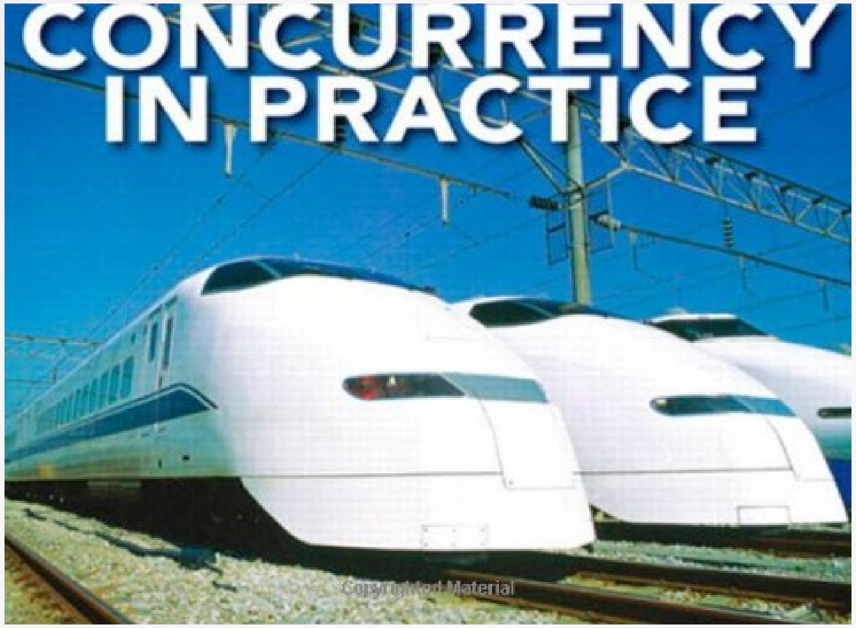
04-Java Multithreading



Join Google+ community

http://goo.gl/U7qVS

You can ask all your doubts, questions and queries by posting on this G+ community during/after webinar

Study Points

- Introduction
- Life Cycle of a Thread
- The Thread Class and the Runnable Interface
- The Main Thread
- Creating a Thread
- Implementing Runnable
- Extending Thread
- Creating Multiple Threads
- Using isAlive( ) and join( )
- Thread Priorities
- Synchronization
- Interthread Communication
- Deadlock

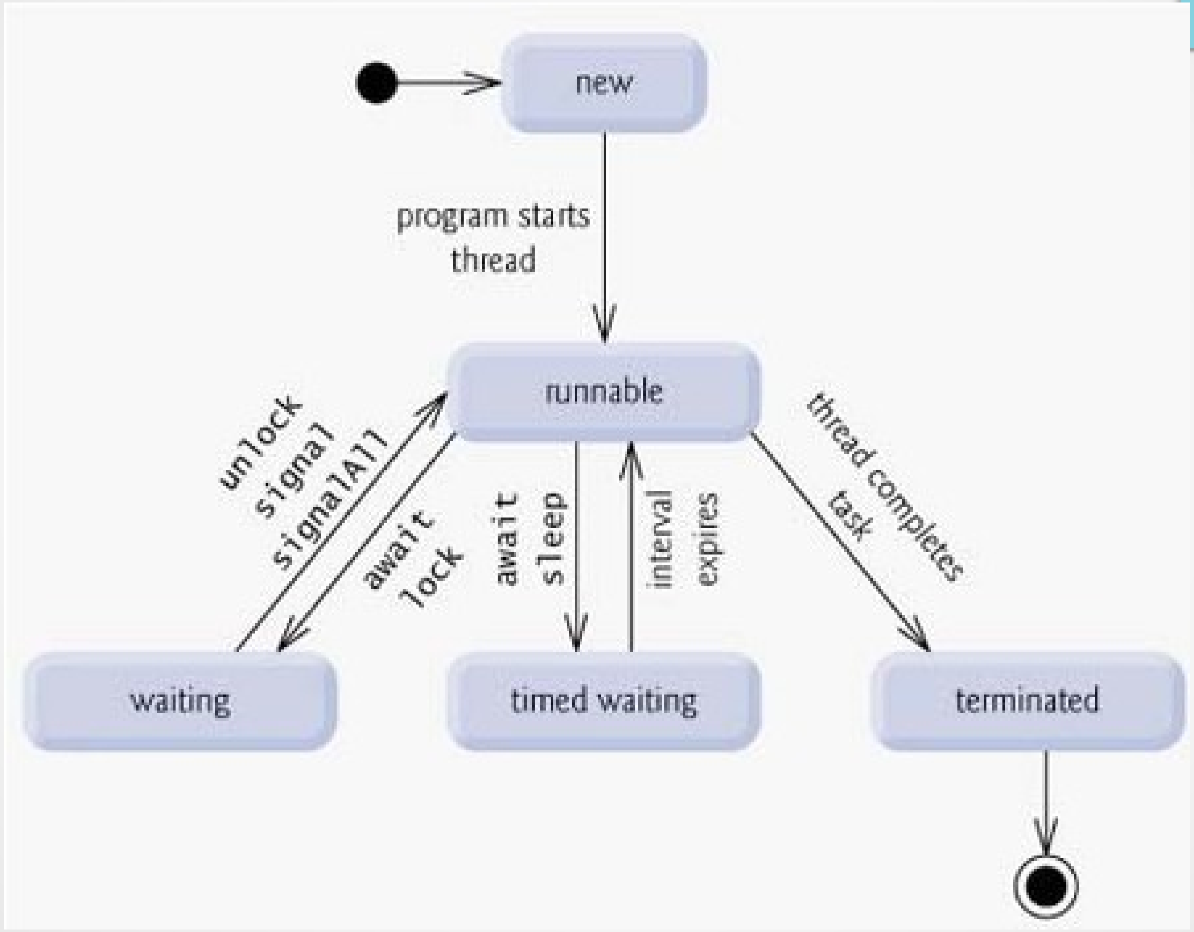
Introduction

- 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.
- Multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking:
  - Process-based
  - Thread-based
- Multitasking threads require less overhead than multitasking processes.
    - Process is
      - Heavyweight
      - Interprocess communication is expensive and limited.
      - Context switching from one process to another is also costly.
    - Multithreading is:
      - Threads are lightweight.
      - Interprocess communication is expensive and limited.
      - Maximum use of the CPU

Life Cycle of a Thread

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

Life Cycle of a Thread – Pictorial View



The Thread Class and the Runnable Interface:

- The Thread class defines several methods that help manage threads. Few most used methods are:
  - 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.
  - Yield: Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:
  - public void run( )

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.
- Thread.currentThread() 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.

```
// 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");
}}}
// Output
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
```

Creating a Thread

- In the most general sense, you create a thread by instantiating an object of type **Thread**.
- Java defines two ways in which this can be accomplished:
  - You can implement the **Runnable** interface.
  - You can extend the **Thread** class, itself.

Implementing Runnable

- Following is the example to implement the Runnable interface.

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

Implementing Runnable Continued…

- The easiest way to create a thread is to create a class that implements the **Runnable**
- interface. To implement **Runnable**, a class need only implement a single method called **run( )**,

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

Extending Thread

- The second way to create a thread is to create a new class that extends Thread.

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

Extending Thread Continued…

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

Creating Multiple Threads

- So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child 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.");
} }
```

Creating Multiple Threads Continued…

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

Creating Multiple Threads Continued…

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.
```

Using isAlive( ) and join( )

- Using isAlive( ) and join( ): As mentioned, often you will want the main thread to finish last. We should ensure that all child threads terminate prior to the main thread.

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

Using isAlive( ) and join( ) Continued…

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

Using isAlive( ) and join( ) Continued…

- Sample 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]
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
```

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.
- 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 called a **context switch.**
- 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 **final** variables within **Thread**.
  - final int getPriority( )
  - final void setPriority(int *level*)

Thread Priorities Continued…

```
// Demonstrate thread priorities.
class clicker implements Runnable {
        int 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();
        }
}
```

Thread Priorities Continued…

```
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);
        }
}
```

- The output of this program shows that higher-priority thread got approximately 90 percent of the CPU time.
  - Low-priority thread: 4408112
  - High-priority thread: 589626904

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.*
- Key to synchronization is the concept of the monitor (also called a *semaphore*).
- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor.
- These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Synchronization Continued…

```
// 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); } }
```

Synchronization Continued…

```
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]
]
]

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]
```

Interthread Communication

- Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.
  - **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
  - **notify( )** wakes up the first thread that called **wait( )** on the same object.
  - **notifyAll( )** wakes up all the threads that called **wait()** on the same object. The highest priority thread will run first.

Deadlock

- A special type of error that you need to avoid that relates specifically to multitasking is *deadlock,* which occurs when **two threads have a circular dependency on a pair of synchronized objects**.

```
// An example of deadlock.
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 to call B.last()");
        b.last();
        }
        synchronized void last() {
        System.out.println("Inside A.last");
        }
}
```

Deadlock Continued…

```
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 to call A.last()");
        a.last();
        }
        synchronized void last() {
        System.out.println("Inside A.last");
        }
}
```

Deadlock Continued…

```
class Deadlock implements Runnable {
        A a = new A();
        B b = new B();
        Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
        }
        public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
        }
        public static void main(String args[]) {
        new Deadlock();
        }
}

// When you run this program, you will see the output shown here:
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Thanks You