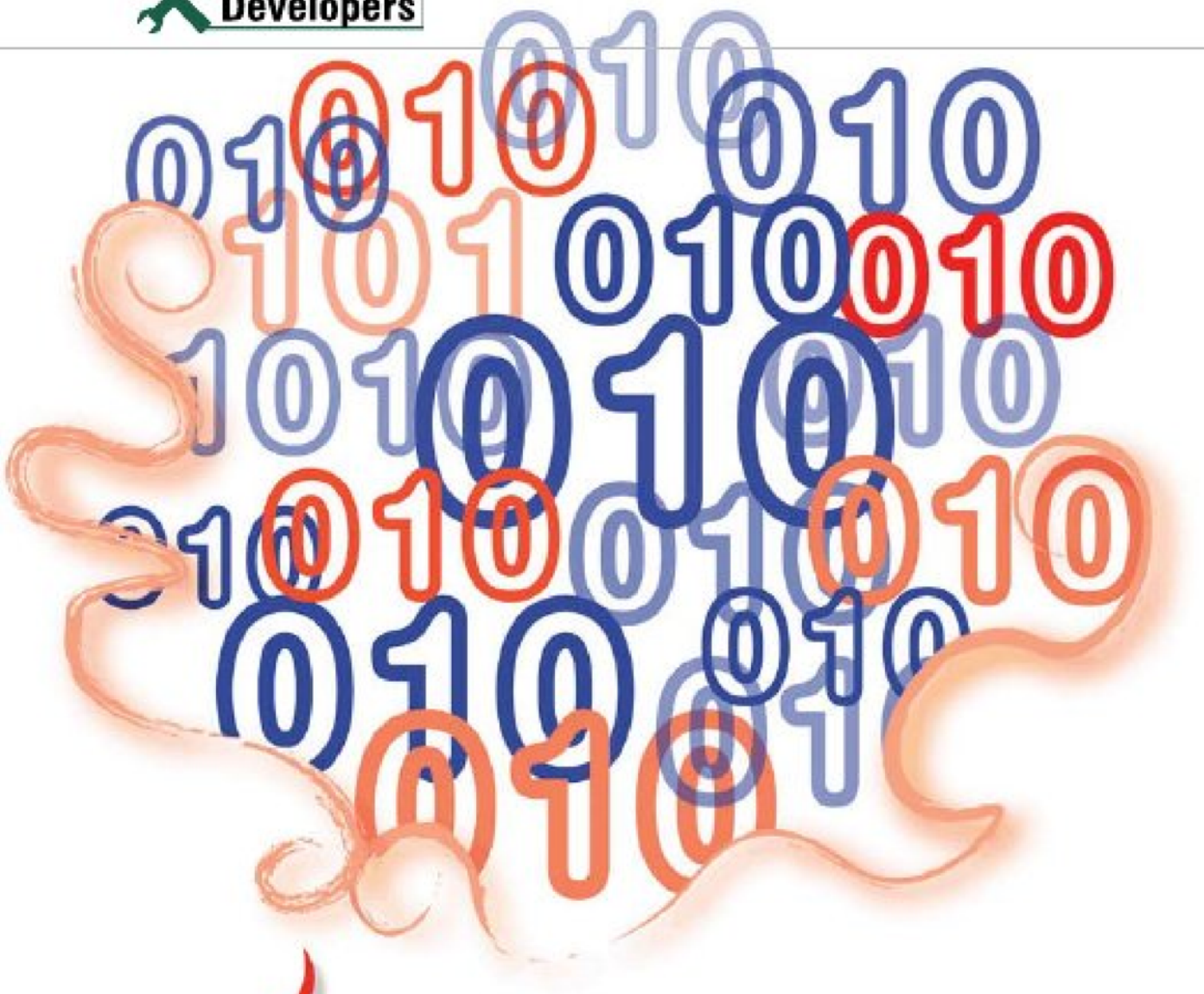

Let's get the basics of multi-threaded programming in Java, and then write some simple programs.

An Introduction to Multi-threading

In a typical application like a word processor, many tasks need to be executed in parallel—for instance, responding to the user, checking spellings, formatting, displaying, saving, etc. These tasks can run as different processes, sharing the resources using inter-process communication. Multi-processing is heavyweight and costly in terms of the system calls made, and requires a process switch.

An easier and cheaper alternative is to use threads. Multiple threads can run in the context of the same process and hence share the same resources. A context switch is enough to switch between threads, whereas a costly process switch is needed for processes.

Java has support for concurrent programming and has support for multi-threading in both language and library levels. In this article, we'll cover the basics of multi-threaded programming in Java and write some simple programs. We'll also look at some difficult problems that can crop up when we write multi-threaded code.

The basics

A Java thread can be created in two ways: by implementing the *Runnable* interface or by extending the *Thread* class. Both have a method called *run*. This method will be called by the runtime when a thread starts executing.

A thread can be created by invoking the start method on a *Thread* or its derived objects. This in turn calls the run method and keeps the thread in running state. Now let us look at a simple example using threads.

```
class MyThread extends Thread {
   public void run() {
            System.out.println("This is new thread");
     }

   public static void main(String args[]) throws Exception {
            MyThread mt = new MyThread();
            mt.start();
            System.out.println("This is the main Thread");
     }
}
```

This program prints:

```
This is the main Thread
This is new thread
```

In this example, the *MyThread* class extends the *Thread* class. We need to override the run method in this class. This run method will be called when the thread runs. In the main function, we create a new thread and start it. The program prints messages from both the main thread and the mt thread (that we

```
class MyThread extends Thread {
public void loop() {
for(int i = 0; i < 3; i++) {
System.out.println("In thread " +
Thread.currentThread().getName() +
"; iteration " + i);
try {
Thread.sleep((int)Math.random());
}
catch(InterruptedException ie) {
ie.printStackTrace();
}
}
}

public void run() {
System.out.println("This is new thread");
   this.loop();
}

public static void main(String args[]) throws Exception {
MyThread mt = new MyThread();
   mt.start();
   System.out.println("This is the main Thread");
   mt.loop();
}
}
```

In a sample run, the output was:

```
This is the main Thread
In thread main; iteration 0
This is new thread
In thread Thread-0; iteration 0
In thread Thread-0; iteration 1
In thread main; iteration 1
In thread Thread-0; iteration 2
In thread main; iteration 2
```

In this program, we have a loop method that has a *for* loop that has three iterations. In the *for* loop, we print the name of the thread and the iteration number. We can set and get the name of the thread using *setName* and *getName* methods, respectively. If we haven't set the name of the thread explicitly, the JVM gives a default name ('main', 'Thread-0', 'Thread-1', etc.). After printing this information, we force the current thread to sleep for

```
class Counter {
public static long count = 0;
}

class UseCounter implements Runnable {
public void run() {
for (int i = 0; i < 3; i++) {
         Counter.count++;
          System.out.print(Counter.count + "  ");
   }
}
}

public class DataRace {
public static void main(String args[]) {
UseCounter c = new UseCounter();
   Thread t1 = new Thread(c);
   Thread t2 = new Thread(c);
   Thread t3 = new Thread(c);
   t1.start();
   t2.start();
   t3.start();
}
}
```

In this program, we have a *Counter* class, which has a static variable count. The *UseCounter* class simply increments the count and prints the count three times in a *for* loop. We create three threads in the main function in the *DataRace* class and start it. We expect the program to print 1 to 9 sequentially as the threads run and increment the counters. However, when we run this program, it does print 9 integer values, but the output looks like garbage! In a sample run, I got these values:

```
3  3  5  6  3  7  8  4  9
```

Note that the values will usually be different every time you run this program. What is the problem?

The expression *Counter.count++* is a write operation and the next *System.out.print* statement has a read operation for *Counter.count*. When the three threads execute, each of them has a local copy of the value

is printed.

Technically, this program has a 'data race'. To avoid this problem, we need to ensure that the write and read operations are done together ('atomically') by a single thread. How do we ensure that? It is by acquiring a lock on the object. Only a single thread can acquire a lock on an object at a time, and only that thread can execute the code protected by the lock (that code segment is known as a 'critical section'). Until then the other threads have to wait. Internally, this is implemented with monitors and the process is called as locking and unlocking.

Thread synchronisation

Java has a keyword synchronised to acquire a lock. Lock is not obtained on code, but on a resource. Any object (not primitive types but of reference type) can be used to acquire a lock.

Here is an improved version of the code that provides synchronised access to *Counter.count*, and does both read and write operations on that in a critical section. For that, only the run method needs to be changed as follows:

```
public void run() {
for (int i = 0; i < 3; i++) {
         synchronized(this) {
                   Counter.count++;
                System.out.print(Counter.count + "  ");
}
}
}
```

The program now prints the expected output correctly:

```
1  2  3  4  5  6  7  8  9
```

In the run method, we acquire lock on the implicit this pointer before doing both reading and writing to *Counter.count.*

In this case, run is a non-static method. What about static methods, as they do not have this pointer? You can obtain a lock for a static method, and the lock is obtained for the class instance and not the object. In fact, in this example, *Counter.count* is a static member, so we might as well have acquired a lock on *Counter.class* for modifying and reading *Counter.count*.

Deadlocks

Obtaining and using locks is tricky and can lead to lots of problems, one of which (a common one) is known as

```
class Balls {
public static long balls = 0;
}

class Runs {
public static long runs = 0;
}

class Counter implements Runnable {
      public void IncrementBallAfterRun() {
         synchronized(Runs.class) {
                  synchronized(Balls.class) {
                  Runs.runs++;
                        Balls.balls++;
}
}
}

public void IncrementRunAfterBall() {
synchronized(Balls.class) {
                  synchronized(Runs.class) {
                  Balls.balls++;
                        Runs.runs++;
}
}
}

      public void run() {
IncrementBallAfterRun();
IncrementRunAfterBall();
}
}

public class Dead {
      public static void main(String args[]) {
         Counter c = new Counter();
            Thread t1 = new Thread(c);
            Thread t2 = new Thread(c);
            t1.start();
            t2.start();
}
}
```

in the opposite order. The run method calls these two methods consecutively. The main method in the *Dead* class creates two threads and starts them.

When the threads t1 and t2 execute, they will invoke the methods *IncrementBallAfterRun* and *IncrementRunAfterBall*. In these methods, locks are obtained in the opposite order. It might happen that t1 acquires a lock on *Runs.class* and waits to acquire a lock on *Balls.class*. Meanwhile, t2 might have acquired *Balls.class* and waits to acquire a lock on *Runs.class*. So, this program can lead to a deadlock. It is not assured that this program will lead to a deadlock every time you execute this program. Why? We never know the sequence in which threads execute, and the order in which locks are acquired and released. For this reason, the problems are said to be 'non-deterministic' and such problems cannot be reproduced consistently.

We will not look at how to resolve this deadlock problem in this article. The objective is to introduce you to problems like these that can occur in multi-threaded programs.

In this article, we explored the basics of writing multi-threaded programming in Java. We can create classes that are capable of multi-threading by implementing a *Runnable* interface or by extending a *Thread* class. Concurrent reads and writes to resources can lead to 'data races'. Java provides thread synchronisation features for protected access to shared resources. We need to use locks carefully. Incorrect usage of lock can lead to problems like deadlocks, livelocks or lock starvation, which are very difficult to detect and fix. END

By: S G Ganesh *is a research engineer in Siemens (Corporate Technology), Bangalore. His latest book is '60 Tips on Object Oriented Programming' published by Tata McGraw-Hill in December 2007. You can reach him at sgganesh@gmail.com*