12. Multi-threading and Concurrency

Contents

[1. Introduction 1](#_Toc15596)

[2. A Quick Look at the Basics 2](#_Toc129)

[3. The Move to Multithreading 2](#_Toc7910)

[4. Java Threading Foundation 6](#_Toc5563)

[5. Creating thread By Extending The Thread class 10](#_Toc21221)

[6. Thread Management Details 10](#_Toc12121)

[7. Concurrency Issues 11](#_Toc13138)

[8. What Is a Race Condition in Concurrent Programming? 11](#_Toc7128)

[9. Analysis of a Race Condition in the Singleton Pattern 11](#_Toc934)

[10. Synchronizing Code to Prevent Race Conditions 12](#_Toc11275)

[11. Understanding the Lock Object in Synchronization 14](#_Toc1951)

[12. Understanding Synchronization Over Multiple Methods 14](#_Toc23052)

[13. Live Coding: A Race Condition in Action, and How to Fix It 16](#_Toc18679)

[14. Coordinating Method Access 18](#_Toc18575)

[15. Manual Synchronization 19](#_Toc10225)

[16. Thread Priorities 20](#_Toc26044)

[17. Demo: Set the Priority for Threads 20](#_Toc30012)

[18. Summary 21](#_Toc5637)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to the next module of the course. In this module, we'll talk about working with multithreading and concurrency. We'll start by looking at the issue of single threading versus multithreading. We'll then look at some of the foundational types that Java provides for working with threading. We'll look at working with thread pools. Then we'll dig into some of the issues that come up as we run code in concurrent environments. We'll see the facilities that Java provides for simplifying the idea of coordinating method access across threads. We'll look at how to manually manage thread synchronization for those scenarios where we need to do so. And then we'll wrap up our discussion by looking at some of the packages and some of the other types that Java provides from managing concurrency inside of our applications.

# A Quick Look at the Basics

=>slides: Pg. 3

To get us started, let's clarify a few of the key terms and ideas we'll be using throughout our discussions in this module. The one key term to understand is process. Now process is an instance of a program or application. In other words, it's the thing that gets created when you launch your application. Now process has resources associated with it, things like memory and so forth, and it needs to do its job, and a process has at least one thread. Now a thread, technically speaking, is a sequence of programmed instructions. What that really means is that it's that thing that actually executes your program's code. In order for a thread to do its job, it utilizes process resources. So we'll look at this as a diagram.

=>slides: Pg. 4

So we'll go in here and say we launch our application. A process gets created. That process has resources allocated. That process runs over some period of time, and during that period of time, the thread is doing its work. And in standard Java, the process is launched with a single main thread. So if we pick a moment in time, and during that moment in time, the thread is interacting with some known set of process resources.

=>slides: Pg. 5

Now if we move into the idea of multithreading, a thread has the ability to create other threads. So our main thread at some point in time may launch another thread, and the main thread may later on launch yet another thread, and one of those threads may launch yet another thread. We're now multithreaded. Our process has multiple threads running inside of it, and at any given moment in time, there are up to four threads running at the same time. Now because these threads are running at the same time, we have the idea of concurrency, those concurrent operations or concurrent execution. So again, if we pick a moment in time, each of those threads are doing their job at that moment in time, but they're all using resources that belong to that single process.

=>slides: Pg. 6

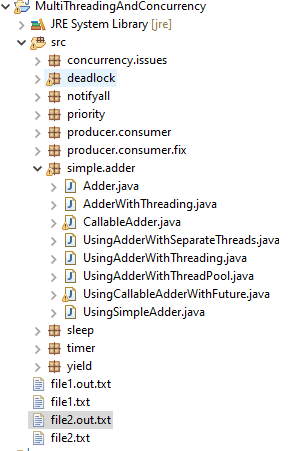
Thread 0 is interacting with some aspect of memory, Thread 1 is, Thread 2 is. All that is fine as long as they're not operating on the same resources, but maybe Thread 3 comes along and it needs to use some of the same resources that Thread 2 does. Well, now we've got issues because now we have problems that are related to concurrency, meaning that we need to coordinate the work that these threads do with the resources to make sure that they don't introduce errors. So we'll dig into this idea of concurrency a little bit later in the module and the facilities that Java provides for preventing those errors. But to get us started, we first need to understand how to make our Java programs multithreaded, and that's what we'll begin looking at in our next section.

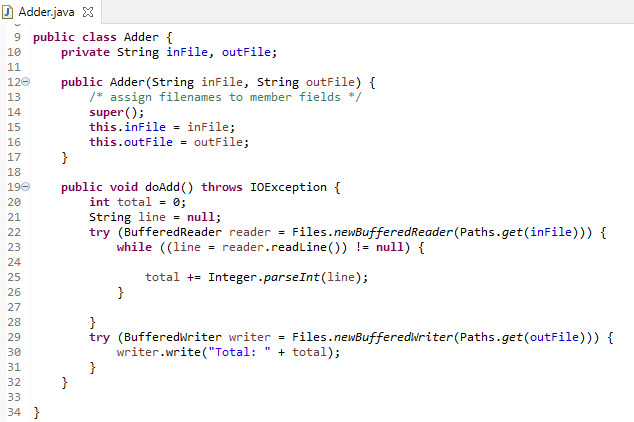
# The Move to Multithreading

=>slides: Pg. 7

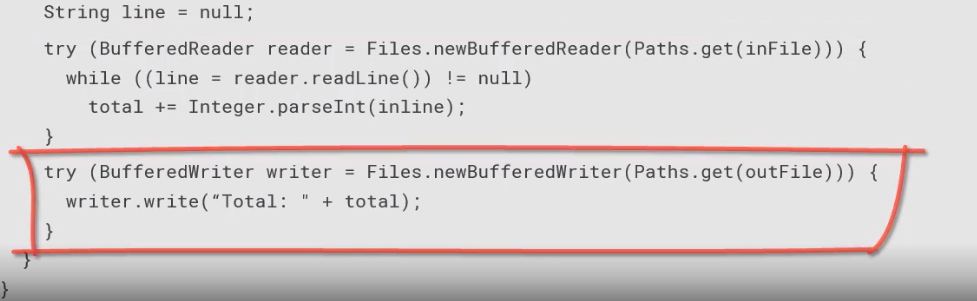
So why is multithreading important? As you might expect, multi‑threaded programming can be much more complicated than single threaded programming, so it must provide some value. And the most fundamental value is that enables more complete usage of our CPU resources. When a thread is running to do a programs work, oftentimes that thread is waiting for non‑CPU things that happen. Right, it's doing things like interacting with a storage device, interacting with the network, and so forth. So when that stuff's going on, the thread can't take advantage of the CPU, so our CPUs might be sitting idle. Also, most modern computers today have multiple cores inside of them. What that means is that most modern computers are built to run things in parallel. Pretty much every desktop computer or server computer is multiple core, and a really large percentage of smart devices now have multi‑core CPUs. So devices are built to run things in parallel, and we use multithreading to do that. And so what that really does for us is that when we use multithreading appropriately, the task that we run can actually take less time. And the reason I say less perceived execution time on the slide is that when we say less time we mean less wall clock time. Multithreading programming takes up however many processing cycles it takes to do the job. In fact, in general, multithreaded programming is going to take more CPU cycles to do the work than single threaded programming would. But because it can do that work in parallel, the amount of time it seems a process to take, or seems a task to take, is actually shorter. So with multithreading, we take full advantage of our CPU resources, and because of that, our work gets done more quickly.

=>slides: Pg. 8





To help us get a better sense of how multi‑threaded programming will work, let's take a look at a little bit of code here. I've got a really simple class here called Adder. And the job of the Adder class is to simply be given an input file, add up all the numbers inside that file, and then write the result out to another file. So in order to do that, we'll need a couple of fields for the file names. We'll go ahead and have a constructor that accepts those file names. We'll have one method we'll call, doAdd. And again doAdd's job is just to add up the numbers inside of a file and write them out. So we'll go ahead and have a look over here to total up the values of a local variable.



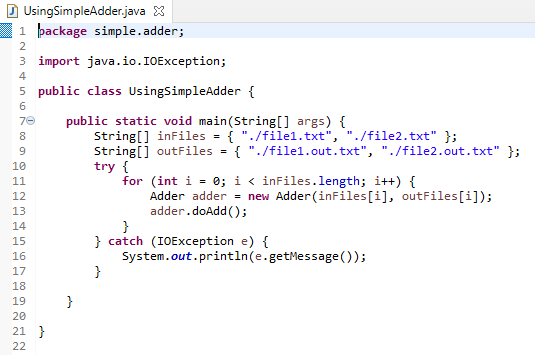
Here's a string which will actually represent the input line. Then to do the work, we'll go ahead and open up a BufferedReader over the input file. We'll loop through the file reading one line at a time, and then for each line we'll convert it into an integer, add it to the total. Once we've added up all the lines, we'll go ahead and open up an output file and then write the result out to that file.



Now, since we're interacting with the BufferedReaders, we know those guys throw IO exceptions, so our doAdd method declaration needs to show that it throws IOException. Since we're doing this IO work, that means we're dealing with IO devices, storage devices. Interacting with things like disks, and so forth, is a non‑CPU task, or at least it's not CPU focused, it relies on storage devices. So things like opening the files, reading from the files, writing to the files, are all things rely on devices other than the CPU. Those are cases where our thread is not taking advance of CPU resources, but relying on other resources. So there's an opportunity there where multithreading may come in handy.

=>slides: Pg. 9

**UsingSimpleAdder**



Now let's take a look at using this class in code. So I'm going to go ahead and just declare an array containing two file names, file1.txt through files2.txt. Another array containing 2 output file names. Then what I want to do is simply create a loop to walk through those arrays. For each pass through the loop, I'll create a new instance of our Adder class, passing in an input file and an output file. So I'm going to call adder.doAdd, in other words, do the work to add them up. Now, since doAdd is declared to throw IO exceptions, we need to go ahead and wrap this guy up in a try catch block to deal with that. All right, so that's really simple code.

=>slides: Pg. 10

Now, the way this is currently written, this is a single thread of application, right. So if we look at this conceptually, we have our process here, when our process starts up, the main thread does any startup work it needs to do, and then it does the work to use doAdd on file1, and then after that's done, it does the work to add up the contents of file,2. It does that for all the rest of files until it finally gets the file6. And then at the end, it does any clean up work. And all this work was done linearly. So it's done over a relatively long period of time because there's a lot of waiting going on. Remember that during the times we're working on file1, the CPU would be waiting for content to be read from a file, so the CPU is kind of idle. Also, there's probably multiple cores on this device. So we're doing this work one by one when there's really an opportunity to do things in parallel. So to take full advantage of this computer, what we should probably do is rework this from a single threaded application to one that takes advantage of multiple threads.

=>slides: Pg. 11

So the way that will look conceptually is that when we start up, our main thread does any startup work, but then it goes ahead and fires off a separate thread to process the contents of file1. It could then go ahead and fire off another thread to process the contents of file2. Do that for all the files inside there until gets to file6. While that work is going on, the main thread is free to do other work, waits for everything to get done. When all six files are done, it goes and does its cleanup work. So we've still done the same amount of processing, but the amount of time that's gone by is much less because we're taking full advantage of the CPU resources of the device.

=>slides: Pg. 12

Now the thing is that as we move to multithreading, there's something we need to be aware of. Multithreading is an explicit choice. In other words, we don't get it for free. That means we have to break our problem up into parts that can take advantage of multithreading. And then we need to hand off that work to the different threads to do it. Java provides different levels of abstraction for doing threading. It has some very direct handling available where we actually manually create and coordinate the work of our threads. But it also has some higher level abstractions which simplifies that process of creating and coordinating. We'll take a look at some of these different options throughout the remainder of this module, starting with direct thread handling in our next section.

# Java Threading Foundation

=>slides: Pg. 13

As we'll see, Java provides a variety of different levels of abstraction over the idea of threading. But the most foundational aspects of that provide a very limited abstraction. In other words, the basic threading types in Java are very close to the way threading works in most operating systems. Basically, each thread is started to do a specific task. When that thread finishes that task, the thread terminates. When we're working at this level, we're responsible for any kind of management that has to go on. We're going to be responsible to do any kind of coordination. That's all going to be our responsibility. And the exceptions that occur in a thread are tied to that thread, meaning that each thread is responsible to handle any exceptions that occur on a thread.

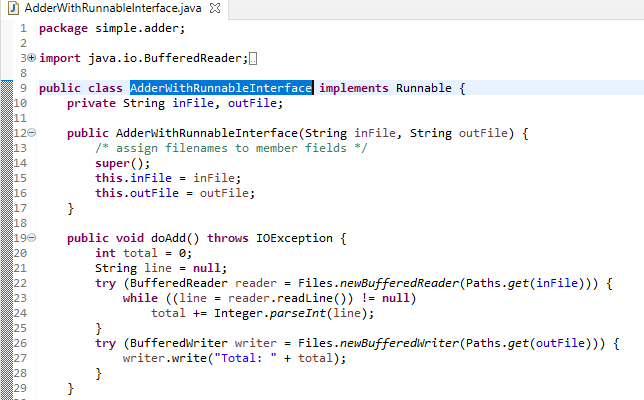
=>slides: Pg. 14

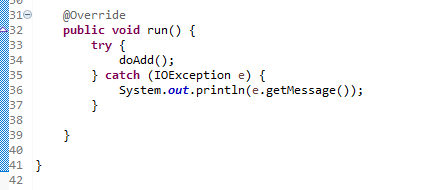
Now there are two core types that we use when working at this level. One is the runnable interface. The runnable interface represents some tasks to be run on a thread. Has exactly one member, the run method. The other type we use a lot at this level is the thread class. It represents a thread of execution. This class allows you to interact with that thread and effect a thread state. Now simply creating this as a thread class does not start the thread running. The thread doesn't start running until we call that thread's start method. Let's look now at how we can apply this idea of threading to the code we wrote earlier.

=>slides: Pg. 15

**AdderWithRunnableInterface**

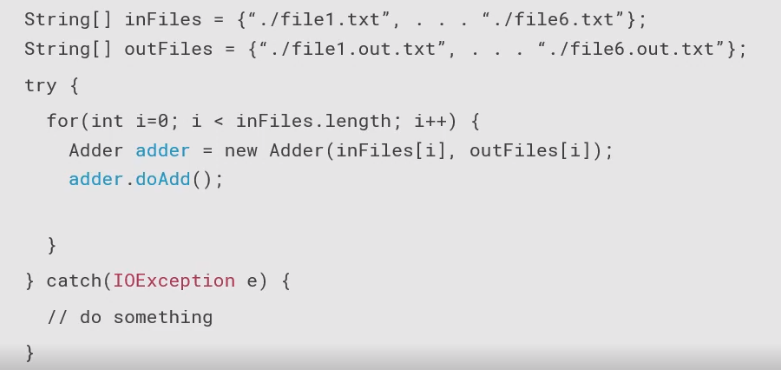
Remember that we wrote that class earlier called Adder. Remember that it had two fields in it, one for an input file, one for an output file. Had a constructor that accepted those file names. And then we had that method doadd that did the actual work. Remember that it took care of opening up the input file, reading through its contents, adding up everything inside the file, and then writing that total out to the output file. Well, we mentioned that this class was a good candidate for taking advantage of threading,





so let's go and update it to do so. Now, in order for this class to indicate that it can be run on an alternate thread, we need to go ahead and implement that runnable interface and implement the method on the interface, which is the run method. So this now indicates that this class understands how to be run on another thread. What we'll do from a run method is the work that we want to do on the other thread, which is simply call our doAdd method. Remember we said though that any work that happens on the threads is now a responsibility of that thread to handle the exceptions that occur. So we're going to have to go ahead and wrap our doAdd and our try‑catch inside the run method. So that simple change of implementing the interface now indicates that our Adder class knows how to be run on an alternate thread.

=>slides: Pg. 16

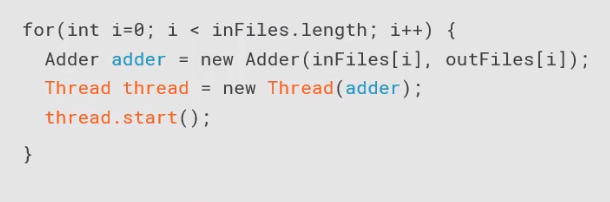


Now let's go to our main application code. Now remember that our application code right now is written to use our Adder class single threaded. Remember that we have the arrays of input files and output files. We have a loop to loop through those files, we go ahead and create the instance of the Adder class. We directly call doAdd on it, and we wrap all that in a try‑catch to handle the exception. This is our single‑threaded implementation.

=>slides: Pg. 17



To move this to take advantage of the multithreading capabilities of our Adder class, what we're going to do is that, instead of calling adder.doAdd directly, what we'll do instead is create an instance of our thread class, passing Adder to the constructor. The thread constructor expects an implementation of runnable. Adder implements runnable so we can pass that to the constructor, and then we can call thread.start. And that will now run each instance of the Adder class on a separate thread. So we're looping through each of the six file names, creating a new Thread for each one with a separate adder instance for each one. So each of the six files now are processed on six separate threads. Remember we said, though, that the exception is the responsibility of the thread.



So since the background thread is handling the IOException that the adder.doAdd method might throw, we'll get rid of the try‑catch block in our code here, The code that we have written currently will take care of running each of the Adder instances on a separate thread, but there is one potential problem if this is our main application code.

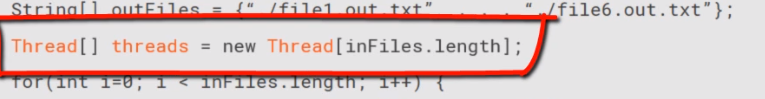
=>slides: Pg. 18

So let's take a look at the diagram representing what's going on here. So we have our process. Our main thread starts up, and then what's it do? It starts firing off other threads. So it fires off one thread to handle the first file to work on, fires off another thread to handle the other file work on, and it does that for each of the six files until it has six threads running six Adder instances working on the six different files. Once that's all done, the main thread has no more work to do. So because the main thread has no more work to do, in some instances, the background threads may never get a chance to finish. Because in some instances, once the main thread terminates, the entire process gets cleaned up. So what we want to do is make sure that all of our background work gets done. So what we want to do is take our process, do the same work we were doing before, go ahead and start up, fire off those background threads to do work on all the six different files.

=>slides: Pg. 19

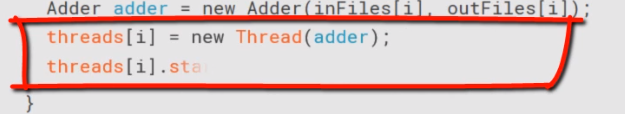
But what we want to do is make sure that our main thread waits for the other threads to finish their work and then shuts down. That's a very simple change to our application. So let's go back to our code.

=>slides: Pg. 21

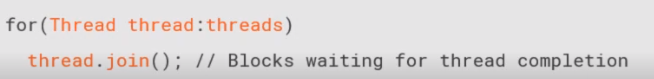




Currently, we have a local thread variable for each of the threads that are created. What we're going to do instead now is create an array of threads. So now we have the ability to store references to multiple threads.

So then our code here that has the local thread variable, we're going to change that so that each time we create an instance of our Thread, we'll add that reference into the array, we'll of course go ahead and do the start against the references. But now, once those six threads are created, we've got references to each of the thread objects.

=>slides: Pg. 21



So what we can do is that after they're all started, we can have another loop that goes through each those thread references and calls join on each of the threads. What that causes to have happen is that this calling thread will block until the other thread finishes. In other words, our main thread, as it calls thread.join to those threads, the main thread will block until the background thread finishes. If the background thread is already finished when we call join, join returns right away. If the background thread is not done running, then our call to join will block until that background thread finishes. This assures that our main thread continues running until all of the background threads are done doing their work. As you can see, when we're working at this level, we're responsible for a lot of the details. So in our next section, we're going to take a look at some of the types that are available that will abstract some of those threading details from our code.

# Creating thread By Extending The Thread class



# Thread Management Details

=>slides: Pg. 22

In the previous sections, we looked at the Thread class. And the Thread class is a really powerful class because it gave us very direct control over threads, their creation, coordination, startup, shutdown, that sort of thing. But with that kind of control, it presents a challenges. It's that we're responsible to use those capabilities effectively. And if you're not a seasoned threading programmer, it's really easy to misuse threads if you're not familiar with how to do that. And honestly, most of us don't want to be caught up in those details anyway. What we really want to do is just have the advantage and capabilities of threads, but let kind of the details be taken care of for us.

1. Concurrency Issues  
   =>slides: Pg. 23

Up until now, the multi‑threading we've been looking at has been relatively independent. What I mean by that is that each thread that we've looked at has been doing a job that's been relatively independent from the job performed by the other threads. Remember, early in this module, we mentioned that as we get into multi‑threading, we sometimes get into concurrency issues, and I said we'd talk about those a little later. Well, we're now at the time where we talk about them. The issue is that concurrency can create challenges because once threads start to share common resources, you have the potential for problems. Now, if those threads are sharing a resource and they're only reading those resources, in other words, they never change them, they only look at what's inside of them, there's really not much issue there. But as soon as threads start to change shared resources, we've got to coordinate that work because if we fail to coordinate that work, we're going to have problems. At a minimum, we can start receiving incorrect results. But things can actually get a lot worse than that, programs can start crashing, kind of just crazy things can start happening.

1. What Is a Race Condition in Concurrent Programming?

=>slides: Pg. 24

We defined what a thread is, what it can do. We saw that creating a thread is about doing several things at the same time. Let us talk about a very important notion called race condition.

=>slides: Pg. 25  
What is a race condition? Well, a race condition deals with the access of data concurrently. What does it mean accessing data concurrently? It means that two different threads might be reading the same variable, the same field defined in a Java class, or the same array. And this concurrency may lead to issues, we are going to see it. A race condition occurs when two different threads are trying to read and write the same variable or the same field, at the same time. All those words are important. Several threads can be able to read the same variable at the same time, if the value of this variable does not change, it will not raise an issue. But if they are reading and writing the same variable, then it may raise a problem. And you see that this notion of same time really needs to be very precisely defined, because on a monocore, or on a multicore CPU, same time does not mean the same thing as we saw in the previous example. This concurrent reading and writing is what is called a race condition. And as we saw, same time does not mean the same thing on a single core or on a multicore CPU. And we are going to see that.

1. Analysis of a Race Condition in the Singleton Pattern   
   =>slides: Pg. 26

Now let us see an example with the very well-known singleton pattern.

=>slides: Pg. 27

Most of the time the singleton pattern is written like that. The idea is to have a class that is allowed only one single instance. So this instance is told in a private static field, here called instance. The constructor of this class is made private, so that it is not possible to build this class outside of itself. And we have a get instance, public static method, that will first check if the instance of the singleton class has already been created. If it has, it is just returned, and if it has not, it created, and returned this instance.

=>slides: Pg. 28

Now the question is, what is happening if two threads are calling this get instance method at the same time?

=>slides: Pg. 29

Let us take a closer look at that. Suppose that the thread scheduler chose T one first. So T one is running its task on the CPU, and the thread T two is waiting for the thread scheduler to give it the hand, that is, to give it a time slice, so that T two can run its task. First, thread T one will check if the field instance is known, because this is the test written in the code. The answer is yes because instance has not been initialized yet. So it will enter the if block. And what happens here is that the thread scheduler pauses T one and gives the hand to the thread T two.

=>slides: Pg. 30

The thread T two will just do exactly the same, check if the field instance is null. And the answer is still yes. Why, because thread T one has not initialized this instance field yet. So it will also enter the if block. And since it is in the if block, it can now create an instance of the singleton class, and copy that instance in the private static instance field. And at some point the thread scheduler will give the hand back to T one. And since T one is also inside the if block, it will not check if the instance field has been initialized one more time, why? Because it already did that,

=>slides: Pg. 31

it will create another instance of singleton, and copy it in the private static field instance, thus erasing the instance that has been created by the thread T two. This is a well-known race condition case, and we are going to see how to prevent that.

1. Synchronizing Code to Prevent Race Conditions  
   =>slides: Pg. 32

So the question is how to prevent that? And in the Java language, the answer is very simple, it is called synchronization. By synchronizing a block of code, we are going to prevent this to happen. =>slides: Pg. =>slides: Pg. 33

Synchronization prevents a block of code to be executed by more than one thread at the same time, and from a technical point of view, it will prevent the thread scheduler to give the hand to a thread that wants to execute the synchronized portion of code that has already been executed by another thread.

=>slides: Pg. 34

How does it work, technically? Well it's pretty simple, we just have to add the synchronized keyword on the declaration of the method, that becomes public static synchronized singleton get instance.

=>slides: Pg. 35

How does synchronization work under the hood? This is what we are going to see now.

=>slides: Pg. 36

The singleton class is a class with a get instance method that we want to synchronize.

=>slides: Pg. 37

If it is not, any thread can run this method freely. Synchronizing means protecting this method by a fence here, a green fence,

=>slides: Pg. 38

declared with the synchronized keyword. Under the hood, the Java machine uses a special object, called a lock object, that has a key. In fact, every object in the Java language, has this key, that is used for synchronization. What does it change to our code?

=>slides: Pg. 39

Well, when a thread want to enter this protected method, this protected block of code, it will make a request on this lock object, give me your key.

=>slides: Pg. 40

If the lock object has the key available, it will give it to this thread, and this thread will be able to run the get instance method freely.

=>slides: Pg. 41

If another thread wants to enter this synchronized block of code, it will make the same request on the lock object, but this time, the lock object has no key available for him. Why? Just because he already gave its key to the red thread. The lock object has only one single key. So the blue thread has to wait for the key to be available.

=>slides: Pg. 42

At some point, the red thread will finish to run the get instance method, will give back the key to the lock object,

=>slides: Pg. 43

so this lock object will be able to give the key to the blue thread and the blue thread will be allowed to run a get instance method.

=>slides: Pg. 44

When it has finished running this method, it will give back the key to the lock object. This mechanism is very simple, and will prevent more than one thread to execute the get instance method at the same time.

1. Understanding the Lock Object in Synchronization   
   =>slides: Pg. 45

So for synchronization to work, we need a special technical object that will hold the key. We are going to see that this object can be made explicit in our code. In fact, every Java object can play this role. This key is defined internally in the object class, thus making it available for all the Java objects we define in our applications. It is worth noting that sometimes this key is also called a monitor in books and articles. Now we may ask the question, how do we know which object has been chosen to hold this key? Well in fact, there are several cases to consider.

=>slides: Pg. 46

In our example, we put the synchronized keyword on a public static method of the singleton class. And in this case, the JVM uses the singleton class object itself. All the classes in Java are represented by objects. And in the case of a synchronized static method, the object chosen to hold the key is the class object itself.

=>slides: Pg. 47

If we put the synchronized keyword on a non-static method, then in this case, the key is held by the instance of the class we are in. So a synchronized, non-static method, uses the instance it is in, as a synchronization object holding the key.

=>slides: Pg. 48

A third possibility is to use a dedicated, explicit object to conduct synchronization. Here we are creating a private final object called key. It doesn't have to be an instance of a special class. In fact, the object class itself is enough. And this key object will be used as a synchronization object holding the key. So instead of synchronizing the init method, we can create the synchronized block inside this method, and pass this key object as a parameter of this synchronized keyword. And this is probably the best thing to do, because it is always a good idea to hide the object used for synchronization, whether we are in a static context or not.

1. Understanding Synchronization Over Multiple Methods  
   =>slides: Pg. 49

Now, let us have a look at some use cases and corner cases. Suppose we have a person class. Here we are looking at an instance of his class called Mary. And we declared two methods in this class, getName and getAge. And we added the synchronized keyword on the declaration of those methods. The lock object used by the JVM is the Mary object itself, that is, the instance of the class we are in. =>slides: Pg. 50

So what is going to happen if a thread wants to execute getName? Well, it will just take the key from the lock object, and remember, the lock object is the Mary object itself,

=>slides: Pg. 51

thus preventing a red thread from executing getAge at the same time. Why because since we did not declare any explicit object on the synchronization of our methods, the same key is used to synchronize both methods. This might not be what we need. If we need to synchronize getName independently of getAge, then we need to create two lock objects in the person class, and synchronize the block of codes inside the methods on those two different objects.

=>slides: Pg. 52

But now suppose that we have two instances of our person class, Mary and John. And once again, the getName and getAge method are synchronized using the synchronize keyword on the method declaration.

=>slides: Pg. 53

It means that the instances Mary and John are used to synchronize those methods. So we have two lock objects with two keys.

=>slides: Pg. 54

The thread executing the getName of the Mary instance object,

=>slides: Pg. 55

does not prevent a thread from executing the getAge from the John instance object.

=>slides: Pg. 56

And it will not prevent another thread to execute this same getName method on another object. You really need to keep in mind that, to understand how synchronization works, you need to identify which object is used as a lock, and what are the keys used in your application. Remember that using the synchronized keyword on a method declaration, uses an implicit lock object, which is the class object in the case of a static method, or the instance object itself in the case of a non-static method.

=>slides: Pg. 57

If what we really want is to prevent two threads to execute the getName method at the same time, in all the instances of the person class, then we need our lock object to be bound not to each instance of our class, but to the class itself. So it has to be the static field of the class person itself.

=>slides: Pg. 58

And in this case indeed, the blue thread executing the getName method from the Mary object will be holding the key.

=>slides: Pg. 59

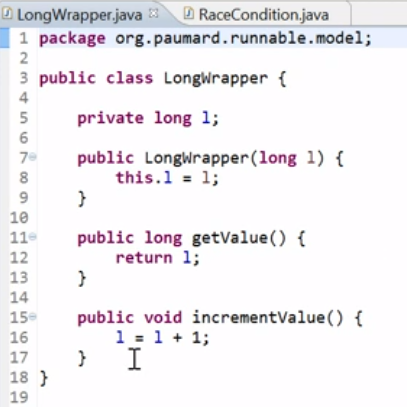
Thus preventing a red thread from entering the getAge method,

=>slides: Pg. 60

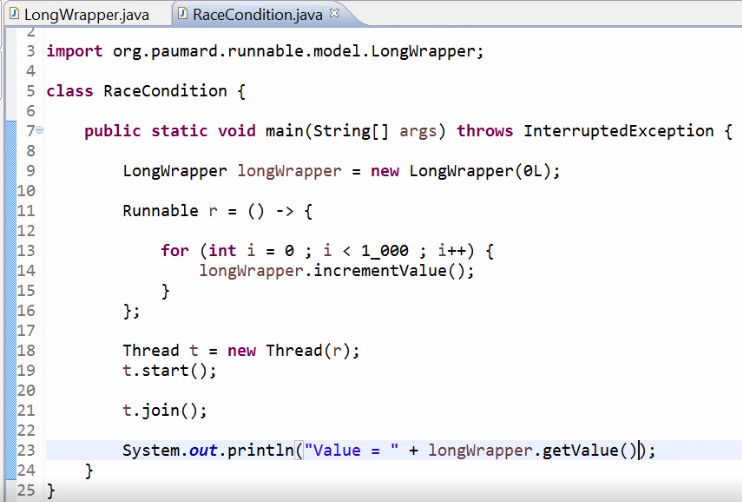
but also a purple thread to execute the getName method of the John instance of the person class.

1. Live Coding: A Race Condition in Action, and How to Fix It

=>slides: Pg. 61



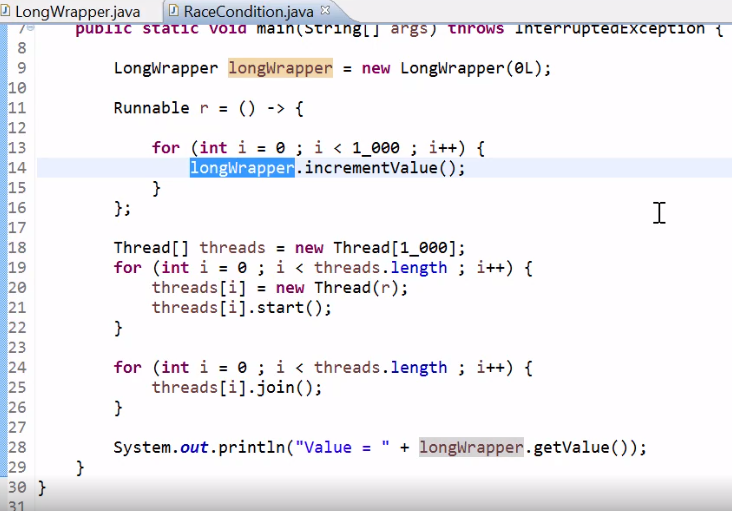
For this second example, I have created a special class, Longwrapper. This Longwrapper is just a very simple class, wrapping a long value. I have a method get value that returns the current value, and increment value that just increments this value.



Let us create now a race condition, using this Longwrapper class. I am going to create a Longwrapper object, = new Longwrapper, let us give it the value zero. I am going to create a runnable using a lambda expression. And this runnable is going to increment... This Longrwapper a thousand times. Let us run this runnable in a new thread. I am going to call the join method here. That throws an InterruptedException. Just to be sure that the code that I'm going to write after the call to this join method is executed once this thread has finished executing this runnable. This is just a trick here. And I am going to print out the result, value equals Longwrapper. getValue. Let us run this code.



Of course the result is the expected result, 1000. Now suppose I modify this code, and instead of using just one thread to run this runnable, I am going to use a thousand threads.



The code becomes the following, I have a for loop here. And I am going to create a new thread for all the content of this array, start all of them, and then join all of them to be sure that they have all executed the runnable correctly. So now I have a thousand threads, incrementing a thousand times, the value in my Longwrapper. So I expect the result to be one million.



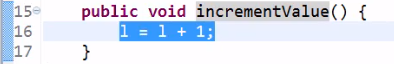
Let us execute this code. The result is not one million.



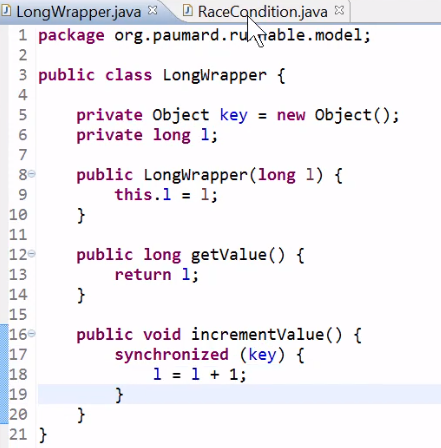
And if I run it several times,



I can see that, the result is not always the same. And if you run the same code on your own machine, there is no reason that you will observe the same result as me. Where does it come from?



Well, it is a very classical example of a race condition. Here in the increment value method, I have a read of the l value, I increment it locally, and copy it into the same field l. So this read is followed by a write operation. So this operation is basically a read and write operation from different threads at the same time, so this is a race condition. How can I fix this code?



Let us create a key object. And let us synchronize this block of code on this key object.



And this time, if I run my code, I will have the correct value, which is one million.

# **Coordinating Method Access**

=>slides: Pg. 62

As we saw in our last practice , there are times where it's really important that we coordinate concurrency. Now Java provides a few different ways to do that. One tool it provides is something called synchronized methods. Now what synchronized methods do is they coordinate thread access to methods. Now any method could be made synchronized by simply adding the synchronized modifier to that method. And a class can have as many synchronized methods as it needs. Now, method synchronization is not managed at the method level, it's actually managed at the class instance level. And what that means is that if you have an instance of a class and one thread calls into a method that's marked as synchronized, no other thread can call into any method marked as synchronized on that same class instance.

=>slides: Pg. 63

So now, as we think about using this idea of synchronized methods, when do we use these? Well, one example that comes to mind pretty quickly is the idea of protecting modification by multiple threads. Right, we saw in our last that things got really mucked up pretty good because multiple threads were trying to modify the same value at the same time. So that's an obvious case. Another case where it's important to utilize synchronize methods is that when we're reading values that might be modified by another thread at the same time we're reading them, it's possible that the value we're reading actually gets corrupted as we're reading it in as another thread is modifying it. There are a few caveats where that's not always necessary. But in general, that's a really good idea. So protect both modification of values, as well as the reading of values that might be modified by another thread. Now you might be thinking that, wow, this synchronization is pretty cool stuff. Why don't I just synchronize every method? And the issue is that this synchronization has a fairly high overhead. It takes a fair bit to go on and actually make all this work. So you really only want to use these synchronized methods in multi‑threading scenarios that require it. Now one thing we want to note here is that constructors are never synchronized because any given object instance is created on exactly one thread. And remember that synchronization occurs at the instance level. All right so there's no way for two threads to create the same instance at the same time. So because of that Java won't even allow us to mark constructors as synchronized.

# **Manual Synchronization**

=>slides: Pg. 64

Let's look now at the idea of manually managing synchronization. Now in the last , we looked at synchronized methods. And what synchronized methods did for us was they automated concurrency management. And what they were actually doing was whenever we called a method against an instance, if that method was synchronized, that call into the method automatically acquired a lock against the current object instance. Well, it turns out that objects do not have to have synchronized methods to have a lock. All Java objects have locks. And because all Java objects have locks, we can actually manually acquire that lock anytime we need to. When we manually acquire a lock, we're using what are called synchronized statement blocks. And the value of synchronized statement blocks is that any code that has a reference to an object can use a synchronized statement block to acquire a lock against that object.

=>slides: Pg. 65

So why bother with synchronized statement blocks if it did the same thing with synchronized statement block we did previously with synchronized method and synchronized method seemed easier? Why do we have these synchronized statement blocks? Well, what it comes down to is that synchronized statement blocks provide flexibility. One thing it allows us to do is use non‑thread safe classes in a thread‑safe way so that even though our classdoes not have synchronized methods, we could still safely use it multithreaded because the synchronized statement block locked access into that account instance. It also makes it easy for us to protect complex blocks of code. If all we have are synchronized methods, that means every time we needed synchronization, we'd always have to write a method to go do it. Synchronized statement blocks allow us to protect code right in the middle of another method, so it allows us to protect blocks of code without having to move them off into their own method. And what it comes down to is that even with synchronized methods, sometimes they just aren't enough. Sometimes we need something more sophisticated than that. In our next , let's take a look at some code that demonstrates where synchronized methods may not be quite enough.

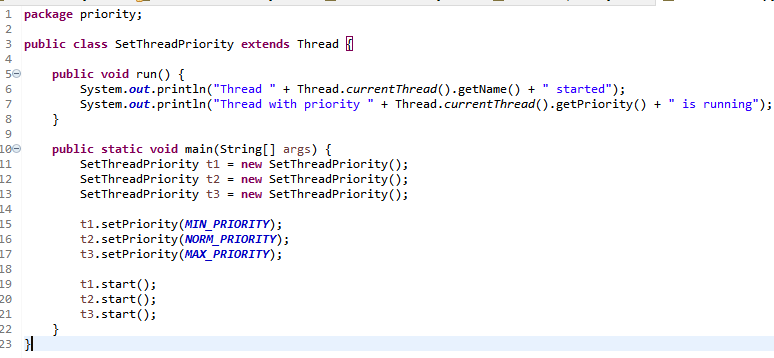
# Thread Priorities

=>slides: Pg. 66

The Thread class has 3 constant variables that define the thread priorities. We can assign priorities for the user threads else the system automatically assigns default values.

# Demo: Set the Priority for Threads

=>slides: Pg. 67  
This shows how we can set the priority for threads using constant variables like ****MIN\_PRIORITY****, ****NORM\_PRIORITY****, ****MAX\_PRIORITY****. The thread that has maximum priority will be initiated first, else it depends on the JVM implementation as well. In this example, we can see that though the threads were started in a different order, it executes the thread with priority 10 first, followed by priority 5, and then finally with priority 1.



# Summary

=>slides: Pg. 68

To wrap up, here are some of the key things you want to remember from this module. We started out by looking at the Thread class, and that represented a thread of execution. Now it's very similar behavior to the way most operating systems represent threads. And when we use the Thread class, we're really responsible to handle most of the threading details ourselves. So it's a very, kind of a low‑level class. It's what a lot of the other things are built on. We can create a thread by directly extending the Thread class. And then we had the Runnable interface, which represented a task to run on a thread. And it was very simple. It had one method, which is the run method. Any code we wanted to run on a thread would go inside of that method. But again, it's a fairly foundational implementation in the sense that it can't return results, and the thread is responsible for any exceptions that occur there. So when working with the Thread class in the Runnable interface, we're going to have a lot of responsibilities to handle on our own.

=>slides: Pg. 69

And then we have the issue of thread synchronization. And as we saw, all Java objects have a lock. Now one way to get access to that lock is by using synchronized methods. When you use a synchronized method, you automatically acquire the lock by calling into a synchronized method, and that method acquires a lock on its current instance. And by using synchronized methods, only one synchronized method on a given object instance can be active at one point in a time. So it actually kind of automates that idea of calling into the method, acquiring the lock, doing the work, and then releasing it. Well we saw we could also manually acquire the lock, and we could do that using synchronized statement blocks, which allowed us to protect a block of code using the lock on an object. And that's available to any code that actually has a reference to the object. And as we saw, there are scenarios where managing synchronization with synchronized methods wasn't easily achievable. So the synchronized statement blocks allow us to take more control over that.

And finally, we discussed Thread priorities. Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only the JVM but also a Java programmer can also assign the priorities of a thread explicitly in a Java program.

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