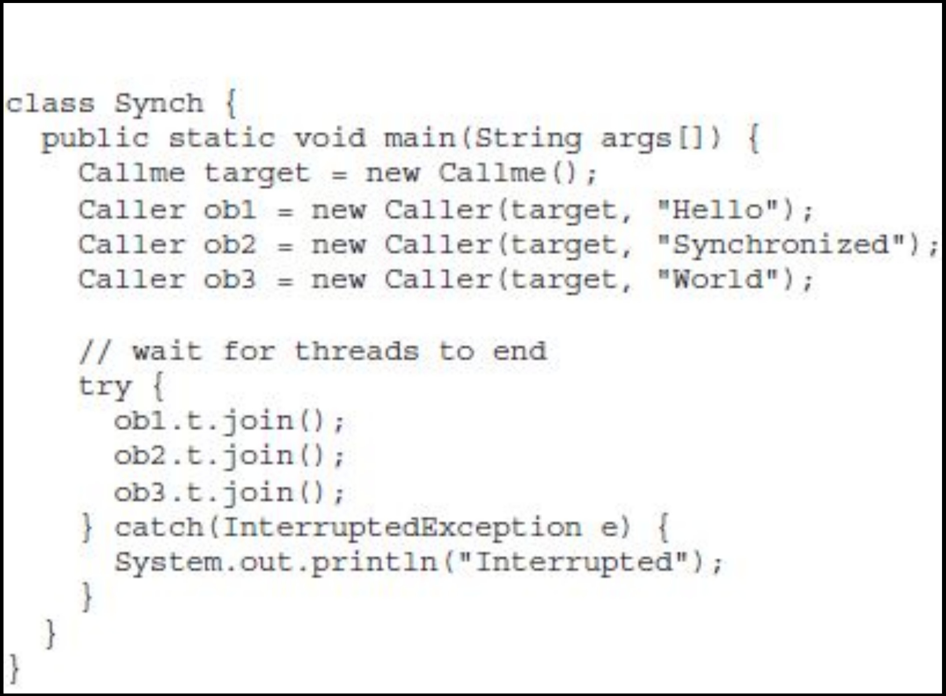
Multithreading

- Thread synchronization ...

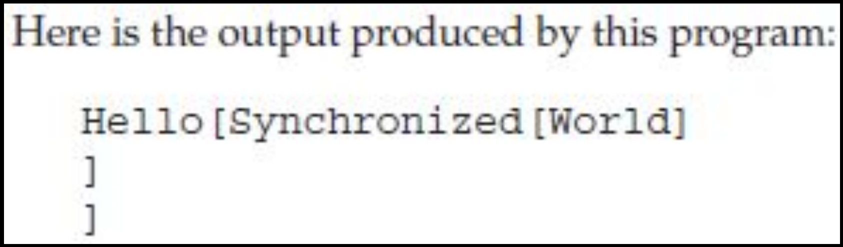
Using Synchronized Methods: ... *Not Synchronized Program*



```
// This program is not 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("]");
  }
}

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();
 }

 public void run() {
    target.call(msg);
 }
}
```



```
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");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```



Here is the output produced by this program:

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

Multithreading

- Thread synchronization ...

Using Synchronized Methods: ...

- To fix the preceding program, you must *serialize access to **call( )***. That is, you must restrict its access to only one thread at a time.
- To do this, you simply need to precede **call( )**'s definition
- with the keyword **synchronized**, as shown here:

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

- This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

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

Multithreading

- Thread synchronization ...

Using Synchronized Methods: ...

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions.
- Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
- However, nonsynchronized methods on that instance will continue to be callable.

Multithreading

- Thread synchronization ...

The synchronized Statement:

- While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class.
- How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply **put calls to the methods defined by this class inside a synchronized block.**

Multithreading

- Thread synchronization ...

The synchronized Statement: ...

- This is the general form of the **synchronized statement:**

```
synchronized(object) {
    // statements to be synchronized
}
```

  Here, *object* is a reference to the object being synchronized.

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

Multithreading

- Thread synchronization ...

The synchronized Statement: ...

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

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();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```

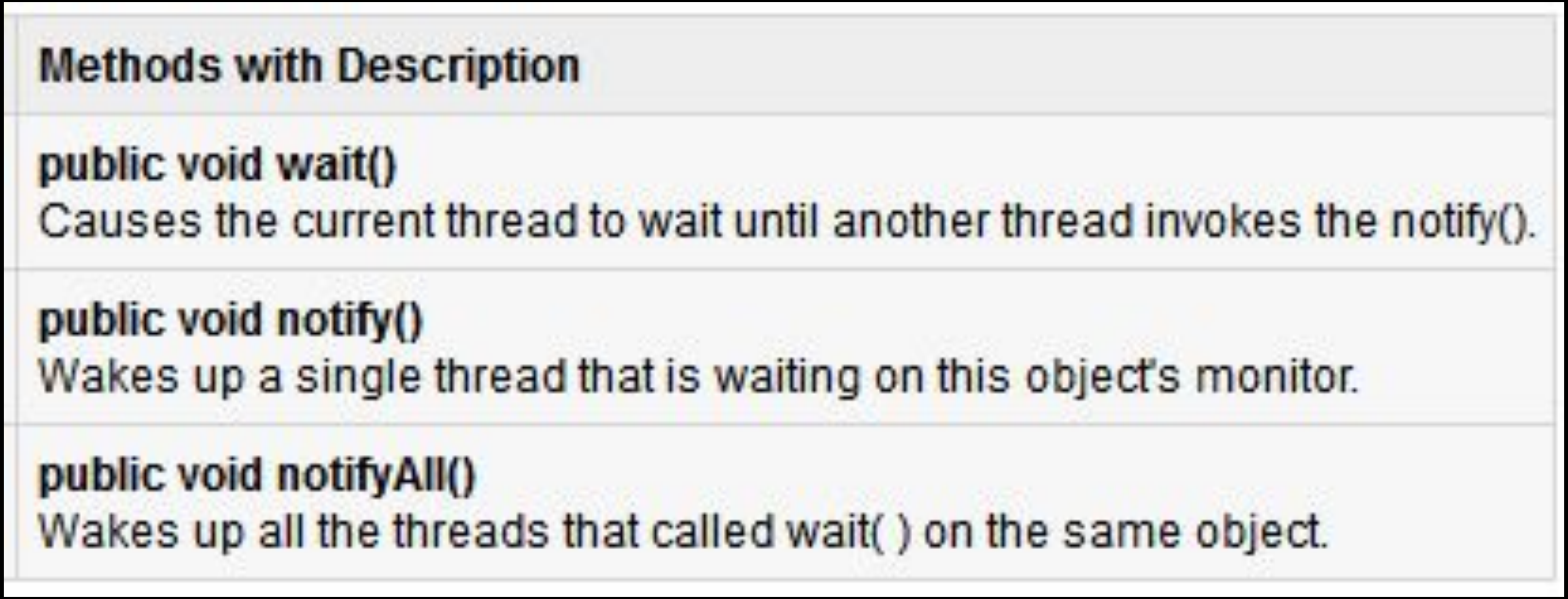
```
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

Multithreading

- Inter-thread communication
- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object** class:
  - wait()
  - notify()
  - notifyAll()

Multithreading

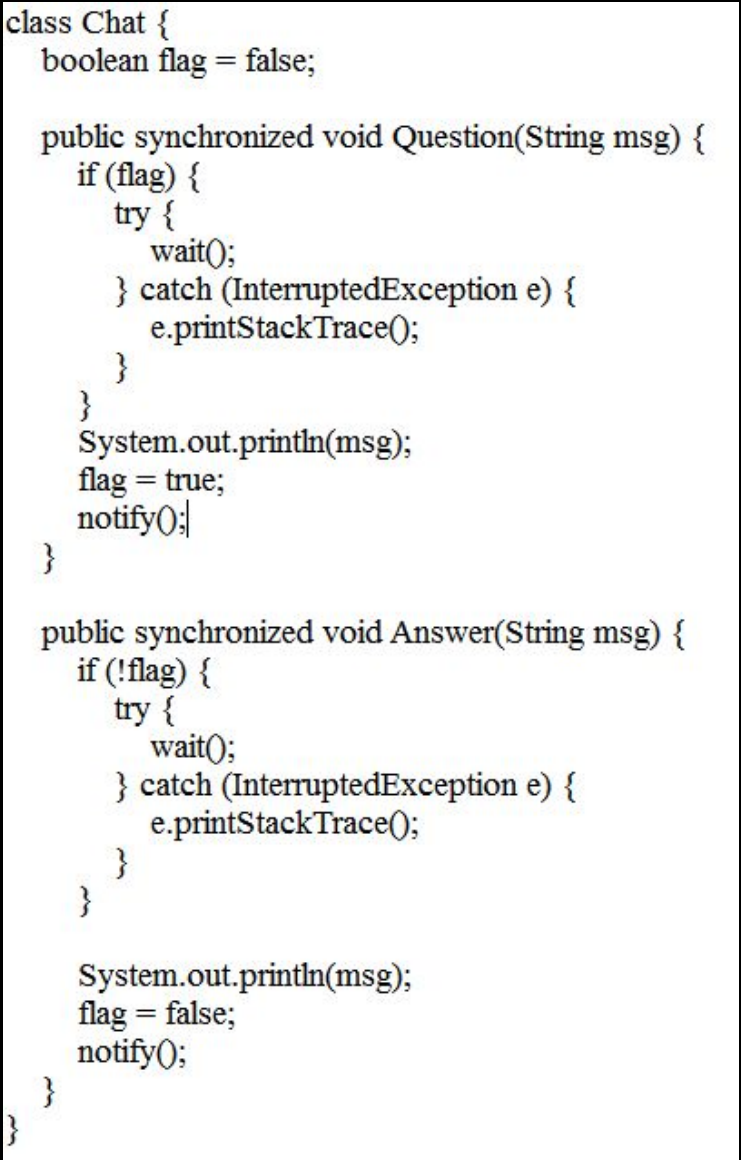
- Inter-thread communication ...
- These methods have been implemented as **final** methods in Object, so they are available in all the classes.
- All three methods can be called only from within a **synchronized** context.



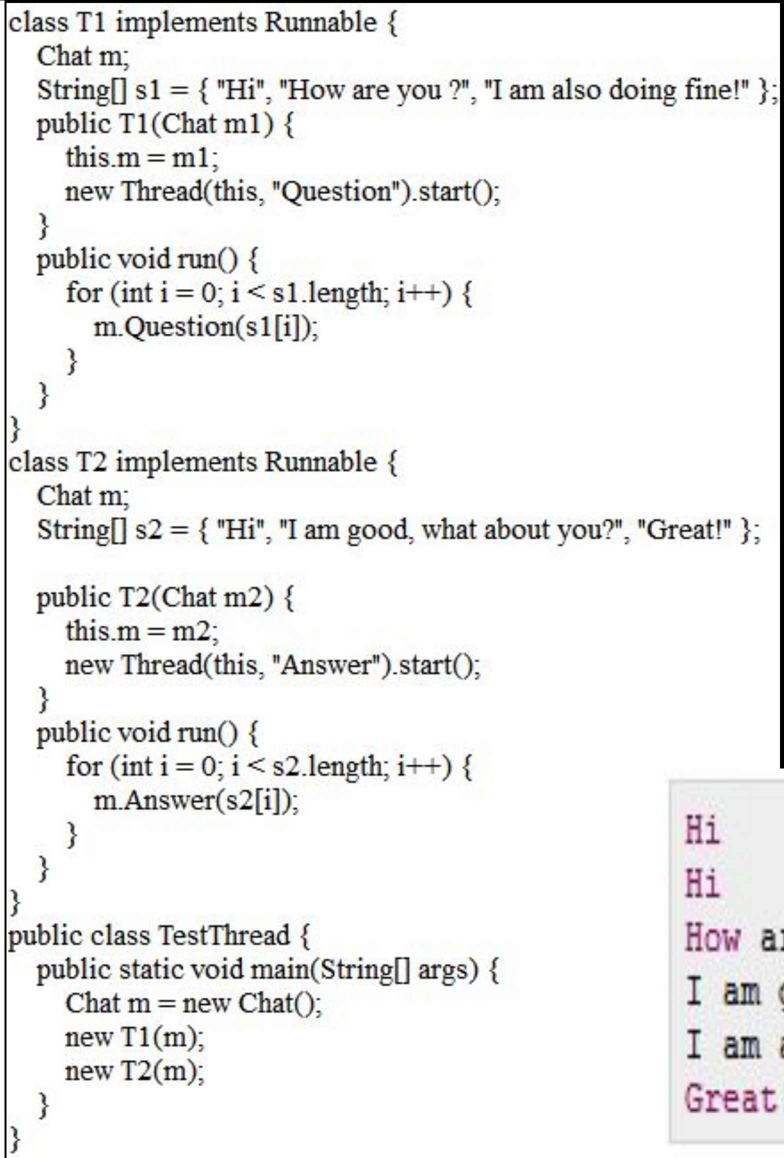
Methods with Description
public void wait() Causes the current thread to wait until another thread invokes the notify().
public void notify() Wakes up a single thread that is waiting on this object's monitor.
public void notifyAll() Wakes up all the threads that called wait( ) on the same object.

Multithreading

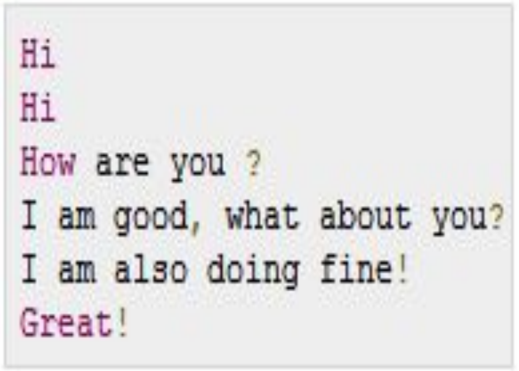
- Inter-thread communication ...



```
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();
   }
}
```



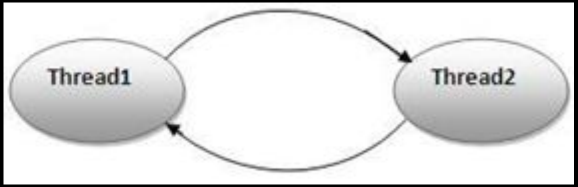
```
class T1 implements Runnable {
   Chat m;
   String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };
   public T1(Chat m1) {
      this.m = m1;
      new Thread(this, "Question").start();
   }
   public void run() {
      for (int i = 0; i < s1.length; i++) {
         m.Question(s1[i]);
      }
   }
}
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);
   }
}
```



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

Multithreading

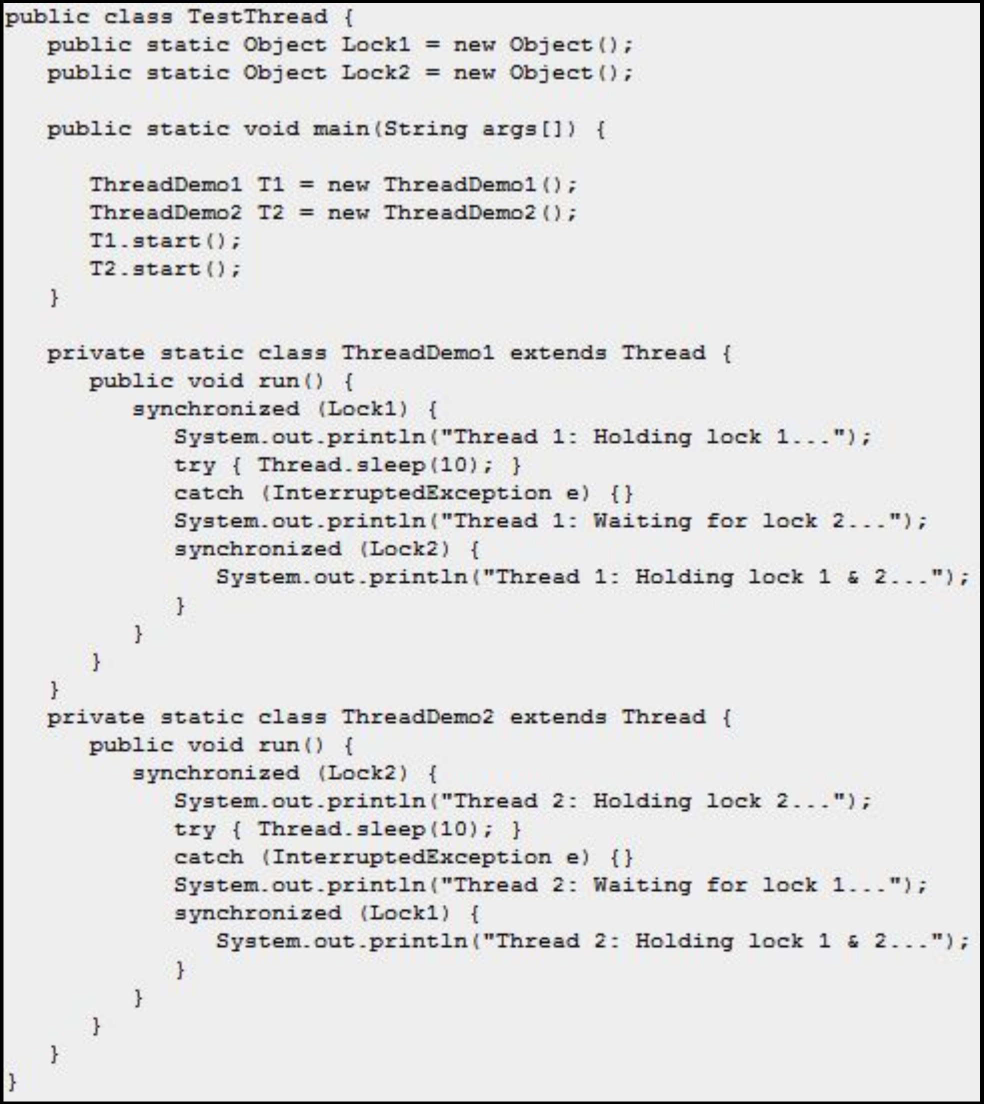
- Deadlocks for threads
- Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called 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.***

Multithreading

- Deadlocks for threads ...*Deadlock Situation*



```
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...");
            }
         }
      }
   }
}
```

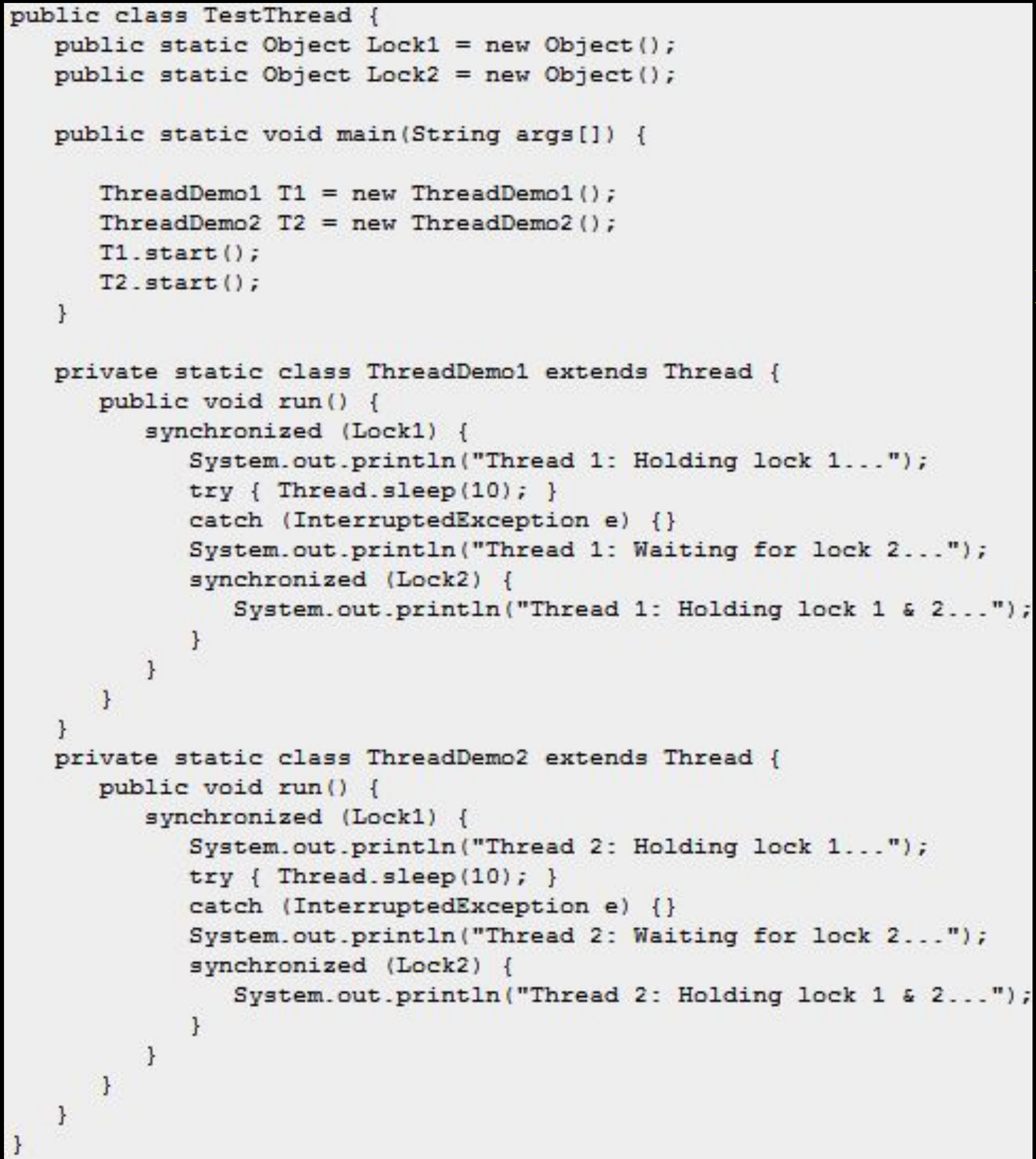
When you compile and execute above program, you find a deadlock situation and below is the output produced by the program:

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

The 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.

Multithreading

- Deadlocks for threads … *Solution*



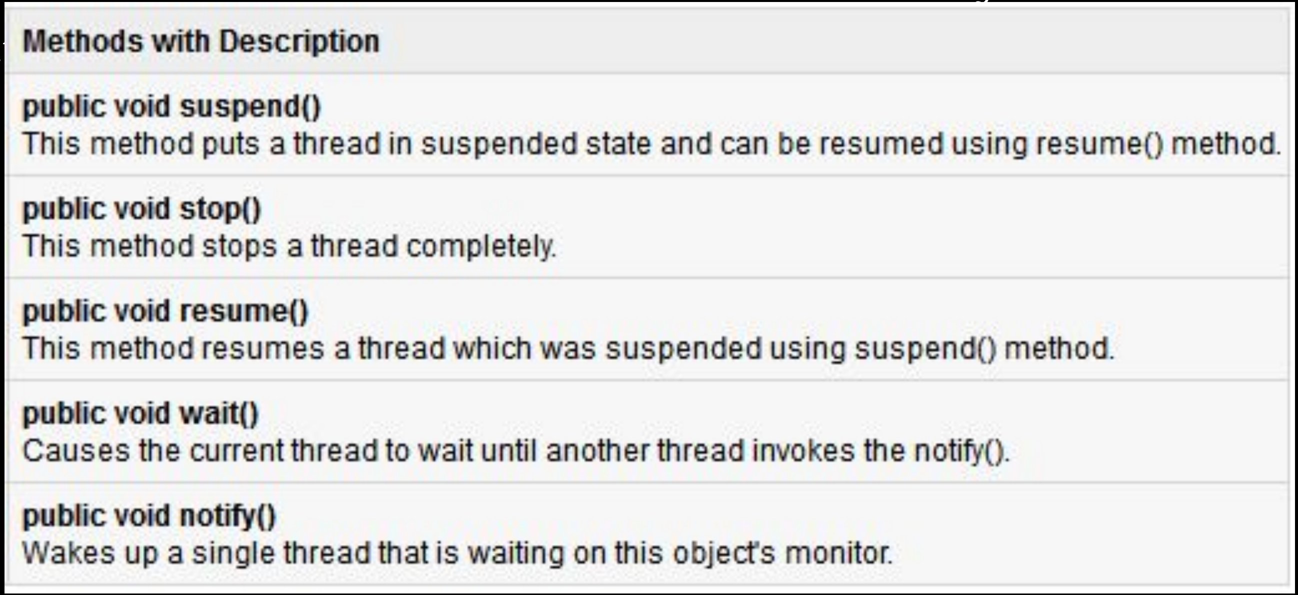
```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   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...
```

Multithreading

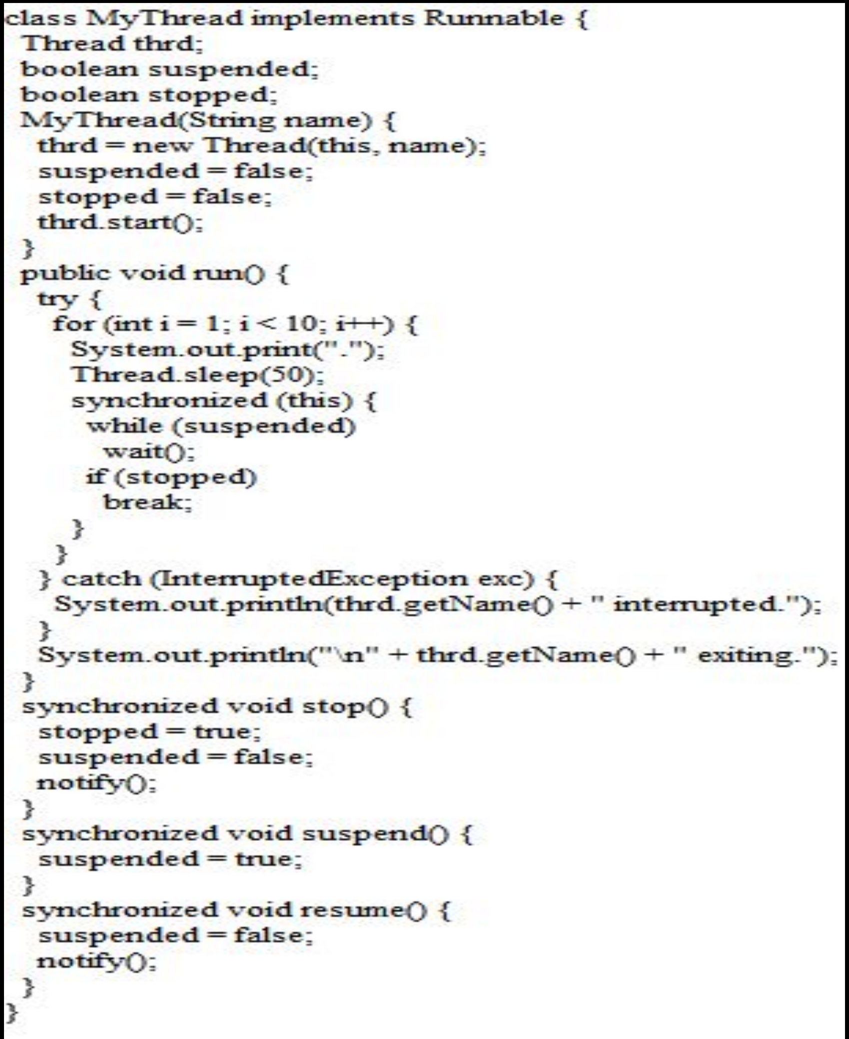
- Suspending & Resuming threads / Suspending, Resuming, and Stopping Threads
- Core Java provides a complete control over multithreaded program. You can develop a multithreaded program which can be suspended, resumed or stopped completely based on your requirements.
- There are various static methods which you can use on thread objec



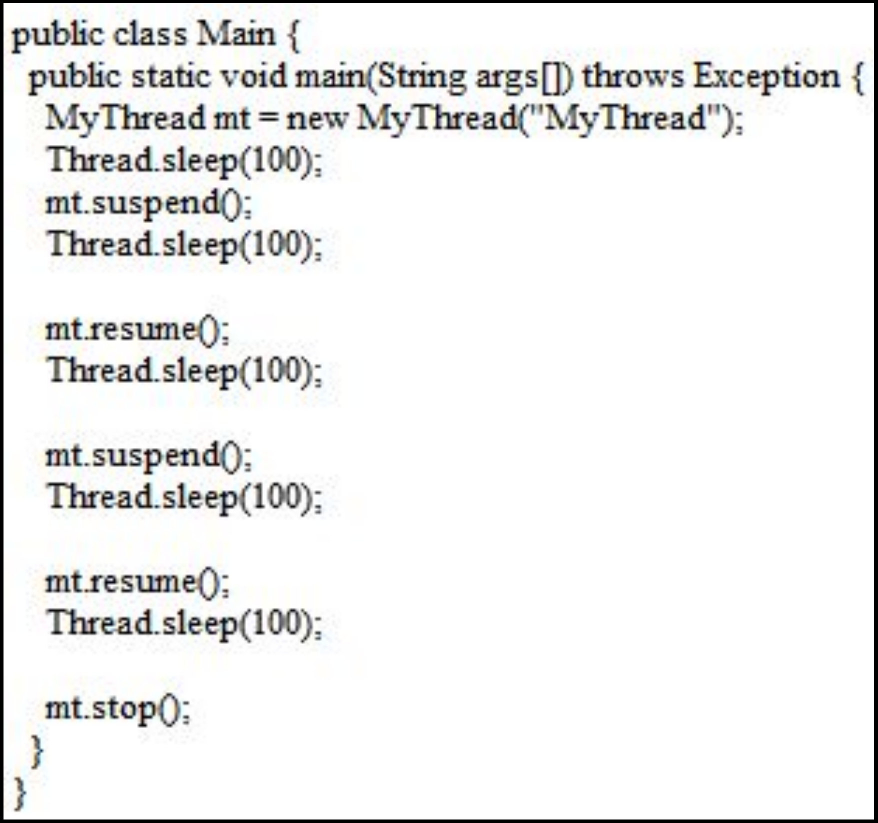
Methods with Description
public void suspend() This method puts a thread in suspended state and can be resumed using resume() method.
public void stop() This method stops a thread completely.
public void resume() This method resumes a thread which was suspended using suspend() method.
public void wait() Causes the current thread to wait until another thread invokes the notify().
public void notify() Wakes up a single thread that is waiting on this object's monitor.

Multithreading

- Suspending & Resuming threads / Suspending, Resuming, and Stopping Threads ...



```
class MyThread implements Runnable {
 Thread thrd;
 boolean suspended;
 boolean stopped;
 MyThread(String name) {
  thrd = new Thread(this, name);
  suspended = false;
  stopped = false;
  thrd.start();
 }
 public void run() {
  try {
   for (int i = 1; i < 10; i++) {
    System.out.print(".");
    Thread.sleep(50);
    synchronized (this) {
     while (suspended)
      wait();
     if (stopped)
      break;
    }
   }
  } catch (InterruptedException exc) {
   System.out.println(thrd.getName() + " interrupted.");
  }
  System.out.println("\n" + thrd.getName() + " exiting.");
 }
 synchronized void stop() {
  stopped = true;
  suspended = false;
  notify();
 }
 synchronized void suspend() {
  suspended = true;
 }
 synchronized void resume() {
  suspended = false;
  notify();
 }
}
```



```
public class Main {
 public static void main(String args[]) throws Exception {
  MyThread mt = new MyThread("MyThread");
  Thread.sleep(100);
  mt.suspend();
  Thread.sleep(100);

  mt.resume();
  Thread.sleep(100);

  mt.suspend();
  Thread.sleep(100);

  mt.resume();
  Thread.sleep(100);

  mt.stop();
 }
}
```

Multithreading

- Using Multithreading
- The key to utilizing Java’s multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.
- With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
- Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your

Multithreading

Additional Resources:

Multithreading-1

Multithreading-2

Multithreading-3