



**Trail:** Essential Java Classes

**Lesson:** Threads: Doing Two or More Tasks At Once

# Synchronizing Threads

So far, this lesson has contained examples with independent, asynchronous threads. That is, each thread contained all of the data and methods required for its execution and didn't require any outside resources or methods. In addition, the threads in those examples ran at their own pace without concern over the state or activities of any other concurrently running threads.

However, there are many interesting situations where separate, concurrently running threads do share data and must consider the state and activities of other threads. One such set of programming situations are known as producer/consumer scenarios where the producer generates a stream of data which then is consumed by a consumer.

For example, imagine a Java application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: the first shares a file, the second shares an event queue. Because the threads share a common resource, they must be synchronized in some way.

## The Producer/Consumer Example

This lesson teaches you about Java thread synchronization through a simple producer/consumer example.

## Locking an Object

The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*. In the Java language, a critical section can be a block or a method and are identified with the `synchronized` keyword. The Java platform then associates a lock with every object that has synchronized code.

## Using the `notifyAll` and `wait` Methods

This section investigates the code in `CubbyHole`'s `put` and `get` methods that helps

the Producer and Consumer coordinate their activities.

## Avoid Starvation and Deadlock

If you write a program in which several concurrent threads are competing for resources, you must take precautions to ensure fairness. A system is fair when each thread gets enough access to limited resource to make reasonable progress. A fair system prevents *starvation* and *deadlock*. Starvation occurs when one or more threads in your program is blocked from gaining access to a resource and thus cannot make progress. Deadlock is the ultimate form of starvation; it occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.

This section uses the dining philosophers problem to illustrate deadlock. It also discusses ways you can prevent deadlock.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)  
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



**Trail:** Essential Java Classes

**Lesson:** Threads: Doing Two or More Tasks At Once

## The Producer/Consumer Example

The **Producer** generates an integer between 0 and 9 (inclusive), stores it in a **CubbyHole** object, and prints the generated number. To make the synchronization problem more interesting, the Producer sleeps for a random amount of time between 0 and 100 milliseconds before repeating the number generating cycle:

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

The **Consumer**, being ravenous, consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
    }
}
```

```

        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}

```

The `Producer` and `Consumer` in this example share data through a common `CubbyHole` object. And you will note that neither the `Producer` nor the `Consumer` makes any effort whatsoever to ensure that the `Consumer` is getting each value produced once and only once. The synchronization between these two threads actually occurs at a lower level, within the `get` and `put` methods of the `CubbyHole` object. However, let's assume for a moment that these two threads make no arrangements for synchronization and talk about the potential problems that might arise in that situation.

One problem arises when the `Producer` is quicker than the `Consumer` and generates two numbers before the `Consumer` has a chance to consume the first one. Thus the `Consumer` would skip a number. Part of the output might look like this:

```

. . .

Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5

```

. . .

Another problem that might arise is when the `Consumer` is quicker than the `Producer` and consumes the same value twice. In this situation, the `Consumer` would print the same value twice and might produce output that looked like this:

```

. . .

Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5

```

. . .

Either way, the result is wrong. You want the `Consumer` to get each integer produced by the `Producer` exactly once. Problems such as those just described are called *race conditions*. They arise from multiple, asynchronously executing threads trying to access a single object at the same time and getting the wrong result.

Race conditions in the producer/consumer example are prevented by having the storage of a new integer into the `CubbyHole` by the `Producer` be synchronized with the retrieval of an integer from the `CubbyHole` by the `Consumer`. The `Consumer` must consume each integer exactly once.

The activities of the `Producer` and `Consumer` must be synchronized in two ways. First, the two threads must not simultaneously access the `CubbyHole`. A Java thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked. [Locking an Object](#) discusses this.

And second, the two threads must do some simple coordination. That is, the `Producer` must have some way to indicate to the `Consumer` that the value is ready and the `Consumer` must have some way to indicate that the value has been retrieved. The `Thread` class provides a collection of methods--`wait`, `notify`, and `notifyAll`--to help threads wait for a condition and notify other threads of when that condition changes. [Using the notifyAll and wait Methods](#) has more information.

## The Main Program

Here's a small stand-alone [Java application](#) that creates a `CubbyHole` object, a `Producer`, a `Consumer`, and then starts both the `Producer` and the `Consumer`.

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

## The Output

Here's the output of `ProducerConsumerTest`.

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)  
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

**Trail:** Essential Java Classes**Lesson:** Threads: Doing Two or More Tasks At Once

## Locking an Object

The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*. In the Java language, a critical section can be a block or a method and are identified with the `synchronized` keyword. The Java platform then associates a lock with every object that has synchronized code.

In the producer/consumer example, the `put` and `get` methods of the `CubbyHole` are the critical sections. The `Consumer` should not access the `CubbyHole` when the `Producer` is changing it, and the `Producer` should not modify it when the `Consumer` is getting the value. So `put` and `get` in the `CubbyHole` class should be marked with the `synchronized` keyword.

Here's a code skeleton for the `CubbyHole` class:

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        ...
    }

    public synchronized void put(int value) {
        ...
    }
}
```

Note that the method declarations for both `put` and `get` contain the `synchronized` keyword. Hence, the system associates a unique lock with every instance of `CubbyHole` (including the one shared by the `Producer` and the `Consumer`). Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called. Other threads cannot call a synchronized method on the same object until the object is unlocked.

So, when the `Producer` calls `CubbyHole`'s `put` method, it locks the `CubbyHole`, thereby preventing the `Consumer` from calling the `CubbyHole`'s `get` method:

```
public synchronized void put(int value) {  
    // CubbyHole locked by the Producer  
    ..  
    // CubbyHole unlocked by the Producer  
}
```

When the `put` method returns, the Producer unlocks the `CubbyHole`.

Similarly, when the Consumer calls `CubbyHole`'s `get` method, it locks the `CubbyHole`, thereby preventing the Producer from calling `put`:

```
public synchronized int get() {  
    // CubbyHole locked by the Consumer  
    ...  
    // CubbyHole unlocked by the Consumer  
}
```

The acquisition and release of a lock is done automatically and atomically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity. Synchronization isn't the whole story. The two threads must also be able to notify one another when they've done their job. Learn more about that after a brief foray into reentrant locks.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)  
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.