Multithreading in Java

Dr. Alekha Kumar Mishra

Introduction

- Two distinct types of multitasking:
  - Process-based
  - Thread-based.

Process vs Thread based multi-tasking

- A program is the smallest unit of code that can be dispatched by the scheduler.
- Process-based multitasking allows to run two or more programs concurrently.
  - For example, run the Java compiler at the same time that you are using a text editor.

Process vs Thread based multi-tasking (2)

- 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
- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.

Benefit of threads

- Threads are lightweight.
- Multitasking threads require less overhead than multitasking processes.
- 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

Rules for context-switch in Java

- A thread can voluntarily relinquish control.
  - 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.

Thread class and Runnable interface

- Java's multithreading system is built upon the **Thread** class and its companion interface, **Runnable**.
- Thread encapsulates a thread of execution. To create a new thread, your program will either extend Thread or implement the Runnable interface.

Thread class common methods

- **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.

The Main Thread

- When a Java program starts up, the main thread of the program begins execution.
- 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.
- Main thread can be controlled through a Thread object.
- The method **currentThread()** returns a reference to main thread (current thread)
  - static Thread currentThread( )
- System.out.println() prints a threat is the following format
  - [*name of the thread, its priority value, name of the group*]

The sleep() method

- The sleep() method suspend the execution of the thread from which it is called for the specified period of milliseconds.
  - static void sleep(long milliseconds) throws InterruptedException
- The second form
  - static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
- You can obtain and set the name of a thread by calling getName() and setName() method.
  - final void setName(String threadName)
  - final String getName()

Example

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

Creating thread by implementing Runnable

- A class using Runnable interface for thread only need to implement run() method
  - public void run( )
- The run( ) method define the code for a new thread.
- It can call other methods, use other classes, and declare variables, just like the main thread.
- Considered as the entry point for another, concurrent thread of execution within your program.
- The thread ends when run( ) returns.

Creating thread by implementing Runnable(2)

- Using Runnable, the class need to instantiate an object of type Thread using following Thread class constructor.
  - Thread(Runnable threadOb, String threadName)
- Here, threadOb is an instance of a class that implements the Runnable interface.
- The name of the new thread is specified by threadName.
- Once the new thread is created, it runs when start() method is called (declared within Thread).
- Actually, start( ) executes a call to run( ).

Example

```
class NewThread implements Runnable {
        Thread t;

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

Creating thread by Extending Thread class

- Create a class that extends Thread
- Then to create an instance of that class.
- The extending class must override the run() method.
- It must also call start( ) to begin execution of the new thread.

Example

```
class NewThread extends Thread {
        NewThread() {
                super("Demo Thread");
                System.out.println("Child thread: " + this);
                start(); // Start the 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.");
        }
}
```

Multiple threads example

```
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
	}

	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.");
	}
}
```

isAlive() & join()

- Often it is desired the main thread to finish last
- The sleep() method does not give any information when another thread has ended
- Thread class provides two ways to check whether a thread has finished.
- The **isAlive()** method
  - final boolean **isAlive**( )
  - It returns true if the thread upon which it is called is still running. Returns false otherwise
- The **join**() method,
  - final void join( ) throws InterruptedException
  - It waits until the thread on which it is called terminates.
  - Additional forms of join() allows to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Example

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

Thread Priorities

- The setPriority( ) method of **Thread**
  - 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. (1 and 10 respectively).
- To return a thread to default priority, specify NORM_PRIORITY, which is 5.
- The getPriority( ) method of **Thread**
  - final int getPriority( )

Example

```
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();
        }
}
```

```
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();
                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);
        }
}
```

Synchronization

- The access to a shared resource by two or more threads, ensuring only one thread access at a time is achieved using synchronization
- 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 (waiting state) until the first thread exits the monitor.
- A thread that owns a monitor can re-enter the same monitor if it so desires.

Synchronization using Java

- 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.

Problem without synchronization

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

Synchronizing the call() method

Synchronized method

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

Synchronized block

```
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);
    }
  }
}
```

Interthread Communication

- Java includes 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().
  - final void wait( ) throws InterruptedException
- **notify**() wakes up a thread that has called **wait**() on the same object.
  - final void notify( )
- **notifyAll**() wakes up all the threads that called wait() on the same object. One of the threads will be granted access.
  - final void notifyAll()

Further Reading

- The example of producer and consumer problem using wait(); and notify() methods.