VIII

CONCURRENCY PATTERNS

Concurrency patterns deal with:

- Ways to lock class code and an order of locking objects to prevent the occurrence of race conditions and deadlocks
- The details of streamlining access to an application resource to improve the overall application responsiveness
- The details of method execution while a required precondition is not met

Chapter	*Pattern Name*	*Description*
41	Critical Section	Stricter form of Monitor. Used to lock the code at the class level to keep multiple threads from executing the locked code even on two different instances of the same class.
42	Consistent Lock Order	Recommends identifying and documenting a well-defined order of locking objects to be followed consistently during the design and the development of an application to eliminate the possibility of the occurrence of a deadlock.
43	Guarded Suspension	Recommends a method to be designed to suspend its execution until the object is in a state that makes a required precondition true.
44	Read-Write Lock	Recommends allowing simultaneous read operations while preventing simultaneous updates to the values of an application resource in order to improve the overall application responsiveness.

41

CRITICAL SECTION

DESCRIPTION

A Critical Section is a segment of code that must be executed by only one thread at a time to produce the expected results. When more than one thread is allowed to execute this code segment, it could produce unpredictable results. By this definition, a critical section looks very similar to the concept of a Monitor discussed in Section III — Basic Patterns. The following is the list of similarities and differences between Monitors and Critical Sections:

- A Critical Section is a stricter form of a Monitor.
- A Monitor locks a single object whereas a Critical Section requires a lock on an entire class of objects.
- In Java:
  - The implementation of a Monitor on a method requires the method to be declared using the synchronized keyword.
  - A Critical Section can be implemented by using the combination of both the static and the synchronized keywords.
- In the case of a Monitor, no two threads are allowed to execute the synchronized code on the same object. Two threads can execute the same synchronized code on two different objects. In contrast, in the case of a critical section, no two threads are allowed to execute the code on two different objects. This is because the code is locked at the class level, not at the object level.

EXAMPLE

During the discussion of the Singleton pattern, we designed a message logging class `FileLogger` as a singleton. The `FileLogger` class maintains a class variable logger of the `FileLogger` type. This variable is used to hold the singleton `FileLogger` instance. The `FileLogger` class offers a class-level method `getFileLogger` that can be used by different client objects to access the singleton `FileLogger` instance. As part of the `getFileLogger` method implementation, the `FileLogger` checks to see if the singleton instance has already been created. Checking to see if the class variable logger is null does this. If logger is found to be uninitialized, a `FileLogger` instance is created by invoking its private constructor and is assigned to the logger class variable. This implementation of the `getFileLogger` method works fine in a single-threaded environment. In a multithreaded environment, it is possible for two threads to simultaneously execute the `getFileLogger` method to see if the class variable logger is null and, as a result, initialize logger twice. This means that the `FileLogger` private constructor gets invoked twice.

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt”,msg, true, true);
  }
}
```

Initializing the logger variable twice in this example does not result in an error. This is because the `FileLogger` private constructor does not do any complex, critical initialization. In contrast, if the singleton constructor method executes such operations as opening a socket connection on a particular port, executing the constructor twice could result in an error.

Let us enhance the design of the `FileLogger` class to make it suitable for use in multithreaded environments. This can be accomplished in two ways.

Approach I (Critical Section)

This involves making the `getFileLogger` method a Critical Section so that only one thread can ever execute it at any given point in time. This can be accomplished by simply declaring the class-level method `getFileLogger` as synchronized.

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  private FileLogger() {
  }
```

```
  public static synchronized FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt”,msg, true, true);
  }
}
```

This simple change turns the getFileLogger method into a Critical Section and guarantees that no two threads ever execute the getFileLogger method at the same time. This completely eliminates the possibility of the FileLogger constructor getting invoked more than once inside the getFileLogger method.

Approach II (Static Early Initialization)

It is to be noted that synchronizing methods can have a significant effect on the overall application performance. In general, synchronized methods run much slower, as much as 100 times slower than their nonsynchronized counterparts. As an alternative to declaring the getFileLogger method as synchronized, the logger variable can be early initialized.

```
public class FileLogger implements Logger {
  //Early Initialization
  private static FileLogger logger = new FileLogger();
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt”,msg, true, true);
  }
}
```

This eliminates the need for any check or initialization inside the getFileLogger method. As a result, the getFileLogger becomes thread-safe automatically without having to declare it as synchronized.

PRACTICE QUESTIONS

1. Design a database connection class as a thread-safe singleton.
2. Design a printer spooler class as a thread-safe singleton.

42

CONSISTENT LOCK ORDER

DESCRIPTION

During the discussion of the Monitor and the Critical Section patterns earlier, we have seen that when the synchronized keyword is used to ensure single-threaded execution of a code block, a thread needs to wait while trying to acquire the lock associated with the specified object. Consider a scenario where two threads hold locks on two different objects and each one is waiting for a lock on the object that is locked by the other thread. Both threads will be waiting forever and are said to be in a state of deadlock. In terms of implementation, this type of situation most often occurs due to an inconsistent order of locking objects. Let us consider the code segment in Listing 42.1 to illustrate how inconsistent locking in a multithreaded environment can cause a deadlock.

Consider a scenario where:

- Two threads, A and B, simultaneously invoke methods — `Method_A` and `Method_B` — respectively on the same `SomeClass` object.
- Thread A acquires a lock on `objectA` and Thread B acquires a lock on `objectB` at the same time. At this point, each of the threads waits for a lock on the object locked by the other thread and this puts Thread A and Thread B in a deadlocked condition.

To address such deadlock issues, the Consistent Lock Order pattern recommends designing an object locking order to be followed consistently across an application. Simply following an object locking order consistently across the application (where objects of a particular class are to be locked before locking other class instances) can eliminate the deadlock problem associated with the example code block. In other words, by ensuring that objects are locked in a consistent order all across the application, the problem of deadlocks can be addressed.

The example code block in Listing 42.1 can be modified so that `ClassA` objects are locked prior to locking `ClassB` objects.

Listing 42.1 Class with Inconsistent Locking Order

```
public class SomeClass {
  private ClassA objectA;
  private ClassB objectB;
  public SomeClass() {
    objectA = new ClassA();
    objectB = new ClassB();
  }
  public void Method_A() {
    synchronized (objectA) {
      synchronized (objectB) {
        process_A();
      }
    }
  }
  public void Method_B() {
    synchronized (objectB) {
      synchronized (objectA) {
        process_B();
      }
    }
  }
  private void process_A() {
    //
  }
  private void process_B() {
    //
  }
}
class ClassA {
}
class ClassB {
}
```

```
public void Method_A() {
  synchronized (objectA) {
    synchronized (objectB) {
      process_A();
    }
  }
}
public void Method_B() {
  synchronized (objectA) {
    synchronized (objectB) {
      process_B();
    }
  }
}
```

This type of object locking order based on the class type does not work when the objects to be locked are instances of the same class. A more sophisticated algorithm may be needed to decide the object locking order. The following example illustrates one such mechanism.

EXAMPLE

Let us build a utility class that offers the functionality to move the contents of a directory to a different directory in the file system.

Let us create a class `Directory`, instances of which can be used to represent directories in the file system.

```
public class Directory {
  private String name;
  public Directory(String n) {
    name = n;
  }
}
```

The utility class `FileSysUtil` in its simplest form can be designed with a method to move the contents between directories.

```
public class FileSysUtil {
  public void moveContents(Directory src, Directory dest) {
    synchronized (src) {
      synchronized (dest) {
        System.out.println("Contents Moved Successfully");
      }
    }
  }
}
```

To move the contents of a directory to another, a client object or thread needs to:

1. Create Directory objects corresponding to the source and destination directories.
2. Invoke the moveContents method by passing both the Directory objects created in Step 1.

As part of its implementation of the moveContents method, the FileSysUtil locks the Directory objects representing the source and destination directories in sequence before actually moving the directory contents. This is to prevent threads from changing or deleting the source or destination directories while the current thread is in the process of moving the source directory contents to the destination directory. For simplicity, the example application displays an appropriate message instead of actually moving the source directory contents.

Let us suppose that there exist two directories — dir1 and dir2 — in the file system. To move the contents of dir1 to dir2, a thread (e.g., Thread_A) needs to create two Directory objects — objDir_1 and objDir_2 — corresponding to dir1 and dir2, respectively and pass them as arguments to the moveContents method.

```
//For Thread_A objDir_1 is the source directory
  moveContents(objDir_1, objDir_2);
```

While executing the moveContents method, Thread_A attempts to acquire locks on objDir_1 and objDir_2 in sequence.

At the same time, a different thread (e.g., Thread_B) invokes the moveContents method on the same FileSysUtil object to move dir2 contents to dir1. Using the same Directory objects used by Thread_A, Thread_B makes a call as follows:

```
//For Thread_B objDir_2 is the source directory
  moveContents(objDir_2, objDir_1);
```

Similar to Thread_A, while executing the moveContents method, Thread_B also attempts to acquire locks on objDir_1 and objDir_2 but in the reverse order.

If Thread_A and Thread_B acquire locks at the same time on objDir_1 and objDir_2, respectively, then each thread continues to wait for a lock on the Directory object locked by the other thread and this causes a deadlock. Because both objDir_1 and objDir_2 are of the same Directory class type, defining an object locking order based on the class type does not work in this case. As an alternative, the built-in Java hashCode method can be used to define an order of locking Directory objects. The hashCode method is defined in the topmost java.lang.Object class and is inherited by all classes in Java.

The hashCode method returns the unique ID or hash code associated with an object. An object locking scheme can be defined based on some kind of order of the hash codes of the objects to be locked.

To eliminate the possibility of a deadlock situation, the moveContents method can be modified so that the objects representing the source and the destination directories are locked in the ascending order of their associated hash codes. This ensures that the Directory objects are always locked in the same order, even if they are passed to the moveContents method by two different threads in different order.

```
      …
      …
  public void moveContents(Directory src, Directory dest) {
    if (src.hashCode() > dest.hashCode()) {
      synchronized (src) {
          synchronized (dest) {
            System.out.println("Contents Moved Successfully");
          }
        }
      } else {
          synchronized (dest) {
            synchronized (src) {
              System.out.println("Contents Moved Successfully");
            }
          }
        }
      }
      …
      …
```

With this change in place, when two threads invoke the moveContents method at the same time to move the contents of two different directories in opposite directions, only one thread is granted lock on the first Directory object to be locked. The second thread simply waits for the lock on the first Directory object itself. The possibility of the second thread locking the second Directory object while the first thread locks the first Directory object does not arise.

The example application uses a simple mechanism to define the locking order for Directory objects. In the case of a real world application, a locking order that is suitable for the application needs to be identified and documented. This locking order can then be followed consistently during the design and the development of the application.

PRACTICE QUESTIONS

1. Design a class `AccountManager` with a method to transfer money from one bank account to another. For this class to be used in a multithreaded environment, it must lock both the account objects before performing the actual transfer. Implement a method to transfer money so that when two different threads attempt to transfer money between two different accounts at the same time in opposite directions, it does not result in a deadlock in a multithreaded environment.
2. Design a class `InventoryManager` with a method to move products from one distribution center to another. For this class to be used in a multithreaded environment, it must lock the objects representing the two distribution centers that are participating in the transaction before performing actual updates to their inventory levels. The method to move products should be implemented in a manner that does not cause a deadlock when two different threads attempt to move items between two distribution centers at the same time in opposite directions.

43

GUARDED SUSPENSION

This pattern was previously described in Grand98 and is based on the material that appeared in Lea97.

DESCRIPTION

In general, each method in an object is designed to execute a specific task. Sometimes, when a method is invoked on an object, the object may need to be in a certain state, which is logically necessary for the method to carry out the action it is designed for. In such cases, the Guarded Suspension pattern suggests suspending the method execution until such a precondition becomes true. In other words, the requirement for the object to be in a particular state becomes a precondition for the method to execute its implementation of the intended task.

Every class in Java inherits the `wait`, `notify` and `notifyAll` methods from the base `java.lang.Object` class. When a thread invokes an object's `wait` method:

- It makes the thread release the synchronization lock it holds on the object.
- The thread remains in the waiting state until it is notified to return via the `notify` or `notifyAll` method.

Using these built-in `wait`, `notify` and `notifyAll` methods, the Guarded Suspension pattern can be implemented in Java.

The generic structure of a Java class when the Guarded Suspension pattern is applied using the built-in `wait`, `notify` and `notifyAll` methods is represented in Listing 43.1.

The class `SomeClass` consists of two synchronized methods — `guardedMethod` and `alterObjectStateMethod`. The `guardedMethod` represents a method that requires some kind of a precondition to become true before proceeding with its execution. Hence, it checks if the precondition is true and as long as the precondition is not true, it waits using the `wait` method.

The `alterObjectStateMethod` method enables different client objects (threads) to change the state of a `SomeClass` instance. This, in turn, could result in the required precondition becoming true. Once the state of the object is

Listing 43.1 Generic Class Structure

```
public class SomeClass {
  synchronized void guardedMethod() {
    while (!preCondition()) {
      try {
        //Continue to wait
        wait();
        //…
      } catch (InterruptedException e) {
        //…
      }
    }
    //Actual task implementation
  }
  synchronized void alterObjectStateMethod() {
    //Change the object state
    //…..
    //Inform waiting threads
    notify();
  }
  private boolean preCondition() {
    //…
    return false;
  }
}
```

changed, this method notifies any waiting thread that is waiting inside the `guardedMethod` using the `notify` method. If the change in the object state makes the precondition true, the waiting thread resumes with the execution of the `guardedMethod`. Otherwise, it continues to wait till the precondition becomes true.

Both the `guardedMethod` and `alterObjectStateMethod` methods are designed as synchronized methods to prevent race conditions in a multithreaded environment.

EXAMPLE

Let us build an application to simulate the parking mechanism at a health club. A member can park his car if there is an empty parking slot. If there is no empty parking slot, a member needs to wait until one of the parking slots becomes available.

Listing 43.2 `ParkingLot` Class

```
class ParkingLot {
  //Assume 4 parking slots for simplicity
  public static final int MAX_CAPACITY = 4;
  private int totalParkedCars = 0;
  public synchronized void park(String member) {
    while (totalParkedCars >= MAX_CAPACITY) {
      try {
        System.out.println(" The parking lot is full " +
                           member + " has to wait ");
        wait();
      } catch (InterruptedException e) {
        //
      }
    }
    //precondition is true
    System.out.println(member + " has parked");
    totalParkedCars = totalParkedCars + 1;
  }
  public synchronized void leave(String member) {
    totalParkedCars = totalParkedCars - 1;
    System.out.println(member +
                       " has left, notify a waiting member");
    notify();
  }
}
```

A simple representation for the parking lot can be designed in the form of the `ParkingLot` class shown in Listing 43.2.

The `ParkingLot` maintains the total number of currently parked cars in its instance variable `totalParkedCars`. This constitutes the state of a `ParkingLot` object.

The existence of an empty slot is the precondition for a member to proceed with parking his car. It can be seen that the `park` method first checks to see if this precondition is satisfied. If the number of currently parked members is greater than or equal to the total number of available slots, it can be inferred that there is no empty parking slot available and the member needs to wait until this condition does not exist. When a member leaves the parking lot, the total number of currently parked members is decremented and the `leave` method notifies one of the waiting threads at random. Once the notification is received, the notified thread attempts to get a lock on the object. Once the lock is obtained, it checks to see if the precondition is satisfied by reentering the while loop. If the precondition is satisfied, it proceeds with the parking action. The example code simply displays a message and increments the total number of currently parked cars. Checking for the precondition by the notified thread may seem redundant but it is required in a multithreaded environment. This is because of the possibility of a different thread altering the object state between the time the waiting thread attempts to obtain a lock on the object and the time it obtains it, so that the precondition becomes false.

Use of `wait()` and `notify()` in the `ParkingLot` Class Design

- The `park` method uses the built-in `java.lang.Object wait()` method to keep a Member thread waiting while the precondition is not true. When the `wait()` method is called, the currently executed thread (in this case a Member) is placed in the wait queue and its lock on the `ParkingLot` object is released (it had a lock on the `ParkingLot` object because park is synchronized). The next Member thread is then free to enter the `park` method and checks if `totalParkedCars >= MAX_CAPACITY`, which if true, is also placed into the wait queue.
- The `leave` method uses the built-in `java.lang.Object notify` method to notify a single waiting thread at random. The choice of the thread is at the discretion of the specific JVM implementation. The notified thread regains a lock on the `ParkingLot` object and returns to executing in the `park` method where the `wait()` method was invoked. Using the built-in `notifyAll` method the `leave` method could also be implemented to notify all waiting threads at once. The waiting threads then contend for the `ParkingLot` object lock. Whatever thread obtains the lock continues execution in the `park` method where the `wait()` method was called.

The representation of a member can be designed as a Java Thread (Listing 43.3) to facilitate the simulation of more than one member looking to park their cars at the same time.

Let us design a test driver `GSTest` to make use of the `Member` class to simulate a real world scenario of multiple members trying to park their cars at the same time.

```
public class GSTest {
  public static void main(String[] args) {
    ParkingLot parking = new ParkingLot();
    new Member("Member1", parking);
    new Member("Member2", parking);
    new Member("Member3", parking);
    new Member("Member4", parking);
    new Member("Member5", parking);
    new Member("Member6", parking);
  }
}
```

Listing 43.3 `Member` Class

```
class Member extends Thread {
  private ParkingLot parking;
  private String name;
  Member(String n, ParkingLot p) {
    name = n;
    parking = p;
    start();
  }
  public void run() {
    System.out.println(name + " is ready to park");
    parking.park(name);
    try {
      sleep(500);
    } catch (InterruptedException e) {
      //
    }
    //leave after 500ms
    parking.leave(name);
  }
}
```

PRACTICE QUESTIONS

1. Design a queue data structure to be used by multiple threads in an application. A thread can retrieve an object from the queue only if the queue contains any elements. Apply the Guarded Suspension pattern in designing the queue class so that when a thread attempts to retrieve an object from the queue and the queue is empty, the thread is made to wait until an object is put into the queue by a different thread.
2. Apply the Guarded Suspension pattern to design the item check-out functionality at a library. Typically, a library maintains multiple copies of an item such as a movie or a book. Member A can check out an item only if the total number of its copies is greater than the number of members prior to Member A with interest in the same item.

44

READ-WRITE LOCK

This pattern was previously described in Grand98 and is based on the material that appeared in Lea97.

DESCRIPTION

During the discussion of the Monitor and the Critical Section patterns earlier, we saw that when multiple threads in an application simultaneously access a resource it could result in unpredictable behavior. Hence the resource must be protected so that only one thread at a time is allowed to access the resource. Though this may be required in most cases, it may lead to unwanted CPU overhead when some of the threads accessing the resource are interested only in reading the values or state of the resource but not in changing it. In such cases, it can be inefficient to prevent a thread from accessing the resource solely to read its values while a different thread is currently reading the same resource values. Because a read operation does not alter the values of the resource, multiple threads can safely be allowed to access the resource at the same time if all of these threads are interested only in reading the resource values. This kind of design improves the overall application responsiveness with reduced CPU overhead. That means, when a thread obtains a lock to simply read the values of a resource, it should not prevent other threads from accessing the resource to read its values. In other words, a read lock should be shared. If a thread is allowed to read a resource's data while a different thread is updating the same resource, the thread that is reading the data may receive an inconsistent view. Allowing more than one thread to update the values of a resource could also result in unpredictable results.

While some threads are interested only in reading the resource values, some other threads may access the resource to read and update its values. To eliminate concurrency problems, when such a thread needs to access the resource to update its values, it must get a write lock on the object representing the resource. A write lock is an exclusive lock on the object and prevents all other threads from accessing the resource at the same time. Further, if a read and a write lock are requested on an object at the same time, the write lock request should be granted first. The write lock is issued only if there are no threads currently holding a read lock on the same object.

Table 44.1 summarizes the criteria for issuing a read-write lock.

Table 44.1 Rules for Issuing Read-Write Locks

Lock	*Rules*
Read Lock	A read lock should be issued if there is no currently issued write lock and there are no threads waiting for the write lock.
Write Lock	A write lock should be issued if no thread is currently issued a (read or write) lock on the object.

In Java, there is no readily available feature for implementing read-write locks. But a custom class can be built (Listing 44.1) with the responsibility of issuing read-write locks on an object to different threads in an application.

Design Highlights of the `ReadWriteLock` Class

Lock Statistics

The `ReadWriteLock` maintains different lock statistics in a set of instance variables as follows:

- `totalReadLocksGiven` — To store the number of read locks already issued on the object.
- `writeLockIssued` — To indicate if a write lock has been issued or not.
- `threadsWaitingForWriteLocks` — To keep track of the number of threads currently waiting for a write lock.

These values are in turn used by the lock issuing methods — `getReadLock` and `getWriteLock`.

Lock Methods

The `ReadWriteLock` offers two methods — `getReadLock` and `getWrite-Lock` — which can be used by client objects to get read and write locks on an object, respectively. As part of its implementation of these two methods, the `ReadWriteLock` issues read-write locks as per the rules listed in Table 44.1.

Lock Release

A client object that currently holds a read-write lock can release the lock by invoking the `done` method. The `done` method updates appropriate lock statistics and allows the lock to be issued to any waiting thread as per the rules listed in Table 44.1.

The `ReadWriteLock` class is a generic implementation for issuing read-write locks and can be readily used in any application.

Listing 44.1 Generic `ReadWriteLock` Implementation

```
public class ReadWriteLock {
  private Object lockObj;
  private int totalReadLocksGiven;
  private boolean writeLockIssued;
  private int threadsWaitingForWriteLock;
  public ReadWriteLock() {
    lockObj = new Object();
    writeLockIssued = false;
  }
  /*
    A read lock can be issued if
      there is no currently issued
      write lock and
      there is no thread(s) currently waiting for the
      write lock
    */
  public void getReadLock() {
    synchronized (lockObj) {
      while ((writeLockIssued) ||
        (threadsWaitingForWriteLock != 0)) {
      try {
        lockObj.wait();
      } catch (InterruptedException e) {
        //
      }
    }
    //System.out.println(" Read Lock Issued");
    totalReadLocksGiven++;
  }
}
/*
  A write lock can be issued if
    there is no currently issued
    read or write lock
  */
```

(continued)

Listing 44.1 Generic `ReadWriteLock` Implementation (Continued)

```
    public void getWriteLock() {
      synchronized (lockObj) {
        threadsWaitingForWriteLock++;
        while ((totalReadLocksGiven != 0) ||
            (writeLockIssued)) {
          try {
            lockObj.wait();
          } catch (InterruptedException e) {
            //
          }
        }
        //System.out.println(" Write Lock Issued");
        threadsWaitingForWriteLock -- ;
        writeLockIssued = true;
      }
    }
    //used for releasing locks
    public void done() {
      synchronized (lockObj) {
        //check for errors
        if ((totalReadLocksGiven == 0) &&
            (!writeLockIssued)) {
          System.out.println(
            " Error: Invalid call to release the lock");
          return;
        }
        if (writeLockIssued)
          writeLockIssued = false;
        else
          totalReadLocksGiven -- ;
          lockObj.notifyAll();
        }
      }
    }
```

EXAMPLE

Applying the Read-Write Lock pattern, let us design an application to allow members of a library to:

- View details of different library items
- Check out an item if it is currently available

The application must ensure that multiple members are allowed to view an item status at the same time, but only one member is allowed to check out an item at a time. In other words, the application must support multiple simultaneous member transactions without producing unpredictable results.

The overall application design becomes much simpler using the `ReadWriteLock` class designed earlier. The representation of a library item can be designed in the form of an `Item` class (Listing 44.2) with methods to allow members to check the status of an item and to check in or check out an item.

Because the status check of an item does not involve changes to its status, the `getStatus` method acquires a read lock. This allows more than one thread to invoke the `getStatus` method to check the status of an item.

In contrast, both the `checkIn` and `checkOut` methods involve changes to the item status and hence acquire a write lock before changing the item status. This ensures that only one thread is allowed to alter the item status even though more than one thread invokes the `checkIn/checkOut` method at the same time. The `Item` class makes use of the services of a `ReadWriteLock` object to acquire an appropriate lock.

By using the exclusive write lock only when needed, the `Item` class allows multiple threads to access an item in a more controlled manner without the overhead of any unwanted waiting and eliminates the scope for unpredictable behavior at the same time.

The representation of a member transaction can be designed as a Java Thread (Listing 44.3) to facilitate the reflection of the real world scenario of different members accessing an item simultaneously.

The `MemberTransaction` class is designed in its simplest form and can be configured with an operation to check an item status or to check in or check out an item when it is instantiated.

To simulate a real world scenario, a test program `RWTest` can be designed to create multiple `MemberTransaction` objects to perform different operations to read the status of an item or check in or check out an item.

```
public class RWTest {
  public static void main(String[] args) {
    Item item = new Item("CompScience-I");
    new MemberTransaction("Member1", item, "StatusCheck");
    new MemberTransaction("Member2", item, "StatusCheck");
    new MemberTransaction("Member3", item, "CheckOut");
    new MemberTransaction("Member4", item, "CheckOut");
```

```
    new MemberTransaction("Member5", item, "CheckOut");
    new MemberTransaction("Member6", item, "StatusCheck");
  }
}
```

When the RWTest is executed, the order in which different read-write locks are issued will be displayed.

Listing 44.2 `Item` Class

```
public class Item {
  private String name;
  private ReadWriteLock rwLock;
  private String status;
  public Item(String n) {
    name = n;
    rwLock = new ReadWriteLock();
    status = "N";
  }
  public void checkOut(String member) {
    rwLock.getWriteLock();
    status = "Y";
    System.out.println(member +
                       " has been issued a write lock-ChkOut");
    rwLock.done();
  }
  public String getStatus(String member) {
    rwLock.getReadLock();
    System.out.println(member +
                       " has been issued a read lock");
    rwLock.done();
    return status;
  }
  public void checkIn(String member) {
    rwLock.getWriteLock();
    status = "N";
    System.out.println(member +
                       " has been issued a write lock-ChkIn");
    rwLock.done();
  }
}
```

Listing 44.3 `MemberTransaction` Class

```
public class MemberTransaction extends Thread {
  private String name;
  private Item item;
  private String operation;
  public MemberTransaction(String n, Item i, String p) {
    name = n;
    item = i;
    operation = p;
    start();
  }
  public void run() {
    //all members first read the status
    item.getStatus(name);
    if (operation.equals("CheckOut")) {
      System.out.println("\n" + name +
                         " is ready to checkout the item.");
      item.checkOut(name);
      try {
        sleep(1);
      } catch (InterruptedException e) {
        //
      }
      item.checkIn(name);
    }
  }
}
```

PRACTICE QUESTIONS

1. Design an application to allow different customers to buy airline tickets. Apply the Read-Write Lock pattern to ensure that multiple customers are allowed to check the seat availability on the same flight, but only one customer is allowed to buy the ticket at a time.
2. Design an application to allow different customers to bid on auctioned items. Apply the Read-Write Lock pattern to ensure that multiple customers are allowed to check the current bid but no two customers are allowed to alter the bid amount at the same time.