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.