# The Secrets of Concurrency (Part 1)

www.javaspecialists.eu/archive/Issue146.html

by Dr. Heinz M. Kabutz
**Abstract:**
Learn how to write correct concurrent code by understanding the Secrets of Concurrency. This is the first part of a series of laws that help explain how we should be writing concurrent code in Java.

Welcome to the 146th issue of

The Java(tm) Specialists' Newsletter. I am on my way to TheServerSide Java Symposium in Barcelona, where on Friday I will be presenting a new talk entitled "The Secrets of Concurrency". After many days of feverishly trying to come up with an outline for my talk, we ended up praying for inspiration. The writer's block disappeared immediately!

As mentioned in the previous newsletter, from July 2007, I will be offering Java code reviews for your team to get an expert's viewpoint of your Java system.

**NEW:** Please see our new "Extreme Java" course, combining concurrency, a little bit of performance and Java 8. Extreme Java - Concurrency & Performance for Java 8.

## Secrets of Concurrency (Part 1)

Have you ever used the **synchronized** keyword? Are you absolutely sure that your code is correct? Here are ten laws that will help you obey the rules and write correct code, which I will explain over the next few newsletters:

## Introduction

After watching myself and those around me for 35 years, I have come to the conclusion that we are exceedingly dim-witted. We walk through life only half-awake. I have gotten in my car to drive somewhere, then arrived at a completely different place to where I wanted to go to. Lost in thought, then lost on the road.

There are some people amongst us who are wide awake. They will hear a concept once and will immediately understand it and forever remember it. These are the exceptions. If like me, you struggle to understand concepts quickly or to remember things for a long time, *you are not alone*.

This is what makes conferences challenging. I know that I risk exposing my own lack of intelligence by saying this, but most of the talks I have attended have ended up going completely over my head. The first quarter of the talk would normally be alright, but suddenly I would be lost, with no way of recovery. I have great admiration for those that sit through an entire talk without opening up their laptops.

Just to illustrate the point, here is a picture taken of me during the "Enterprise Java Tech Day" in Athens. The caption on the Java Champion read "Heinz Kabutz busy preparing for his talk ..." Had the picture been taken from behind, the caption would have had to be "Heinz Kabutz

busy playing backgammon on his laptop ..." In fairness, the speaker was talking in Greek, which I could not follow anyway.

To make my talk as simple as possible to understand, I have summarised The Secrets of Concurrency in ten easily remembered laws. Here is a challenge. Try to forget the name "The Law of the Sabotaged Doorbell". Let me know if you manage.

## Law 1: The Law of the Sabotaged Doorbell

*Instead of suppressing interruptions, manage them properly.*

Many years ago we used to live in a house without a fence. We would constantly be interrupted by people coming to ring our door bell. Neighbours who had dropped a ball behind our house, kids selling brownies, men looking for a gardening job (after seeing how I looked after my lawn). When my son was born, we had to contend with a newborn baby and a constant stream of people waking up said baby with the chime. After some frustrating pacing, holding our new bundle of joy, I got fed up and sabotaged the doorbell by removing the battery.

Now all was calm, except that I was also hiding those chimes that were actually quite important. These InterruptedExceptions were being caused by other threads and sometimes I was supposed to listen to it.

How often have you seen code like this? I have even seen Sun Java Evangelists doing this in their talks!

```
try {
 Thread.sleep(1000); // 1 second
} catch(InterruptedException ex)
{
 // ignore - won't happen
}
```

There are two questions we will try to answer:

1. What does InterruptedException mean?
2. How *should* we handle it?

The first question has a simple and a difficult answer. The simple answer is that the thread was interrupted by another thread.

A more difficult answer is to examine what happens and then see how we can use this in our coding. The thread is interrupted when another thread calls its `interrupt()` method. The first thing that happens is that the interrupted status of the thread is set to `true`. This interrupted status is important when we look at methods that put a thread into a WAITING or a TIMED_WAITING state. Examples of these methods are: wait(), Thread.sleep(), BlockingQueue.get(), Semaphore.acquire() and Thread.join().

If the thread is currently in either the WAITING or TIMED_WAITING states, it immediately causes an InterruptedException and returns from the method that caused the waiting state. If the thread is not in a waiting state, then only the interrupted status is set, nothing else. However, if later on the thread calls a method that would change the state into WAITING or TIMED_WAITING, the InterruptedException is immediately caused and the method returns.

Note that attempting to lock on a monitor with **`synchronized`** puts the thread in BLOCKED state, not in WAITING nor TIMED_WAITING. Interrupting a thread that is blocked will do nothing except set the interrupted status to true. You cannot stop a thread from being blocked

by interrupting it. Calling the `stop()` method similarly has no effect when a thread is blocked. We will deal with this in a later law.

The interrupted status is nowadays commonly used to indicate when a thread should be shut down. The problem with the `Thread.stop()` method was that it would cause an asynchronous exception at any point of your thread's execution code. The InterruptedException is thrown at well defined places. Compare it to firing employees when they are idle, rather than when they are in the middle of important work.

Note that the interrupted status is set to false when an InterruptedException is caused or when the `Thread.interrupted()` method is explicitely called. Thus, when we catch an InterruptedException, we need to remember that the thread is now not interrupted anymore! In order to have orderly shutdown of the thread, we should keep the thread set to "interrupted".

What should we do when we call code that may cause an InterruptedException? Don't immediately yank out the batteries! Typically there are two answers to that question:

1. Rethrow the InterruptedException from your method. This is usually the easiest and best approach. It is used by the new java.util.concurrent.* package, which explains why we are now constantly coming into contact with this exception.
2. Catch it, set interrupted status, return. If you are running in a loop that calls code which may cause the exception, you should set the status back to being interrupted. For example:

```
while (!Thread.currentThread().isInterrupted())
{
 // do something
 try {
 TimeUnit.SECONDS.sleep(1000);
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 break;
 }
}
```

Remember the Law of the Sabotaged Doorbell - don't just ignore interruptions, manage them properly!

In the next newsletter, I will explain the Law of the Distracted Spearfisherman and the Law of the Overstocked Haberdashery.

Kind regards from Athens Airport

Heinz

Concurrency Articles Related Java Course