Programming with Shared Memory

Java Threads and Synchronization Review

The following notes are based upon the Java tutorial at http://java.sun.com/docs/books/tutorial/essential/concurrency/

Additional information from Raj Buyya’s slides “Multithreaded Programming using Java Threads” http://www.buyya.com
Thread class

Each thread is an object of the **Thread** class.

(Java tutorial says: “Each thread is associated with an instance of the class **Thread**.”)

Java provides two basic ways to create a thread:

1. Define a class that is derived class of the class **Thread**.

2. Make your class implement the **Runnable** interface
Simplest way is:

1. Define a class that is derived class of the class Thread.
   - Object of this class is a thread.
   - Provide the method called **run** (which will override the inherited run method, which does nothing).
   - The **run** method defines the code for the thread.
   - Invoke the **start** method, which initiates the computation of the thread.
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        HelloThread myThread = new HelloThread();
        myThread.start();
    }
}
Simpler version if name of thread object not needed

```java
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

However, usually one does need the object by name to apply other thread methods.
The **Thread** class actually implements the interface called **Runnable**.

The **Runnable** interface defines the single method, **run**, meant to contain the code executed in the thread.

Alternate more powerful way to create threads:

2. Make your class explicitly implement the **Runnable** interface
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {

        HelloRunnable myThread = new HelloRunnable();  // Runnable object

        Thread tr = new Thread(myThread);  // Create Thread

        tr.start();  // Start thread and execute run method
    }
}
Slightly simplified version:

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread tr = new Thread(new HelloRunnable())
        tr.start();
    }
}
```

Even simpler if thread object name not needed:

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```
Runnable object can subclass a class other than Thread, i.e.:

```java
public class MyRunnable extends SomeClass implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Note: both the Thread class and the Runnable interface are part of the standard Java libraries (java.lang package)
A Program with Three Java Threads using 1st method

class A extends Thread {
    public void run() {
        for(int i=1;i<=5;i++) System.out.println("\t From ThreadA: i = "+i);
        System.out.println("Exit from A");
    }
}

class B extends Thread {
    public void run() {
        for(int j=1;j<=5;j++) System.out.println("\t From ThreadB: j = "+j);
        System.out.println("Exit from B");
    }
}

class C extends Thread {
    public void run() {
        for(int k=1;k<=5;k++) System.out.println("\t From ThreadC: k = "+k);
        System.out.println("Exit from C");
    }
}

class ThreadTest {
    public static void main(String args[]) {
        new A().start();
        new B().start();
        new C().start();
    }
}

Based on Raj Buyya’s slides
Sample Output

From ThreadA: i = 1
From ThreadA: i = 2
From ThreadA: i = 3
From ThreadA: i = 4
From ThreadA: i = 5
Exit from A

From ThreadC: k = 1
From ThreadC: k = 2
From ThreadC: k = 3
From ThreadC: k = 4
From ThreadC: k = 5
Exit from C

Exit from B

Run 2

From ThreadA: i = 1
From ThreadA: i = 2
From ThreadA: i = 3
From ThreadA: i = 4
From ThreadA: i = 5
Exit from A

From ThreadC: k = 1
From ThreadC: k = 2
From ThreadC: k = 3
From ThreadC: k = 4
From ThreadC: k = 5
Exit from C

Exit from B
Exit from A
Thread Priority

• In Java, each thread assigned priority, which affects the order in which it is scheduled for running.

• Threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.

• Java allows users to change priority:

  • `ThreadName.setPriority(intNumber)`

    – MIN_PRIORITY = 1
    – NORM_PRIORITY=5
    – MAX_PRIORITY=10
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread started");
        for(int i=1;i<=4;i++)
            System.out.println("\t From Thread: i= "+i);
    }
}

class ThreadPriority {
    public static void main(String args[]) {
        MyThread threadA=new MyThread();
        MyThread threadB=new MyThread();
        MyThread threadC=new MyThread();
        threadA.setPriority(Thread.MIN_PRIORITY);
        threadB.setPriority(Thread.NORM_PRIORITY);
        threadC.setPriority(Thread.MAX_PRIORITY);
        threadA.start();
        System.out.println("Started Thread A");
        threadB.start();
        System.out.println("Started Thread B");
        threadC.start();
        System.out.println("Started Thread C");
    }
}
Thread class

Various instance and class methods, setters and getters:

• Class methods:
  • `sleep()`
  •...

• Instance methods:
  • `destroy()`
  • `interrupt()`
  • `join()`
  • `start()`
  •...

• Deprecated methods (unsafe and can cause deadlock)
  • `resume()`, `stop()` `suspend()`
Thread.sleep causes the current thread to suspend execution for a specified period.

Example
Sleep to print messages at four-second intervals:

```java
public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        for (int i = 0; i < importantInfo.length; i++) {
            Thread.sleep(4000); //Pause for 4 seconds
            System.out.println(importantInfo[i]); //Print a message
        }
    }
}
```

exception that sleep throws when another thread interrupts current thread while sleep is active. Not caught in sample code.
Java Synchronization

Java provides **Synchronized** keyword to methods that cause only one invocation of a synchronized method on the same object at a time.

**Example**

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```
Implementation of Java synchronization

Every object has an intrinsic lock associated with it.

A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it is done with them.
Example using synchronized methods

On-line banking

Several entities can access account potentially simultaneously (maybe a joint account, maybe automatic debits, …)

Suppose three entities each trying to perform an operation, either:

• deposit()
• withdraw()
• enquire()
Create three threads, one for each entities

class InternetBankingSystem {
    public static void main(String [] args ) {
        Account accountObject = new Account ();
        Thread t1 = new Thread(new MyThread(accountObject));
        Thread t2 = new Thread(new YourThread(accountObject));
        Thread t3 = new Thread(new HerThread(accountObject));

        t1.start();
        t2.start();
        t3.start();

        // DO some other operation
    } // end main()
} // end main()
class MyThread implements Runnable {
    Account account;
    public MyThread (Account s) { account = s;}
    public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable {
    Account account;
    public YourThread (Account s) { account = s;}
    public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable {
    Account account;
    public HerThread (Account s) { account = s; }
    public void run() { account.enquire(); }
} // end class HerThread
Synchronized account methods

class Account {
    int balance;
    // if 'synchronized' is removed, outcome unpredictable

    public synchronized void deposit() {
        balance += deposit_amount;
    }

    public synchronized void withdraw() {
        balance -= deposit_amount;
    }

    public synchronized void enquire() {
        display balance.
    }
}
Synchronized Statements

Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

Uses construct ion:

```java
synchronized ( expression ) {
    statements
}
```

Evaluate to an object or an array. Used to identify lock.

“critical section”
Synchronized Statements

Example

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```
atomic action

An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

Read/writes can be declared atomic with the `volatile` keyword, e.g.

```
private volatile int x;
```

Sometimes can be more efficient than synchronized methods.
Coordinating threads
Wait/notify mechanism

Sometimes need a thread to stop running and wait for an event before continuing.

wait() and notify() methods are methods of class Object.

Every object can maintain a list of waiting threads.

wait() When a thread calls wait() method of an object, any locks the thread holds are temporarily released and thread added to list of waiting threads for that object and stops running.

notify() When another thread calls notify() method on the same object, object wakes up one of the waiting threads and allows it to continue.
Join

Sometimes one thread needs to stop and wait for another thread to complete.

join() -- waits for a thread to die, i.e. thr1.join() waits for thread thr1 to die.

Calling return() from the run method implicitly causes the thread to exit.
Java Example using a monitor

```java
public class Adder {
    public int[] array;
    private int sum = 0;
    private int index = 0;
    private int number_of_threads = 10;
    private int threads_quit;

    public Adder() {
        threads_quit = 0;
        array = new int[1000];
        initializeArray();
        startThreads();
    }

    public synchronized int getNextIndex() {
        if (index < 1000) return (index++);
        else return (-1);
    }
```
public synchronized void addPartialSum(int partial_sum)
{
    sum = sum + partial_sum;
    if(++threads_quit == number_of_threads)
        System.out.println("The sum of the numbers is "+sum);
}

private void initializeArray()
{
    int i;
    for(i = 0;i < 1000;i++) array[i] = i;
}

public void startThreads()
{
    int i = 0;
    for(i = 0;i < 10;i++)
    {
        AdderThread at = new AdderThread(this,i);
        at.start();
    }
}

{
    Adder a = new Adder();
}
public static void main(String args[])
{
    class AdderThread extends Thread
    {
        int partial_sum = 0;
        Adder parent;
        int number;
        public AdderThread(Adder parent, int number)
        {
            this.parent = parent;
            this.number = number;
        }

        public void run()
        {
            int index = 0;
            while (index != -1) {
                partial_sum = partial_sum + parent.array[index];
                index = parent.getNextIndex();
            }
            System.out.println("Partial sum from thread " + number + " is " + partial_sum);
            parent.addPartialSum(partial_sum);
        }
    }
}
More information

http://java.sun.com/docs/books/tutorial/essential/concurrency/