CHAPTER 3

Waiting and Notification

Java provides a small API that supports communication between threads. Using this API, one thread waits for a *condition* (a prerequisite for continued execution) to exist. In the future, another thread will create the condition and then notify the waiting thread. In this chapter, I introduce you to this API.

Wait-and-Notify API Tour

The `java.lang.Object` class provides a Wait-and-Notify API that consists of three `wait()` methods, one `notify()` method, and one `notifyAll()` method. The `wait()` methods wait for a condition to exist; the `notify()` and `notifyAll()` methods notify waiting threads when the condition exists:

- `void wait()`: Cause the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for this object, or for some other thread to interrupt the current thread while waiting.
- `void wait(long timeout)`: Cause the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for this object, or for the specified amount of time measured in milliseconds (identified by `timeout`) to pass, or for some other thread to interrupt the current thread while waiting. This method throws `java.lang.IllegalArgumentException` when `timeout` is negative.
- `void wait(long timeout, int nanos)`: Cause the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for this object, or for the specified amount of time measured in milliseconds (identified by `timeout`) plus nanoseconds (identified by `nanos`) to pass, or for some other thread to interrupt the current thread while waiting. This method throws `IllegalArgumentException` when `timeout` is negative, `nanos` is negative, or `nanos` is greater than `999999`.

- `void notify()`: Wake up a single thread that's waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.
- `void notifyAll()`: Wake up all threads that are waiting on this object's monitor. The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

The three `wait()` methods throw `java.lang.InterruptedException` when any thread interrupted the current thread before or while the current thread was waiting for a notification. The interrupted status of the current thread is cleared when this exception is thrown.

■ **Note** A thread releases ownership of the monitor associated with the object whose `wait()` method is called.

This API leverages an object's *condition queue*, which is a data structure that stores threads waiting for a condition to exist. The waiting threads are known as the *wait set*. Because the condition queue is tightly bound to an object's lock, all five methods must be called from within a synchronized context (the current thread must be the owner of the object's monitor); otherwise, `java.lang.IllegalMonitorStateException` is thrown.

The following code/pseudocode fragment demonstrates the noargument `wait()` method:

```
synchronized(obj)
{
   while (<condition does not hold>)
      obj.wait();

   // Perform an action that's appropriate to condition.
}
```

The `wait()` method is called from within a synchronized block that synchronizes on the same object as the object on which `wait()` is called (`obj`). Because of the possibility of *spurious wakeups* (a thread wakes up without being notified, interrupted, or timing out), `wait()` is called from within a `while` loop that tests for the condition holding and reexecutes `wait()` when the condition still doesn't hold. After the `while` loop exits, the condition exists and an action appropriate to the condition can be performed.

■ **Caution** Never call a `wait()` method outside of a loop. The loop tests the condition before and after the `wait()` call. Testing the condition before calling `wait()` ensures *liveness*. If this test was not present, and if the condition held and `notify()` had been called prior to `wait()` being called, it's unlikely that the waiting thread would ever wake up. Retesting the condition after calling `wait()` ensures *safety*. If retesting didn't occur, and if the condition didn't hold after the thread had awakened from the `wait()` call (perhaps another thread called `notify()` accidentally when the condition didn't hold), the thread would proceed to destroy the lock's protected invariants.

The following code fragment demonstrates the `notify()` method, which notifies the waiting thread in the previous example:

```
synchronized(obj)
{
   // Set the condition.

   obj.notify();
}
```

Notice that `notify()` is called from a critical section guarded by the same object (`obj`) as the critical section for the `wait()` method. Also, `notify()` is called using the same `obj` reference. Follow this pattern and you shouldn't get into trouble.

■ **Note** There has been much discussion about which notification method is better: `notify()` or `notifyAll()`. For example, check out "Difference between `notify()` and `notifyAll()`" (`http://stackoverflow.com/questions/14924610/difference-between-notify-and-notifyall`). If you're wondering which method to use, I would use `notify()` in an application where there are only two threads, and where either thread occasionally waits and needs to be notified by the other thread. Otherwise, I would use `notifyAll()`.

Producers and Consumers

A classic example of thread communication involving conditions is the relationship between a producer thread and a consumer thread. The producer thread produces data items to be consumed by the consumer thread. Each produced data item is stored in a shared variable.

Imagine that the threads are running at different speeds. The producer might produce a new data item and record it in the shared variable before the consumer retrieves the previous data item for processing. Also, the consumer might retrieve the contents of the shared variable before a new data item is produced.

To overcome those problems, the producer thread must wait until it's notified that the previously produced data item has been consumed, and the consumer thread must wait until it's notified that a new data item has been produced. Listing 3-1 shows you how to accomplish this task via `wait()` and `notify()`.

Listing 3-1. The Producer-Consumer Relationship Version 1

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}

class Shared
{
   private char c;
   private volatile boolean writeable = true;

   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
            wait();
         }
         catch (InterruptedException ie)
         {
         }
      this.c = c;
      writeable = false;
      notify();
   }
   synchronized char getSharedChar()
   {
      while (writeable)
         try
         {
            wait();
         }
         catch (InterruptedException ie)
         {
         }
      writeable = true;
      notify();
      return c;
   }
}

class Producer extends Thread
{
   private final Shared s;

   Producer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         s.setSharedChar(ch);
         System.out.println(ch + " produced by producer.");
      }
   }
}
class Consumer extends Thread
{
   private final Shared s;

   Consumer(Shared s)
   {
      this.s = s;
   }
   @Override
   public void run()
   {
      char ch;
      do
      {
         ch = s.getSharedChar();
         System.out.println(ch + " consumed by consumer.");
      }
      while (ch != 'Z');
   }
}
```

This application creates a `Shared` object and two threads that get a copy of the object's reference. The producer calls the object's `setSharedChar()` method to save each of 26 uppercase letters; the consumer calls the object's `getSharedChar()` method to acquire each letter.

The `writeable` instance field tracks two conditions: the producer waiting on the consumer to consume a data item and the consumer waiting on the producer to produce a new data item. It helps coordinate execution of the producer and consumer. The following scenario, where the consumer executes first, illustrates this coordination:

1. The consumer executes `s.getSharedChar()` to retrieve a letter.
2. Inside of that synchronized method, the consumer calls `wait()` because `writeable` contains `true`. The consumer now waits until it receives notification from the producer.
3. The producer eventually executes `s.setSharedChar(ch);`.
4. When the producer enters that synchronized method (which is possible because the consumer released the lock inside of the `wait()` method prior to waiting), the producer discovers `writeable`'s value to be `true` and doesn't call `wait()`.
5. The producer saves the character, sets `writeable` to `false` (which will cause the producer to wait on the next `setSharedChar()` call when the consumer has not consumed the character by that time), and calls `notify()` to awaken the consumer (assuming the consumer is waiting).
6. The producer exits `setSharedChar(char c)`.
7. The consumer wakes up (and reacquires the lock), sets `writeable` to `true` (which will cause the consumer to wait on the next `getSharedChar()` call when the producer has not produced a character by that time), notifies the producer to awaken that thread (assuming the producer is waiting), and returns the shared character.

Compile Listing 3-1 as follows:

```
javac PC.java
```

Run the resulting application as follows:

```
java PC
```

You should observe output such as the following excerpt during one run:

```
W produced by producer.
W consumed by consumer.
X produced by producer.
X consumed by consumer.
Y produced by producer.
Y consumed by consumer.
Z produced by producer.
Z consumed by consumer.
```

Although the synchronization works correctly, you might observe multiple producing messages before multiple consuming messages:

```
A produced by producer.
B produced by producer.
A consumed by consumer.
B consumed by consumer.
```

Also, you might observe a consuming message before a producing message:

```
V consumed by consumer.
V produced by producer.
```

Either strange output order doesn't mean that the producer and consumer threads aren't synchronized. Instead, it's the result of the call to `setSharedChar()` followed by its companion `System.out.println()` method call not being synchronized, and by the call to `getSharedChar()` followed by its companion `System.out.println()` method call not being synchronized. The output order can be corrected by wrapping each of these method call pairs in a synchronized block that synchronizes on the `Shared` object referenced by `s`. Listing 3-2 presents this enhancement.

Listing 3-2. The Producer-Consumer Relationship Version 2

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}

class Shared
{
   private char c;
   private volatile boolean writeable = true;

   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
            wait();
         }
         catch (InterruptedException ie)
         {
         }
      this.c = c;
      writeable = false;
      notify();
   }

   synchronized char getSharedChar()
   {
      while (writeable)
         try
         {
            wait();
         }
         catch (InterruptedException ie)
         {
         }
      writeable = true;
      notify();
      return c;
   }
}
```

```
class Producer extends Thread
{
   private final Shared s;

   Producer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         synchronized(s)
         {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
         }
      }
   }
}
class Consumer extends Thread
{
   private final Shared s;

   Consumer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      char ch;
      do
      {
         synchronized(s)
         {
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
         }
      }
      while (ch != 'Z');
   }
}
```

Compile Listing 3-2 (`javac PC.java`) and run this application (`java PC`). Its output should always appear in the same alternating order as shown next (only the first few lines are shown for brevity):

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 3's content:

1. Define condition.
2. Describe the API that supports conditions.
3. True or false: The `wait()` methods are interruptible.
4. What method would you call to wake up all threads that are waiting on an object's monitor?
5. True or false: A thread that has acquired a lock doesn't release this lock when it calls one of `Object`'s `wait()` methods.
6. Define condition queue.
7. What happens when you call any of the API's methods outside of a synchronized context?
8. Define spurious wakeup.
9. Why should you call a `wait()` method in a loop context?
10. Create an `Await` application that demonstrates a higher-level concurrency construct known as a *gate*. This construct permits multiple threads to arrive at a synchronization point (the *gate*) and wait until the gate is unlocked by another thread so that they can all proceed.

The `main()` method first creates a runnable for the threads that will wait at the gate. The runnable prints a message stating that the thread is waiting, increments a counter, sleeps for 2 seconds, and waits (make sure to account for spurious wakeups). Upon wakeup, the thread outputs a message stating that the thread is terminating. `main()` then creates three `Thread` objects and starts three threads to execute the runnable. Next, `main()` creates another runnable that repeatedly sleeps for 200 milliseconds until the counter equals 3, at which point it notifies all waiting threads. Finally, `main()` creates a `Thread` object for the second runnable and starts the thread.

Summary

Java provides an API that supports communication between threads. This API consists of `Object`'s three `wait()` methods, one `notify()` method, and one `notifyAll()` method. The `wait()` methods wait for a condition to exist; `notify()` and `notifyAll()` notify waiting threads when the condition exists.

The `wait()`, `notify()`, and `notifyAll()` methods are called from within a synchronized block that synchronizes on the same object as the object on which they are called. Because of spurious wakeups, `wait()` is called from a `while` loop that reexecutes `wait()` while the condition doesn't hold.

A classic example of thread communication involving conditions is the relationship between a producer thread and a consumer thread. The producer thread produces data items to be consumed by the consumer thread. Each produced data item is stored in a shared variable.

To overcome problems such as consuming a data item that hasn't been produced, the producer thread must wait until it's notified that the previously produced data item has been consumed, and the consumer thread must wait until it's notified that a new data item has been produced.

Chapter 4 presents additional thread capabilities.