Concurrent Programming Design

In this chapter, we will cover:

- Using immutable objects when possible
- Avoiding deadlocks by ordering the locks
- Using atomic variables instead of synchronization
- Holding locks as short a time as possible
- Delegating to executors the management of threads
- Using concurrent data structures instead of programming yourself
- Taking precautions using lazy initialization
- Using Fork/Join framework instead of executors
- Avoiding the use of blocking operations inside a lock
- Avoiding the use of deprecated methods
- Using executors instead of thread groups

Introduction

Implementing a concurrent application is a difficult task. You have more than one thread in an execution at a time and all of them share resources, such as files, memory, objects, and so on. You have to be very careful with the design decisions you take. A bad decision can determine that your program has poor performance or simply provoke data inconsistency situations.

In this chapter, we include some advices to help you to take correct design decisions that make your concurrent application better.

Using immutable objects when possible

When you develop an application in Java using object-oriented programming, you create some classes formed by attributes and methods. The methods of a class determine the operations that you can do with the class. The attributes store the data that define the object. Normally, in each class, you implement some methods to establish the value of the attributes. Also, objects change as the application runs and you use those methods to change the value of their attributes.

When you develop a concurrent application, you have to pay special attention to the objects shared by more than one `Thread`. You must use a synchronization mechanism to protect the access to that objects. If you don't use it, you may have data inconsistency problems in your application.

There are special kinds of objects that you can implement when you work with concurrent applications. They are called **immutable objects** and their main characteristic is that they can't be modified after they have been created. If you need to change an object, you must create a new one instead of changing the values of the attributes of the object.

This mechanism presents the following advantages when you use it in concurrent applications:

- These objects cannot be modified by more than one thread at a time, so you won't need to use any synchronization mechanism to protect the access to their attributes.
- You won't have any inconsistency data problems. As the attributes of these objects won't be modified, you will always have access to a coherent copy of this data.

The only drawback of this approach is the overhead that may affect the fact of creating objects instead of modifying existing objects.

The Java language provides some immutable classes such as the `String` class. When you have a `String` object and you assign to it a new value, you are creating a new `String` object instead of modifying the old value of the object.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement an immutable class, so you can have immutable objects in your application:

1. Mark the class as `final`. It may not be extended by another class.

2. All the attributes must be `final` and `private`. You can assign a value to an attribute only once.
3. Don't provide methods that can assign a value to an attribute. The attributes must be initialized in the constructor of the class.

How it works...

If you want to implement a class that stores the first name and the last name of a person, you normally would implement something like the following:

```
public class PersonMutable {
  private String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

}
```

You can convert this class into an immutable class following the rules explained earlier. Following is the result:

```
public final class PersonImmutable {

  final private String firstName;
  final private String lastName;

  public PersonImmutable (String firstName, String lastName) {
    this.firstName=firstName;
    this.lastName=lastName;
  }
```

```
    public String getFirstName() {
      return firstName;
    }

    public String getLastName() {
      return lastName;
    }
  }
```

Basically, you have followed the basic principles of the immutable classes, which are as follows:

- The class is marked as `final`.
- The attributes are marked as `final` and `private`.
- The value of the attributes can only be established in the constructor of the class. The rest of its methods returns the value of an attribute, but don't modify them.

There's more...

Immutable objects can't always be used. Analyze each class of your application to decide if you can implement them as immutable objects or not. If you can't implement a class as an immutable class and their objects are shared by more than one thread, you must use a synchronization mechanism to protect the access to the attributes of the class.

See also

- The *Using atomic variables instead of synchronization* recipe in *Appendix*, *Concurrent Programming Design*

Avoiding deadlocks by ordering locks

When you need to acquire more than one lock in the methods of your application, you must be very careful with the order in which you get the control of your locks. A bad choice can lead to a deadlock situation.

In this recipe, you will implement an example of a deadlock situation and then you will learn how to solve it.

How to do it...

Follow these steps to implement the example:

1. Create a class named `BadLocks` with two methods named `operation1()` and `operation2()`.

```
public class BadLocks {

  private Lock lock1, lock2;

  public BadLocks(Lock lock1, Lock lock2) {
    this.lock1=lock1;
    this.lock2=lock2;
  }

  public void operation1(){
    lock1.lock();
    lock2.lock();

    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      lock2.unlock();
      lock1.unlock();
    }
  }

  public void operation2(){
    lock2.lock();
    lock1.lock();

    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      lock1.unlock();
      lock2.unlock();
    }
  }

}
```

2. Analyze the previous code. If a thread calls to the `operation1()` method and another thread calls to the `operation2()` method, you can have a deadlock. If the `operation1()` method executes its first sentence and the `operation2()` method executes its first sentence at the same time, you will have the `operation1()` method waiting to get the control of the `lock2` Lock and the `operation2()` method waiting to get the control of the `lock1` Lock. Now, you have a deadlock situation.
3. To solve the previous situation, you can follow this rule:
    - If you have to get the control of more than one lock in different operations, try to lock them in the same order in all methods.
    - Then, release them in the inverse order and encapsulate the locks and their unlocks in a single class, so you don't have the synchronization-related code distributed throughout the code.

How it works...

Using this rule, you will avoid deadlock situations. For example, in the case presented earlier, you can change the `operation2()` method to get first the `lock1` Lock and then the `lock2` Lock. Now, if the `operation1()` method executes its first sentence and the `operation2()` method executes its first sentence, one of them will be blocked waiting for the `lock1` Lock and the other will get the `lock1`, `lock2` Locks, and will do its operation. After that, the blocked thread will get the `lock1` and `lock2` Locks and will do its operation.

There's more...

You can find a situation where a requirement prevents you to get the locks in the same order in all the operations. In this situation, you can use the `tryLock()` method of the `Lock` class. This method returns a `Boolean` value to indicate if you have or not the control of the lock. You can try to get all the locks that you need to get to do the operation using the `tryLock()` method. If you can't get the control of one of the locks, you must release all the locks that you got and start the operation again.

See also

- The *Holding locks as short a time as possible* recipe in *Appendix*, *Concurrent Programming Design*

Using atomic variables instead of synchronization

When you have to share data between multiple threads, you have to protect the access to that data using a synchronization mechanism. You can use the `synchronized` keyword in the declaration of the method that modifies the data, so only one thread can modify the data at a time. Another possibility is the utilization of a `Lock` class to create a critical section with the instructions that modify the data.

Since Version 5, Java includes the atomic variables. When a thread is doing an operation with an atomic variable, the implementation of the class includes a mechanism to check that the operation is done in one step. Basically, the operation gets the value of the variable, changes the value in a local variable, and then tries to change the old value for the new one. If the old value is still the same, it does the change. If not, the method begins the operation again. Java provides the following types of atomic variables:

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicLong`
- `AtomicReference`

Java's atomic variables offer a better performance than solutions based on synchronization mechanisms. Almost all the classes in the `java.util.concurrent` package use atomic variables instead of synchronization. In this recipe, you will develop an example that shows how an atomic attribute has better performance than using synchronization.

Getting ready...

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `TaskAtomic` and specify that it implements the `Runnable` interface.

   ```
   public class TaskAtomic implements Runnable {
   ```

2. Declare a private `AtomicInteger` attribute named `number`.

   ```
     private AtomicInteger number;
   ```

3. Implement the constructor of the class to initialize its attributes.

   ```
   public TaskAtomic () {
     this.number=new AtomicInteger();
   }
   ```

4. Implement the `run()` method. In a loop with 1,000,000 steps, assign to the atomic attribute the number of the step as a value using the `set()` method.

   ```
    @Override
   public void run() {
     for (int i=0; i<1000000; i++) {
       number.set(i);
     }
   }
   ```

5. Create a class named `TaskLock` and specify that it implements the `Runnable` interface.

   ```
   public class TaskLock implements Runnable {
   ```

6. Declare a private `int` attribute named `number` and a private `Lock` attribute named `lock`.

   ```
     private Lock lock;
     private int number;
   ```

7. Implement the constructor of the class to initialize its attributes.

   ```
     public TaskLock() {
       this.lock=new ReentrantLock();
     }
   ```

8. Implement the `run()` method. In a loop with 1,000,000 steps, assign to the integer attribute the number of the step. You have to get the lock before the assignment and release it after.

   ```
      @Override
     public void run() {
       for (int i=0; i<1000000; i++) {
         lock.lock();
         number=i;
         lock.unlock();
       }

     }
   ```

9. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

10. Create a `TaskAtomic` object named `atomicTask`.

```
    TaskAtomic atomicTask=new TaskAtomic();
```

11. Create a `TaskLock` object named `lockTask`.

```
    TaskLock lockTask=new TaskLock();
```

12. Declare a number of threads and create an array of `Thread` objects to store that number of threads.

```
    int numberThreads=50;
    Thread threads[]=new Thread[numberThreads];
    Date begin, end;
```

13. Launch the specified number of threads to execute the `TaskLock` object. Calculate and write in the console its execution time.

```
    begin=new Date();
    for (int i=0; i<numberThreads; i++) {
      threads[i]=new Thread(lockTask);
      threads[i].start();
    }

    for (int i=0; i<numberThreads; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    end=new Date();

    System.out.printf("Main: Lock results: %d\n",(end.getTime()-
begin.getTime()));
```

14. Launch the specified number of threads to execute the `TaskAtomic` object. Calculate and write in the console its execution time.

```
    begin=new Date();
    for (int i=0; i<numberThreads; i++) {
      threads[i]=new Thread(atomicTask);
      threads[i].start();
    }
```

```
        for (int i=0; i<numberThreads; i++) {
          try {
            threads[i].join();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
        end=new Date();

        System.out.printf("Main: Atomic results: %d\n",(end.getTime()-
    begin.getTime()));
```

How it works...

When you execute the example, you will see how the execution time of the `TaskAtomic` tasks that use atomic variables are always better than the `TaskLock` tasks that use locks. You will obtain a similar result if you use the `synchronized` keyword instead of locks.

The conclusion of this recipe is that the utilization of atomic variables will give you better performance than other synchronization methods. If you don't have an atomic type that fits your needs, maybe you can try to implement your own atomic type.

See also

- The *Implementing your own atomic object* recipe in *Appendix*, *Customizing Concurrency Classes*

Holding locks as short a time as possible

Locks, as other synchronization mechanisms, allow the definition of a critical section that only one thread can execute at a time. You must be very careful to define the critical section. It only must include those instructions that really need the mutual exclusion. This is especially true if the critical section includes long operations. If the critical section includes lengthy operations that do not use shared resources, the application performance will be worse than it could be.

In this recipe, you will implement an example to see the difference of performance between a task with a long operation inside the critical section and a task with a long operation outside the critical section.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `Operations`.

   ```
   public class Operations {
   ```

2. Implement a `public static` method named `readData()`. It puts the current thread to sleep for 500 milliseconds.

   ```
     public static void readData(){
       try {
         TimeUnit.MILLISECONDS.sleep(500);
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
     }
   ```

3. Implement a `public static` method named `writeData()`. It puts the current thread to sleep for 500 milliseconds.

   ```
     public static void writeData(){
       try {
         TimeUnit.MILLISECONDS.sleep(500);
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
     }
   ```

4. Implement a `public static` method named `processData()`. It puts the current thread to sleep for 2000 milliseconds.

   ```
     public static void processData(){
       try {
         TimeUnit.SECONDS.sleep(2);
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
     }
   ```

5. Implement a class named `Task1` and specify that it implements the `Runnable` interface.

   ```
   public class Task1 implements Runnable {
   ```

6. Declare a private `Lock` attribute named `lock`.

   ```
     private Lock lock;
   ```

7. Implement the constructor of the class to initialize its attributes.

```
public Task1 (Lock lock) {
  this.lock=lock;
}
```

8. Implement the `run()` method. Acquire the lock, call the three operations of the `Operations` class, and release the lock.

```
 @Override
public void run() {
  lock.lock();
  Operations.readData();
  Operations.processData();
  Operations.writeData();
  lock.unlock();
}
```

9. Implement a class named `Task2` and specify that it implements the `Runnable` interface.

```
public class Task2 implements Runnable {
```

10. Declare a private `Lock` attribute named `lock`.

```
private Lock lock;
```

11. Implement the constructor of the class to initialize its attributes.

```
public Task2 (Lock lock) {
  this.lock=lock;
}
```

12. Implement the `run()` method. Acquire the lock, call the `readData()` operation, and release the lock. Then, call the `processData()` method, acquire the lock, call the `writeData()` operation, and release the lock.

```
 @Override
public void run() {
  lock.lock();
  Operations.readData();
  lock.unlock();
  Operations.processData();
  lock.lock();
  Operations.writeData();
  lock.unlock();
}
```

13. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

  public static void main(String[] args) {
```

14. Create a `Lock` object named `lock`, a `Task1` object named `task1`, a `Task2` object named `task2`, and an array of 10 threads.

```
    Lock lock=new ReentrantLock();
    Task1 task1=new Task1(lock);
    Task2 task2=new Task2(lock);
    Thread threads[]=new Thread[10];
```

15. Launch 10 threads to execute the first task controlling its execution time.

```
    Date begin, end;

    begin=new Date();
    for (int i=0; i<threads.length; i++) {
      threads[i]=new Thread(task1);
      threads[i].start();
    }

    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    end=new Date();
    System.out.printf("Main: First Approach: %d\n",(end.getTime()-
begin.getTime()));
```

16. Launch 10 threads to execute the second task controlling its execution time.

```
    begin=new Date();
    for (int i=0; i<threads.length; i++) {
      threads[i]=new Thread(task2);
      threads[i].start();
    }

    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    end=new Date();
    System.out.printf("Main: Second Approach: %d\n",(end.
getTime()-begin.getTime()));
```

How it works...

If you execute the example, you will see a big difference in the execution time of the two approaches. The task that has all the operations inside the critical section takes longer than the other task.

When you need to implement a block of code protected by a lock, analyze it carefully to only include the necessary instructions. Split the method in various critical sections and use more than one lock if it's necessary to get the best performance of your application.

See also

- The *Avoiding deadlocks by ordering the locks* recipe in *Appendix*, *Concurrency Programming Design*

Delegating to executors the management of threads

Before Java 5, the Java Concurrency API, when we wanted to implement a concurrent application, you had to manage the threads for yourself. We implemented the `Runnable` interface or an extension of the `Thread` class. Then, we created a `thread` object and started its execution using its `start()` method. We also had to control its status to know if the thread finished its execution or it's still running.

In Java Version 5, the concept of executor as a provider of a pool of execution threads had appeared. This mechanism, implemented by the `Executor` and `ExecutorService` interfaces and the `ThreadPoolExecutor` and `ScheduledThreadPoolExecutor` classes, allows you to concentrate only in the implementation of the logic of the task. You implement the task and send it to the executor. It has a pool of threads and it is the one that is responsible for the creation, management, and finalization of the threads. In Java Version 7, another implementation of the Executor mechanism in the Fork/Join framework specialized for those problems that can be broken in smaller sub-problems, had appeared. This approach has numerous advantages, which are as follows:

- We don't have to create threads for all the tasks. When we send a task to the executor and it's executed by a thread of the pool, we save the time used in the creation of a new thread. If our application has to execute a lot of tasks, the total saved time will be significant and the performance of the application will be better.
- As we create less threads, our application also uses less memory. This circumstance can also help to get a better performance in our application.
- We can build the concurrent tasks executed in the executor implementing the `Runnable` interface or the `Callable` interface. The `Callable` interface allows us to implement tasks that return a result, which provide a big advantage over the traditional tasks.
- When we send a task to an executor, it returns a `Future` object that allows us to know the status of the task, and the returned result if it has finished its execution easily.
- We can schedule our tasks and execute them repeatedly with the special executor implemented by the `ScheduledThreadPoolExecutor` class.
- We can control easily the resources used by an executor. We can establish the maximum number of threads in the pool, so our executor will never have more than that number of tasks running at a time.

The use of executors has a lot of advantages over the direct utilization of threads. In this recipe, you are going to implement an example that shows how you can obtain better performance using an executor than creating the threads yourself.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` and specify that it implements the `Runnable` interface.

   ```
   public class Task implements Runnable {
   ```

2. Implement the `run()` method. Create a loop with 1,000,000 steps and in each step, do some mathematical operations with an integer variable.

   ```
     @Override
   public void run() {
     int r;
     for (int i=0; i<1000000; i++) {
       r=0;
   ```

```
      r++;
      r++;
      r*=r;
    }
  }
```

3. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

  public static void main(String[] args) {
```

4. Create 1,000 threads to execute 1,000 task objects and wait for its finalization, controlling the total execution time.

```
    Thread threads[]=new Thread[1000];
    Date start,end;

    start=new Date();
    for (int i=0; i<threads.length; i++) {
      Task task=new Task();
      threads[i]=new Thread(task);
      threads[i].start();
    }

    for (int i=0; i<threads.length; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    end=new Date();
    System.out.printf("Main: Threads: %d\n",(end.getTime()-start.
getTime()));
```

5. Create an `Executor` object, send to it 1,000 `Task` objects, and wait for their finalization. Measure the total execution time.

```
    ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
newCachedThreadPool();

    start=new Date();
    for (int i=0; i<threads.length; i++) {
      Task task=new Task();
      executor.execute(task);
```

```
        }
        executor.shutdown();
        try {
          executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        end=new Date();
        System.out.printf("Main: Executor: %d\n",(end.getTime()-start.
getTime()));
```

How it works...

In all the executions of this example, we always obtained a smaller execution time for the executor than creating directly the thread. If your application has to execute a lot of tasks, better employ an executor.

See also

- The *Using executors instead of thread groups* recipe in *Appendix*, *Concurrent Programming Design*
- The *Using the Fork/Join framework instead of executors* recipe in *Appendix*, *Concurrent Programming Design*

Using concurrent data structures instead of programming yourself

Data structures are an essential part of every program. You always have to manage data in your application that you store in a data structure. Arrays, lists, or trees are examples of common data structures. Java API provides a lot of ready-to-use data structures, but when you work with concurrent applications, you have to be careful because not all the structures provided by the Java API are **thread-safe**. If you choose a data structure that is not thread-safe, you can have inconsistent data in your applications.

When you want to use a data structure in your concurrent application, you have to review the documentation of the class that implements that data structure to check that it supports concurrent operations. Java provides the following two kinds of concurrent data structures:

- **Non-blocking data structures**: All the operations provided by these data structures to insert or take off elements in the data structure, if they can't be done now because the data structure is full or empty, return a `null` value to indicate that the operation can't be done now.

- **Blocking data structures**: These data structures provide the same operations provided by the non-blocking data structures, but they also provide operations to insert and to take off data that, if they can't be done immediately, block the thread until the operation can be done.

These are some of the data structures provided by the Java API that you can use in your concurrent applications:

- `ConcurrentLinkedDeque`: It is a non-blocking data structure based on linked nodes that allows you to insert data at the beginning or at the end of the structure.
- `LinkedBlockingDeque`: It is a blocking data structure based on linked nodes. It can have fixed capacity. You can insert elements at the beginning or at the end of the structure. It provides operations that, if the operation can't be done immediately, blocks the thread until the operation can be done.
- `ConcurrentLinkedQueue`: It is a non-blocking queue that allows you to insert elements at the end of the queue and take elements from its beginning.
- `ArrayBlockingQueue`: A blocking queue with a fixed size. You insert the elements at the end of the queue and take elements from its beginning. It provides operations that, if the operation can't be done because the queue is full or empty, puts the thread to sleep until the operation can be done.
- `LinkedBlockingQueue`: A blocking queue that allows you to insert elements at the end of the queue and take elements from its beginning. If it provides operations that, if the operation can't be done because the queue is full or empty, it puts the thread to sleep until the operation can be done.
- `DelayQueue`: It is a `LinkedBlockingQueue` queue with delayed elements. Every element inserted in this queue must implement the `Delayed` interface. An element can't be taken from the list until its delay is 0.
- `LinkedTransferQueue`: It is a blocking queue that provides operations thought to work in situations that can be implemented as a producer/consumer problem. It provides operations that, if the operation can't be done because the queue is full or empty, puts the thread to sleep until the operation can be done.
- `PriorityBlockingQueue`: It is a blocking queue that orders its elements based on its priority. All the elements inserted in the queue must implement the `Comparable` interface. The value returned by the `compareTo()` method will determine the position of the element in the queue. As all the blocking data structures, it provides operations that, if they can't be done immediately, puts the thread to sleep until they can be done.
- `SynchronousQueue`: It is a blocking queue, where every `insert` operation must wait for a `remove` operation for the other thread. The two operations must be done at the same time.
- `ConcurrentHashMap`: It is a **Hash-Map** that allows concurrent operations. It's a non-blocking data structure.

- `ConcurrentSkipListMap`: This data structure associates keys with values. Every key can have only one value. It stores the keys in an ordered way and provides a method to find elements and to get some elements from the map. It's a non-blocking data structure.

There's more...

If you need to use a data structure in your concurrent application, look in the Java API documentation to find the data structure that better fits your needs. Implement your own concurrent data structure that has some problems, which are as follows:

- They have a complex internal structure
- You have to take into account a lot of different situations
- You have to make a lot of tests to guarantee that it works correctly

If you don't find a data structure that fits your needs completely, try to extend one of the existing concurrent data structures to implement one adequately to your problem.

See also

- The recipes in *Chapter 6*, *Concurrent Collections*

Taking precautions using lazy initialization

Lazy initialization is a common programming technique that delays object creation until they are needed for the first time. This normally causes the initialization of the objects to be made in the implementation of the operations instead of the constructor of the classes. The main advantage of this technique is that you can save memory, because you only create the indispensable objects needed for the execution of your applications. You could have declared a lot of objects in one class, but you don't use every object in every execution of your program, so your application doesn't use the memory needed for the objects that you don't use in an execution of the program. This advantage can be very useful for applications that run in environments with limited resources.

By contrast, this technique has the disadvantage of you having a little worse performance in your application, as you create the objects the first time they are used inside an operation.

This technique can also provoke problems if you use it in concurrent applications. As more than one thread can be executing an operation at a time, those threads can be creating an object at the same time and this situation can be problematic. This has a special importance with **singleton** classes. An application has only one object of those classes and, as we mentioned earlier, a concurrent application can create more than one object. Consider the following code:

```
public static DBConnection getConnection(){
  if (connection==null) {
    connection=new DBConnection();
  }
  return connection;
}
```

This is the typical method in a singleton class to obtain the reference of the unique object of that class existing in the application using lazy initialization. If the object hasn't been created yet, it creates the object. Finally, it always returns it.

If two or more threads run at once the comparison of the first sentence (`connection==null`), both threads will create the `Connection` object. That isn't a desirable situation.

In this recipe, you will implement an elegant solution to the lazy initialization problems.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `DBConnectionOK`.

   ```
   public class DBConnectionOK {
   ```

2. Declare a private static `DBConnectionOK` attribute named `connection`.

   ```
     private static DBConnectionOK connection;
   ```

3. Declare a `private` constructor. Write the name of the thread that executes it.

   ```
     private DBConnectionOK() {
       System.out.printf("%s: Connection created.\n",Thread.
   currentThread().getName());
     }
   ```

4. Declare a `private static` class named `LazyDBConnectionOK`. It has a `private static final DBConnectionOK` instance named `INSTANCE`.

   ```
       private static class LazyDBConnection {
           private static final DBConnectionOK INSTANCE = new
   DBConnectionOK();
       }
   ```

5. Implement the `getConnection()` method. It doesn't receive any parameter and returns a `DBConnectionOK` object. It returns the `INSTANCE` object.

```
    public static DBConnectionOK getConnection() {
        return LazyDBConnection.INSTANCE;
    }
```

6. Create a class named `Task` and specify that it implements the `Runnable` interface. Implement the `run()` method. Call the `getConnection()` method of the `DBConnectionOK()` method.

```
public class Task implements Runnable {

  @Override
  public void run() {

    System.out.printf("%s: Getting the connection...\n",Thread.
currentThread().getName());
    DBConnectionOK connection=DBConnectionOK.getConnection();
    System.out.printf("%s: End\n",Thread.currentThread().
getName());
  }

}
```

7. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

  public static void main(String[] args) {
Create 20 Task objects and 20 threads to execute them.
    for (int i=0; i<20; i++){
      Task task=new Task();
      Thread thread=new Thread(task);
      thread.start();
    }

  }
```

How it works...

The key of the example is the `getConnection()` method and the `private static class LazyDBConnection` instance. When the first thread calls the `getConnection()` method, the `LazyDBConnection` class initializes the `INSTANCE` object calling the constructor of the `DBConnection` class. That object is returned by the `getConnection()` method. When the rest of the threads call the `getConnection()` method, the object is already created, so all the threads use the same object that is created only once.

When you run the example, you will see the start and end messages of the 20 tasks, but only one creation message.

Using Fork/Join framework instead of executors

Executors allow you to avoid the creation and management of threads. You implement tasks implementing the `Runnable` or `Callable` interfaces and send them to the executor. It has a pool of threads and uses one of them to execute the tasks.

Java 7 provides a new kind of executor with the Fork/Join Framework. This kind of executor, implemented in the `ForkJoinPool` class, is designed for those problems that can be split into smaller parts using the divide and conquer technique. When you implement a task for the Fork/Join framework, you have to check the size of the problem you have to resolve. If it's bigger than a predefined size, you divide the problem in two or more sub-problems and create as many subtasks as divisions you have made. The task sends those sub-tasks to the `ForkJoinPool` class using the `fork()` operation and waits for its finalization using the `join()` operation.

For these kind of problems, Fork/Join pools get better performance than classical executors. In this recipe, you are going to implement an example where you can check this point.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `TaskFJ` and specify that it extends the `RecursiveAction` class.

   ```
   public class TaskFJ extends RecursiveAction {
   ```

2. Declare and initialize UID's serial version of the class.

```
private static final long serialVersionUID = 1L;
```

3. Declare a private array of `int` numbers named `array`.

```
private int array[];
```

4. Declare two private `int` attributes named `start` and `end`.

```
private int start, end;
```

5. Implement the constructor of the class to initialize its attributes.

```
public TaskFJ(int array[], int start, int end) {
  this.array=array;
  this.start=start;
  this.end=end;
}
```

6. Implement the `compute()` method. If this task has to process a block of more than 1,000 elements (determined by the `start` and `end` attributes), create two `TaskFJ` objects, send them to the `ForkJoinPool` class using the `fork()` method, and wait for its finalization using the `join()` method.

```
 @Override
protected void compute() {
  if (end-start>1000) {
    int mid=(start+end)/2;
    TaskFJ task1=new TaskFJ(array,start,mid);
    TaskFJ task2=new TaskFJ(array,mid,end);
    task1.fork();
    task2.fork();
    task1.join();
    task2.join();
```

7. Otherwise, increment the elements this task has to process. After every increment operation, put the thread to sleep for 1 millisecond.

```
  } else {
    for (int i=start; i<end; i++) {
      array[i]++;
      try {
        TimeUnit.MILLISECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

8. Create a class named `Task` and specify that it implements the `Runnable` interface.

```
public class Task implements Runnable {
```

9. Declare a private array of `int` number named `array`.

```
  private int array[];
```

10. Implement the constructor of the class to initialize its attribute.

```
  public Task(int array[]) {
    this.array=array;
  }
```

11. Implement the `run()` method. Increment all the elements of the array. After every increment operation, put the thread to sleep for 1 millisecond.

```
   @Override
  public void run() {
    for (int i=0; i<array.length; i++ ){
      array[i]++;
      try {
        TimeUnit.MILLISECONDS.sleep(1);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

12. Implement the main class of the example by creating a class named `Main` and add the `main()` method to it.

```
public class Main {

  public static void main(String[] args) {
```

13. Create an `int` array with 100,000 elements.

```
    int array[]=new int[100000];
```

14. Create a `Task` object and a `ThreadPoolExecutor` object to execute them. Execute the task controlling the time for which the task is running.

```
    Task task=new Task(array);
    ThreadPoolExecutor executor=(ThreadPoolExecutor)Executors.
newCachedThreadPool();

    Date start,end;
    start=new Date();
    executor.execute(task);
    executor.shutdown();
```

```
        try {
          executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        end=new Date();
        System.out.printf("Main: Executor: %d\n",(end.getTime()-start.
    getTime()));
```

15. Create a `TaskFJ` object and a `ForkJoinPool` object to execute them. Execute the task controlling the time for which the task is running.

```
        TaskFJ taskFJ=new TaskFJ(array,1,100000);
        ForkJoinPool pool=new ForkJoinPool();
        start=new Date();
        pool.execute(taskFJ);
        pool.shutdown();
        try {
          pool.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        end=new Date();
        System.out.printf("Core: Fork/Join: %d\n",(end.getTime()-
    start.getTime()));

      }
```

How it works...

When you execute the example, you will see how the `ForkJoinPool` and `TaskFJ` classes get a better performance than the `ThreadPoolExecutor` and `Task` classes.

If you have to solve a problem that can be split using the divide and conquer technique, use a `ForkJoinPool` class instead of a `ThreadPoolExecutor` class. You will get a better performance.

See also

- The *Delegating to executors the management of threads* recipe in *Appendix, Concurrent Programming Design*

Avoiding the use of blocking operations inside a lock

Blocking operations are operations that block the execution of the current thread until an event occurs. Typical blocking operations are those involving input or output operations with the console, a file, or the network.

If you use a blocking operation inside the critical section of a lock, you're deteriorating the performance of the application. While a thread is waiting for the event that makes the blocking operation finish, the rest of the application is waiting for that event and none of the other threads can enter in the critical section and execute its code.

In this recipe, you will implement an example of this situation. The threads read a line from the console inside the critical section. This instruction blocks the rest of the threads until this line is introduced.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or other IDE such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. Create a class named `Task` and specify that it implements the `Runnable` interface.

   ```
   public class Task implements Runnable {
   ```

2. Declare a private `Lock` attribute named `lock`.

   ```
     private Lock lock;
   ```

3. Implement the constructor of the class to initialize its attribute.

   ```
     public Task (Lock lock) {
       this.lock=lock;
     }
   ```

4. Implement the `run()` method.

   ```
      @Override
     public void run() {
       System.out.printf("%s: Starting\n",Thread.currentThread().
   getName());
   ```

5. Acquire the lock using the `lock()` method.

   ```
       lock.lock();
   ```

6. Call the `criticalSection()` method.

```
    criticalSection();
```

7. Read a line from the console.

```
    System.out.printf("%s: Press a key to continue: \n",Thread.
currentThread().getName());
    InputStreamReader converter = new InputStreamReader(System.
in);
    BufferedReader in = new BufferedReader(converter);
    try {
      String line=in.readLine();
    } catch (IOException e) {
      e.printStackTrace();
    }
```

8. Free the lock using the `unlock()` method.

```
    lock.unlock();
  }
```

9. Implement the `criticalSection()` method. Wait for a random period of time.

```
  private void criticalSection() {
    Random random=new Random();
    int wait=random.nextInt(10);
    System.out.printf("%s: Wait for %d seconds\n",Thread.
currentThread().getName(),wait);
    try {
      TimeUnit.SECONDS.sleep(wait);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
```

10. Implement the main class of the application by creating a class named `Main` and add the `main()` method to it.

```
public class Main {
  public static void main(String[] args) {
```

11. Create a new `ReentrantLock` object named `lock`. Create 10 `Task` objects and 10 threads to execute them.

```
    ReentrantLock lock=new ReentrantLock();
    for (int i=0; i<10; i++) {
      Task task=new Task(lock);
      Thread thread=new Thread(task);
      thread.start();
    }
```

How it works...

When you execute this example, 10 threads start their execution, but only one enters in the critical section, which gets implemented in the `run()` method. As every task reads a line from the console before releasing the lock, all the applications will be blocked until you introduce a text in the console.

See also

- The *Holding locks as short a time as possible* recipe in *Appendix*, *Concurrent Programming Design*

Avoiding the use of deprecated methods

Java concurrency API also has deprecated operations. These are operations that were included in the first versions of the API, but now you shouldn't use them. They have been replaced by other operations that implement better practices than the original ones.

The most critical deprecated operations are those provided by the `Thread` class. These operations are:

- `destroy()`: In the past, this method destroyed the thread. Actually, it throws a `NoSuchMethodError` exception.
- `suspend()`: This method suspends the execution of the thread until it's resumed.
- `stop()`: This method forces the thread to finish its execution.
- `resume()`: This method resumes the execution of the thread.

The `ThreadGroup` class also has some deprecated methods, which are as follows:

- `suspend()`: This method suspends the execution of all the threads that belong to this thread group until they have resumed
- `stop()`: This method forces the finish of the execution of all the threads of this thread group
- `resume()`: This method resumes the execution of all the threads of this thread group

The `stop()` operation has been deprecated because it can provoke inconsistent errors. As it forces the thread to finish its execution, you can have a thread that finishes its execution before the completion of an operation and can leave the data in an inconsistent status. For example, if you have a thread that is modifying a bank account and it's stopped before they finish, the bank account will probably have erroneous data.

The `stop()` operation also can cause a deadlock situation. If this operation is called when the thread is executing a critical section protected by a synchronization mechanism (for example, a lock), this synchronization mechanism will continue blocking and no thread can enter the critical section. This is the reason because the `suspend()` and `resume()` operations have been deprecated.

If you need an alternative to these operations, you can use an internal attribute to store the status of the thread. This attribute must be protected with synchronized access or use an atomic variable. You must check the value of this attribute and take the actions according to it. Take into account that you have to avoid data inconsistency and deadlock situations to guarantee the correct operation of your application.

Using executors instead of thread groups

The `ThreadGroup` class provides a mechanism to group threads in a hierarchical structure, so you can do operations with all the threads that belong to a thread group with only one call. By default, all the threads belong to the same group, but you can specify a different one when you create the thread.

Anyway, thread groups don't provide any feature that make their use interesting:

- You have to create the threads and manage their status
- The methods that control the status of all the threads of the thread group have been deprecated and their use is very discouraging

If you need to group threads under a common structure, it is better to use an `Executor` implementation such as `ThreadPoolExecutor`. It provides more functionality, which are as follows:

- You don't have to worry about the management of the threads. The executor creates them and reuses them to save execution resources.
- You can implement your concurrent tasks implementing the `Runnable` interface or the `Callable` interface. The `Callable` interface allows you to implement tasks that return a result, which provide a big advantage over the traditional tasks.
- When you send a task to an executor, it returns a `Future` object that allows you to know the status of the task, and the returned result if it has finished its execution easily.
- You can schedule your tasks and execute them repeatedly with the special executor implemented by the `ScheduledThreadPoolExecutor` class.
- You can control easily the resources used by an executor. You can establish the maximum number of threads in the pool, so your executor will never have more than that number of tasks running at a time.

For these reasons, it is better that you don't use thread groups and use executors instead.

See Also

- The *Delegating to executors the management of threads* recipe *Appendix*, *Concurrent Programming Design*