Methods wait and notifyAll

We provide an example that shows the need for two more thread methods: wait() and notifyAll(). We implement a “dropbox”, a place where threads, called producers, can deposit an integer for other threads, called consumers, to pick up. A dropbox is like a vending machine that can hold only one item, one candy bar, and if there is no candy bar in it, you have to wait until the vendor puts one there.

A producer wanting to put a new integer into a full dropbox must wait until a consumer takes the current integer out. A consumer wanting to take out an integer must wait until there is one there.

To implement waiting, Java maintains two lists for a synchronized object sob.

1. The locklist: The list of threads that are waiting to obtain the lock on sob; they are ready to execute a synchronized statement or method.
2. The waitlist: The list of threads that had the lock but couldn’t proceed because sob was not in a suitable state (e.g. a consumer found the dropbox empty). So they executed method wait(). They are waiting until the state of sob changes.

Suppose a thread t uses a synchronized statement or method to acquire the lock on sob. Two things can happen:

1. Thread t can’t complete its task (e.g. a consumer finds the dropbox empty). Therefore, t executes a call wait(), which puts t on the waitlist and makes it relinquish the lock on sob. Thus, another thread can obtain the lock. Of course, when t is given another chance, t’s call on wait() completes and t continues executing from that point.
2. Thread t can complete its task. It does so and executes a call notifyAll(). This call moves all threads from the waitlist to the locklist. Why? Thread t changed the state of sob, so threads on the waitlist may now be able to complete their task. Moving them to the locklist gives them a chance to try again.

**Class Dropbox**

Class Dropbox has two fields: full tells whether the Dropbox contains an integer; if it does, the integer is in field box.

The two calls on wait() are in try-statements; this is necessary because an InterruptedException may be thrown while the method is waiting.

A consumer will call method take(). Suppose the call gets the lock and the Dropbox is full. The while-loop terminates. Field full is set to indicate that the Dropbox is empty, notifyAll() is called to move all threads on the waitlist to the locklist, and the value in box is returned.

Suppose the call take() gets the lock but the Dropbox is empty. Then wait() is called, so the thread is put on the waitlist and gives up the lock. When it is given another chance, execution wait(), the try-statement, and the repentend all terminate and the while-condition !full is evaluated again.

Why is a while-loop necessary? Suppose an if-statement is used instead; the method body is:

```java
if (!full)
    try {wait();} catch (...) {} 
full= false; notifyAll(); return box;
```

Consider this execution sequence: (1) take() is called and the Dropbox is empty. The call wait() is executed. (2) A producer puts an integer into the dropbox and does notifyAll(). (3) A different consumer is given the lock and takes the value out, thus emptying the buffer. (4) This consumer is given the lock; its call wait() completes, and it continues as if the Dropbox were full, but it is not. The while loop is needed to make sure that a consumer processes normally only if the Dropbox is full.

Another reason for the while-loop: If an InterruptedException is thrown and caught, full might be false.
A producer calls method put(int) to put an integer into the Dropbox, but if the Dropbox is full, the producer must wait until it is empty. The structure of method put(int) is similar to that of method take(), and we leave you to study it.

**A complete application**

To the right, we show a class Main that creates two Consumer threads and one Producer thread and starts all three executing.

Class Consumer alternately takes a value from a Dropbox and sleeps. Class Producer is similar.

The producers and consumers are given the Dropbox with which to work as a parameter of their constructors. This is a typical way to write such classes. It is more flexible than the method used in pdf file 6 under the JavaHyperText entry for Threads, which used instead a public static variable declared in the main class.

We have placed a println statement in the while-loop of the Consumer’s method run(). Put these classes into an Eclipse or DrJava project and execute this application and see what happens. Add other println statements to class Dropbox, if you want, in order to see how the Producer and two Consumers work together.

**What about method notify()?**

Method notifyAll() moves all threads from the waitlist to the locklist. There is a method notify(), which instead moves only one thread, chosen arbitrarily, from the waitlist to the locklist. Using notify() instead of notifyAll() may work, but in some cases it causes deadlock — no process can make progress. Don’t use notify() unless you know what you are doing.

The following document in JavaHyperText entry Threads explains why notify() may not work:

Warning: use notifyAll and not notify unless you know what you are doing.

```java
import java.util.Random;

public class Main {
    /** Create two Consumer threads and a * Producer thread and start all three. */
    public static void main(String[] args) {
        Dropbox box= new Dropbox();
        new Thread(new Consumer(box)).start();
        new Thread(new Consumer(box)).start();
        new Thread(new Producer(box)).start();
    }
}

/** An instance alternately takes an integer * from a Dropbox and sleeps */
public class Consumer implements Runnable {
    private Dropbox box;
    /** Constructor: a Consumer using db */
    public Consumer(Dropbox db) {box= db; }
    /** Forever: Get a value from the Dropbox * and sleep for a random time. */
    public void run() {
        Random random= new Random();
        while (true) {
            int i= box.take();
            System.out.println(
                Thread.currentThread().getName() + " " + i);
            try {
                Thread.sleep(random.nextInt(100));
            } catch (InterruptedException e) { }
        }
    }
}

/** An instance repeatedly sleeps and * puts a random number into a Dropbox. */
public class Producer implements Runnable {
    private Dropbox box;
    /** Constructor: a Producer using db */
    public Producer(Dropbox db) {box= db; }
    /** Forever: sleep for a random time and * put a random number into the Dropbox. */
    public void run() {
        Random random= new Random();
        while (true) {
            int n= random.nextInt(10);
            try {
                Thread.sleep(random.nextInt(100));
            } catch (InterruptedException e) { }
            box.put(n);
        }
    }
}
```

©David Gries, 2018