13.Implementing the ProducerConsumer Pattern Using Wait Notify

Contents

[1. Introduction: Agenda of the Module 1](#_Toc9559)

[2. Launching a Task in a New Thread with the Runnable Pattern 2](#_Toc1399)

[3. How to Stop a Thread Using the interrupt() Method 2](#_Toc27037)

[4. Implementing a First Producer/Consumer Example 2](#_Toc27660)

[5. A First Synchronized Version of the Producer/Consumer 2](#_Toc25286)

[6. What Is Needed to Fix This First, Flawed Version 2](#_Toc9330)

[7. Understanding How the wait() and notify() Methods Work 2](#_Toc19984)

[8. Fixing the Producer / Consumer Code Using Wait / Notify 3](#_Toc9410)

[9. Understanding How Synchronization Works with wait() and notify() 3](#_Toc1975)

[10. Live Coding: Unsynchronized, Flawed Producer/Consumer in Action 3](#_Toc17580)

[11. Live Coding: Synchronized, Flawed Producer/Consumer in Action 3](#_Toc19334)

[12. Live Coding: wait/notify, Correct Producer/Consumer in Action 3](#_Toc28729)

[13. Understanding the States of a Thread 3](#_Toc16138)

[14. Understanding the State Diagram of a Thread 4](#_Toc8651)

[15. Wrap-up on the State of a Thread 4](#_Toc14873)

[16. Wrap-up of the Module 4](#_Toc25843)

# 1. Introduction: Agenda of the Module

=>slides: Pg. 1

Welcome to the second module of this course, module called Implementing the Producer/Consumer Pattern Using Wait and Notify.

=>slides: Pg. 2

What are we going to see in this module? Well, we are going to see in details the Runnable pattern, pattern in Java used to launch and stop threads. We are going to see what is the producer/consumer pattern. This pattern is very widely used. There are several solutions to implement it in Java, and we are going to see the most simple one, the one based on the Wait/Notify pattern. So we will see how to implement this producer/consumer pattern using a synchronization and the wait/notify pattern.

=>slides: Pg. 3

So, let us see now the Runnable pattern.

=>slides: Pg. 4

This pattern is about launching threads, and it is the first pattern introduced in Java to do this. It has been introduced in Java 1. 0, that is in the very early days of the language and it is still used nowadays, of course. There are other patterns introduced in Java 5, in the java. util. concurrent API. We are not going to see them. They are beyond the scope of this course.

# Launching a Task in a New Thread with the Runnable Pattern

=>slides: Pg. 5

The first question we are going to answer is how to launch a new thread.

=>slides: Pg. 6

A thread is something that executes a task. In Java 1, the model for a task is the Runnable interface.

=>slides: Pg. 7

It is a very simple interface called Runnable. There is only one method in it, run. That does not take any argument and that does not return anything.

=>slides: Pg. 8

Since this interface has only one method, in Java 8, it became a functional interface with this functional interface annotation added to it. It does not change anything in Java 7. It does not introduce any kind of backward incompatibility. It is just leveraging a new feature of the Java 8 compiler.

=>slides: Pg. 9

This Runnable interface can be implemented using a number expression as it is the case here.So first of all, the Runnable pattern consists in creating an instance of Runnable like this.

=>slides: Pg. 10

Then, creating a thread object by passing this Runnable task as a parameter to the constructor of the thread class.

=>slides: Pg. 11

And then, launching this thread by calling the start method of the thread object.

=>slides: Pg. 12

Now, there is also a run method in this thread class. Be careful when you launch a new thread. It is the start method you want to call and not the run method. The run method will indeed call the run method of the task object you have passed as a parameter but it does not execute this call in another thread, it executes it in the current thread

=>slides: Pg. 13

so this is not what you want to do. What you want to do is really create a new thread and this is what is done by calling the start method.

=>slides: Pg. 14

There is a trick to knowing which thread a task is executed. There is a static method in the thread class called current thread that will return a reference to the thread that is running this task. So when I'm writing thread. currentthread. getname, what I'm doing here is printing out the name of the thread running this task.

# How to Stop a Thread Using the interrupt() Method

=>slides: Pg. 15

The second question is how to stop a thread. It might look simple to answer but in fact, it is a little tricky and there are traps behind that.

=>slides: Pg. 16

Stopping a thread is more tricky than it seems. There is indeed a method in the thread class called stop. If I am mentioning this method here, it is to tell you that you should not use this method at all. If you check the documentation on this method, you will have the exact reasons that this method has in fact been introduced in the first version of the thread class before the people who wrote it realized that it was a wrong idea to create such a method. The problem is once this method was published, it was not possible to remove it on the thread class without introducing a backward incompatibility. This method is here just for legacy and backward compatibility reasons. It will not be removed in future versions but the only thing you want to know with this method is that you should stay away from it. The right pattern to stop a thread is in fact to use the interrupt method. Now the trick is the interrupt method will not stop a thread, but merely send a signal to the task the thread is running telling it that it is time for this task to stop itself.

=>slides: Pg. 17

So how does it work? I have a thread, whatever it is, I call the interrupt method on this running thread. The code of the task this thread is executing should call the isInterrupted method inside itself to decide to terminate itself.

=>slides: Pg. 18

So in my Runnable task as I wrote previously, I need some kind of while loop and while the thread that is running the is not interrupted, then I can carry on with whatever I have to do but if the thread has been interrupted, then I should stop myself, cleaning all the resources I have opened, for instance.

=>slides: Pg. 19

So how can I stop a thread? Well, I need to call interrupt on the thread I want to stop and this calling will cause the isInterrupted method to return true So this isInterrupted method should be scanned, should be regularly checked by the Runnable that is executed by that thread. If the thread is blocked or waiting, then the corresponding method will throw an interrupted exception. We have not seen in which case a thread can be blocked but there are several examples of that, for instance, the wait and notify method that we are going to see now, offering this exception and the join method that we saw in the live connect session of the previous module, also throws an InterruptedException.

# Implementing a First Producer/Consumer Example

=>slides: Pg. 20

Let us talk now about the Producer/Consumer pattern.

=>slides: Pg. 21

First of all, what is a producer/consumer? Well it is a very simple and very easy to understand pattern. =>slides: Pg. 22

We have a producer that is producing values stored in a buffer, think about an array for instance. And we have a consumer that is consuming the values from this buffer. Most of the time, I will have more than one producer, more than one consumer, and they all be executed in their own thread. Of course, we need to take care of the fact that the buffer can be empty or full. So if it is empty, a consumer cannot consume values. If it is full, a producer should not try to add values in it. As I said, producers and consumers are run in their own thread. It means that these buffer, which is shared among all the threads, may be the object of a race condition if I do not properly synchronize my code.

=>slides: Pg. 23

This is a very simple producer. I have a count variable that will just count the number of elements present in the buffer. My buffer is just an array of int and my producer has a method produce that will check if the buffer is full. While the buffer is full, of course, it should not try to add any object in it and when there is some room in the buffer, then it will just add a value in this buffer, the one int,

=>slides: Pg. 24

and then the consumer has a method consume. While the buffer is empty, the consumer should not try to consume any value from the buffer, just because there are none, and when there are values available, then it will just take a value from this buffer here, setting the one to zero.

# A First Synchronized Version of the Producer/Consumer

=>slides: Pg. 25

Okay so that was the very first and very naïve way of writing a producer and a consumer. The first question is: What is wrong with this code? Well of course, there is one main thing that is completely wrong, is that I have quite an obvious race condition here.

=>slides: Pg. 26

If my producer and my consumer are run in their own threads, it mean that I have several threads reading and writing the buffer and the count variable at the same time. This is the exact definition of the race condition we gave in the previous module of this course. In Java, the effect will be very nasty. It will corrupt the array, I will not be able to write or read values from this array because of concurrent excess on this array.

=>slides: Pg. 27

So how can I fix my producer/consumer? One way to fix things here is just to synchronize the access to the array. And this is what we learned in the previous module so let us just do that.

=>slides: Pg. 28

So here is our first naïve attempt at fixing this problem, just synchronize each consume and produce method by adsing the synchronized keyword on it. Does it really fix our problems? Well in fact it does not because synchronization will indeed be the solution of our problem but not if we write it like that. Why? Because as we said in the previous module, if we write synchronized like that, it means that the object holding the key that the thread will need to run the consume and the produce method is the consumer instance itself and what we want is to avoid a thread from running the consume method when another thread is running the produce method so we need a common synchronization object command to all the instances of consumer and producer.

=>slides: Pg. 29

So we need a code that will look like this one, a private object lock that will be made common to all the consumers and producers instances and that will be used in a synchronized block inside the consume method for the consumer and the produce method for the producer. This version of the code will work if the lock object is the same for all the instances of producers and consumers.

=>slides: Pg. 30

So the code, in the end, will look like this one, with a synchronized block inside the consume and the produce method.

# What Is Needed to Fix This First, Flawed Version

=>slides: Pg. 31

Now, is our code really fixed? If we write it like that, will it really work?

=>slides: Pg. 32

Let us take a closer look at what we have written. And the question is: What happens if the buffer is empty?

=>slides: Pg. 33

Well, the thread executing this consumer will be running the isEmpty method inside this infinite while loop forever so this thread will be blocked inside the consume method, inside the synchronized block, while holding the key of the lock object.

=>slides: Pg. 34

So what is going to happen in the producer? The thread running the producer will be waiting for the key held by this lock object and since this key is not available because it is held by the consumer thread, it will not be able to execute its synchronized block. So it will have no chance to add any object to the buffer. So in fact, this way of writing things, this way of naively synchronizing our two methods is is not the right way to do it, it will lead to a deadlock, not the exact same kind of deadlock we saw in the first module but still a deadlock.

=>slides: Pg. 35

So we need another fix for this one. What we need to do is a way to kind of park the consumer thread while it is waiting for some data to be produced. And when this thread is parked, it should not be blocking all the other threads and of course, namely, the producer threads not are going to add data in our buffer. So the consequence of that is that the key held by the consumer thread should be released and made available to the producer threads while this consumer thread is parked. And this is exactly what does this wait and notify pattern.

# Understanding How the wait() and notify() Methods Work

=>slides: Pg. 36

So how does it work, this Wait/Notify pattern? What does it do? This is what we are going to see now. =>slides: Pg. 37

From a pure technical point of view, wait and notify are two methods from the object class was available and all the Java objects we create. These methods are invoked on a given object. Now, there is a rule the thread executing the invocation should be holding the key of that object. If the thread that is executing a wait method does not hold the key of the object on which it is executing this method, then an exception is raised. Now, we must keep in mind that the only way for a thread to hold the key of an object is to be in a synchronized block, synchronized on this object. So the consequence of that is that wait and notify cannot be invoked outside a synchronized block. And if you have an invocation of these methods outside of a synchronized block, this is a bug, it will crash when you try to run that.

=>slides: Pg. 38

Let us see first what is happening when a thread calls the wait method. First of all, calling the wait on a lock object releases the key held by the thread. So this key becomes available to the other thread and this is exactly what we want, as we saw in our example. The second thing it does is that it puts the current thread in a particular state called the WAIT state. This WAIT state is not the same as the state in which a thread is when it is waiting at the entrance of a synchronized block. It is a special thread state. The only way to release a thread from a WAIT state is to call notify on the lock object this thread is using. So what happens when we call notify?

=>slides: Pg. 39

Calling notify released a thread that is in a WAIT state so a thread that has called a wait method and it puts it in the Runnable state. And this is the only way to release a waiting thread. So if you never call notify in your code, at some point, the new application will most probably not work properly. If there are more than one thread in the WAIT state, and this is the case most of the time, the released thread by the notify method is chosen randomly among those threads. I also have a notifyALL method that will awake all the threads in the WAIT state.

# Fixing the Producer / Consumer Code Using Wait / Notify

=>slides: Pg. 40

So let us have a look at the second version of our code. I still have my produce method with a synchronized block inside. If my buffer is full, then instead of checking if it is full once again, I am just going to put this thread in the WAIT state by calling lock. wait. So at this point, the key held by the thread running this method will be released and made available for the consumers. If my buffer wasn't full, then I add objects to my buffer and since I put objects in my buffer, I am going to notify all the consumers that may be in the WAIT state.

=>slides: Pg. 41

On the consumer part, how does it work? Well, I also have a synchronized block on this locked object. If my buffer is empty, then I need a producer to add an object in it so I am going to park this consumer thread in WAIT state by calling the wait method on this lock object. If it is not the case, it means that I can consume and object from my buffer and since I have made some room in my buffer, I can now notify all the producers that are in a WAIT state.

# Understanding How Synchronization Works with wait() and notify()

=>slides: Pg. 42

Let us explain that in more details. So I have a buffer that is empty for the moment. I have this wait list associated to my lock object. And this lock object has a key which is green. I have two instances of consumer and producer, with a consume and produce method guarded by the same key here in green. So let us consider an orange thread that is trying to run my consumer.

=>slides: Pg. 43

This orange thread is entering the synchronize block so it is going to take the key from my lock object. At the same time, the purple thread is trying to run my producer but it cannot enter this synchronized block because the key is not available.

=>slides: Pg. 44

At some point, my consumer thread will try to consume an object from my buffer but this buffer is empty for the moment. So since this buffer is empty, it will call the wait method on this lock object. The wait method has two consequences.

=>slides: Pg. 45

First, it is going to park my orange thread in the wait list and release the green key. And then the key becomes available for the purple thread to take it

=>slides: Pg. 46

so it is going to take it, run the produce method from my producer,

=>slides: Pg. 47

at some point, create some data and add this data to the buffer,

=>slides: Pg. 48

and then call the notify or notifyALL method, of this lock object. It has the consequence of taking my orange thread outside of the wait list, making it available to run again in my consume method.

=>slides: Pg. 49

At some point, the purple thread will exit the producing method, releasing the green key,

=>slides: Pg. 50

giving the chance to my orange thread to get the key and continue to run the consume method where it was.

=>slides: Pg. 51

At this point, it is going to take the object that has been added to the buffer,

=>slides: Pg. 52

then continue to run the consume method,

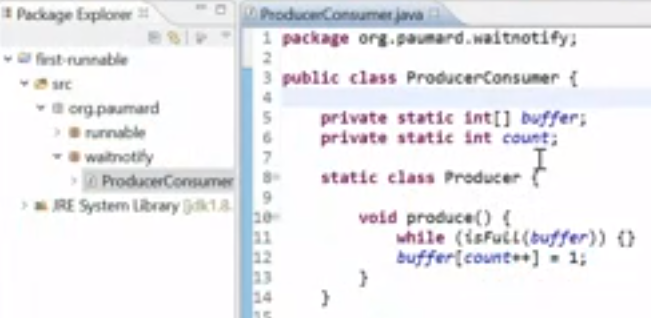
=>slides: Pg. 53

exit this method, thus releasing the key. This is exactly how the consumer/producer pattern works with the wait/notify pattern in Java.

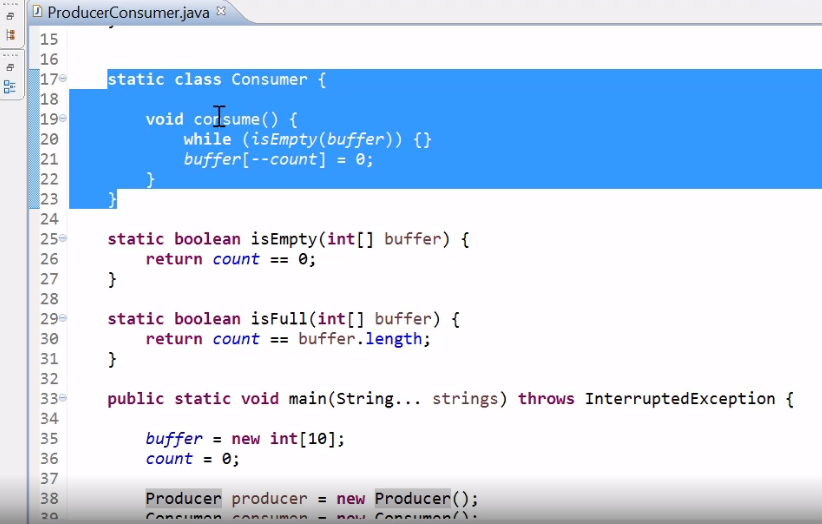
# Live Coding: Unsynchronized, Flawed Producer/Consumer in Action

=>slides: Pg. 54

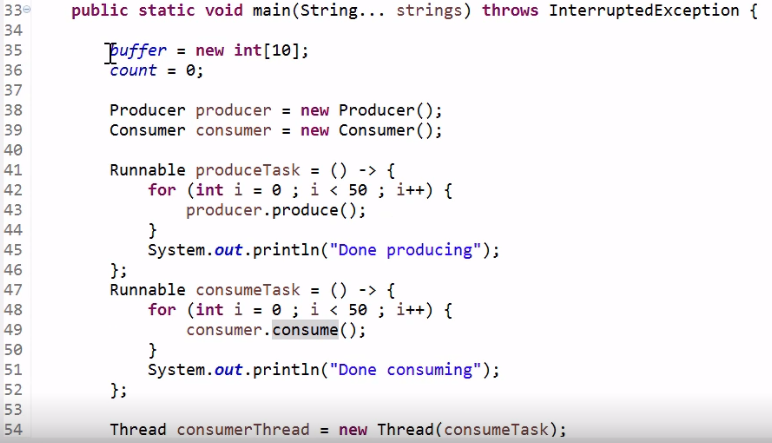
What are we going to see in this live coding session? Well first of all, we are going to see some code, of course, and we are going to see this producer/consumer pattern in action with the wait/notify that we just saw.

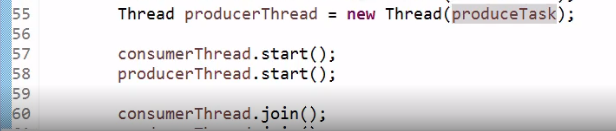


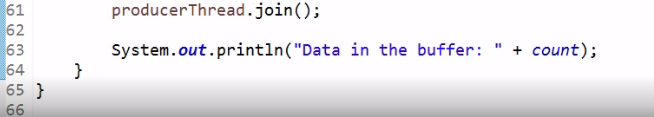
Let us see now, this wait/notify pattern on the producer/consumer. So what do we have here? We have a producer/consumer class with our buffer and our counter just as in the slide. We then have member class producer with one single method produce. Produce is just checking if the buffer is full and if the buffer is not full, it's just a data one inside this buffer.



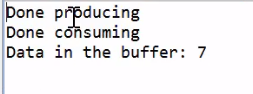
Then we have the same kind of class here, consumer, with only one method, consume. If the buffer is empty, it just waits on some data to be added in this buffer and it will just consume one element of data once there is one available.



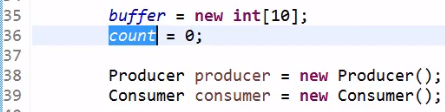




And then we have our main method, so we just initialize the buffer with 10 elements, count to zero, create a producer and a consumer. Then we create two tasks, one produceTask instance of the Runnable interface. It is going to produce 50 elements inside this for loop and we'll just print a message once this is done, Done producing. And the same goes for the consume task. The consume task is going to consume the 50 elements produced by our produceTask. And then, we have a consumerThread that is created on the consumeTask and a producerThread that is created on the produceTask. We launch our two threads here and here and then wait for them to complete using the join method. And at the end, we just show what data is left in the buffer by printing out the value of count. Of course, we expect in this case, count to be equal to zero so we expect the message Data in buffer zero, once we have run this code. So let us run it and check what is happening.



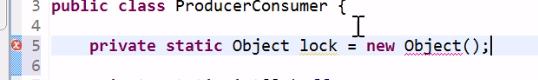
Well, our producer is done producing so it has produced 50 elements. Our consumer is also done consuming so it has consumed 50 elements. And the magic is there are seven elements left in our buffer. What is the reason for that? Well the reason for that is, of course, a race condition.



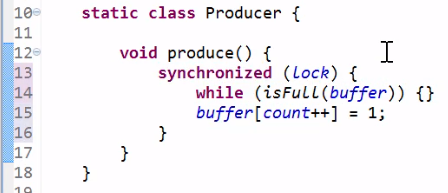
We have a races condition here, absolutely everywhere on the buffer and on the count object here.

# Live Coding: Synchronized, Flawed Producer/Consumer in Action

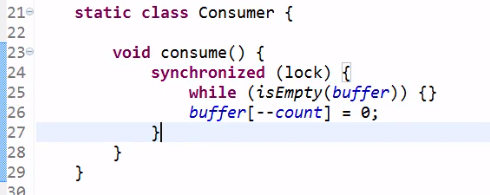
So one first way of fixing this race condition is to naively synchronize everything. Let us do that.



We need a lock object here. Private static object lock object. And we're going to use it to synchronize the production and consumption of our element



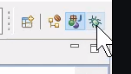
so here, synchronize them. Lock.



And synchronize on consume. Let us run this code again.



This time, it does not go too well because the JVM does not terminate. We don't have the message done producing or done consuming, meaning that the threads are still running. They are probably locked somewhere. So let us kill this and run it now in debug mode. So we are still locked.

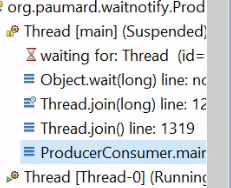


Let us switch to the debug perspective on eclipse.

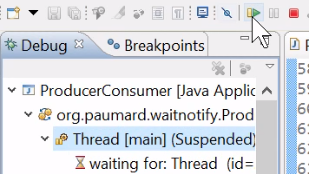




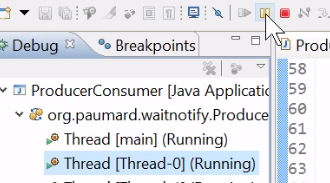
Let us see where our main thread is blocked.



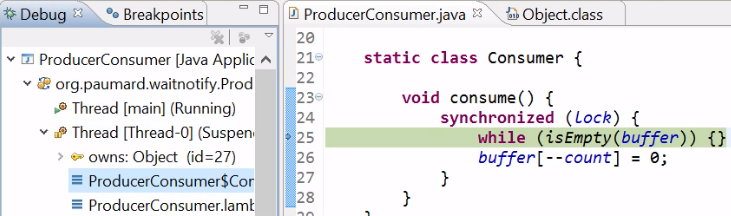
Well, it is blocked on the joining of the consumer so obviously, the consumer is still running.



Let us release the thread.



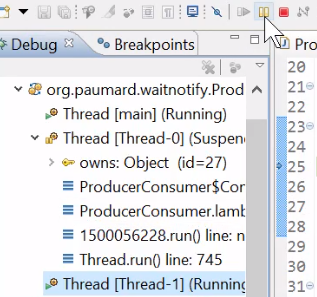
Let us pause thread zero,



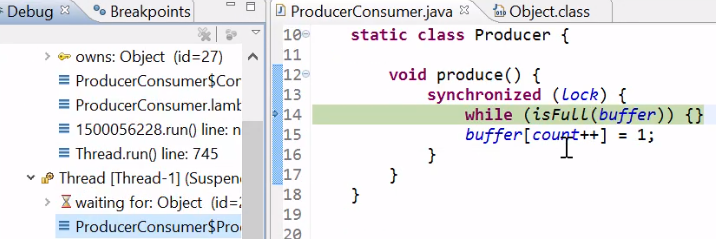
which would be the consumer thread and indeed it is.



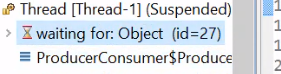
We can see that this thread is owning object ID 27, which is obviously our lock. And it's probably running again and again, this is empty buffer code here.



Let us see what is happening for the other thread. The producing thread.



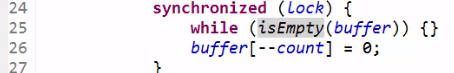
Well, this thread seems to be on the same method but the difference is that



it is waiting for the object



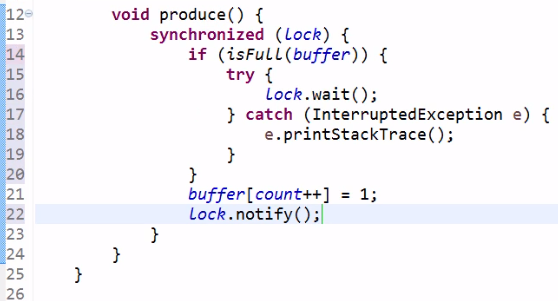
so it is not executing this line in fact, it is waiting for the key of the lock object and cannot execute while it's full buffer. Why can't it execute this code?



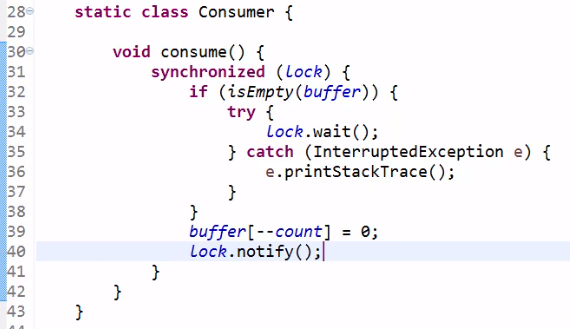
Well, because the key off the lock is held by the consumer that will never release it since the buffer cannot be filled by the producer.

# Live Coding: wait/notify, Correct Producer/Consumer in Action

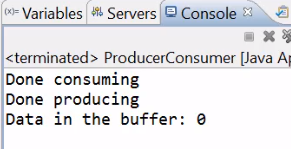
So let us kill this JVM. This solution does not work. The right solution is different from this one.

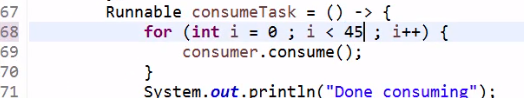


Inside the synchronize block, if the buffer is full here, then we are going to park this thread, releasing the lock. So lock. wait here. This wait method throws an interrupted exception so let us catch it like this. And once we've added some data in our buffer, we need to notify the threads that are waiting by calling the notify method.

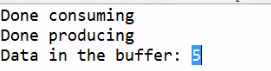


And we are going to do the same for the consumer so instead of a while here, we are executing an if. If the buffer is empty, then I need to wait for a producer to add some data in it. Let us handle the exception properly and notify once we have consumed a data. Let us run this code again



and this time, the data left in the buffer is indeed zero, which shows that our system is working properly. 

We can modify it a little by, for instance, consuming only 45 items. Let us run it again.



Indeed, there are five items left in the buffer, which is the expected result.

# Understanding the States of a Thread

=>slides: Pg. 55

We mentioned several times the state of the thread but we did not define what it is exactly so let us do this now.

=>slides: Pg. 56

A thread has a state, as we saw it. A thread can be running or not, which is already a first glimpse at what the state of a thread could be. The question is: If a thread is not running, can the thread scheduler give it the hand? That is, can the thread scheduler awake the thread and give it a time slice so that the thread can run its task? This question is interesting because it has, in fact, several answers. In most of the time, the answer is yes but we saw several cases in which the answer is no and namely, if the thread is in the wait list, then the thread scheduler should not try to give this thread a hand and a time slice to run its task. So the answer to this question is not that simple. It's not that obvious, and we need to know the exact different states a thread can have and what are the relationships between those states. So basically, we need to have a look at the state diagram to understand this exactly.

# Understanding the State Diagram of a Thread

=>slides: Pg. 57

The first state we have is the state New. When we create a thread by thread T equals new thread of some task, this thread is in the New state. It has not been run yet. It has not executed its task yet. The second state is the Runnable state. Once we call the start method on this thread, the thread is set in the Runnable state. It means that the thread scheduler is free to give a time slice of the CPU to the thread so that this thread can execute its task. And once the task is completed, this thread enters the Terminated state, in which the thread scheduler knows that the thread should not be run anymore. This is the basic state machine of the different states of a thread but as we saw, there are other states in which a thread can be. The first one is the Blocked state. When a thread is blocked at the entrance of a synchronized block because the key of the lock object is not available, it is in a Blocked state and the thread scheduler will not try to awake this thread as long as the key is not available. This is a very common state in which threads can be put. The second state is the Waiting state. We also saw the Waiting state. It is a state in which the thread is put once the wait method has been called. In this case, the thread is parked in the wait list and can be awakened only by a notify call as we saw in a previous section of this module. And then we have a last state that we did not see, which is called the Timed\_Waiting state. In fact, the wait method that we saw can also take a time out, which is a time out expressed in milliseconds. At the end of its time out, the thread will be automatically modified by the system and in this case, this thread will be awakened without calling the notify method on the right object. So in that case, the thread will be put in a Timed\_Waiting state. It is the same state as the one the thread is put when we call the sleep method on a given thread. So the thread can go from the Runnable state to one of those three states and can go back to the Runnable state depending on the actions taken on our system.

# Wrap-up on the State of a Thread

=>slides: Pg. 58

So if we take a look at what the thread scheduler can do, a thread scheduler can run the threads that are in the state Runnable. A blocked thread can only run again if the key is released. Remember that a blocked thread is blocked at the entrance of a synchronized block of code. So this block of code is guarded by a monitor that has a key. And a waiting thread can only run again when the notify method is called. It is very important to keep in mind that in the two last cases, if the key is never released or if the notify method is never called, then the blocked and the waiting threads will never be released thus, probably blocking our entire application.

=>slides: Pg. 59

We can read the state of a thread using a getstate method that is present on the thread class itself. It returns an enumerated value of type Thread. State.

=>slides: Pg. 60

This state enumeration is a member enumeration of the Thread class and is public so we have access to it and it defines the six states a thread can have, New, Runnable, Terminated, Blocked, Waiting, and Time\_Waiting.

# Wrap-up of the Module

=>slides: Pg. 61

Time to wrap up this module. What did we learn in this module? First, we saw precisely how the Runnable pattern works with the Runnable interface and the thread class. It is fairly simple to set up. We also saw how to set up a producer/consumer pattern. And we saw what can go wrong with it if the producer and the consumer are running in different threads, which is the case, of course, most of the time. And we saw how we can fix this pattern using synchronization and the proper use of the wait/notify pattern. That is all for this module.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*