

Understanding, debugging, and optimizing JVM applications

HOW TO READ Java

Laurențiu Spilcă



MANNING



MEAP Edition
Manning Early Access Program
How to Read Java
Understanding, debugging, and optimizing JVM applications

Version 8

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *How to Read Java*.

You've made a significant step forward in understanding how essential it is to grow your skills in investigating how an app works. Through years of experience, I observed I spent much more time trying to understand code, rather than writing it. I found myself asking many times "Why does the app work this way?"

I had the unpleasant but lucky chance of having to work both on dirty codebases as well as on complex systems. In the beginning, I was spending days, sometimes, figuring out what an app does and what happens behind the scenes. Sometimes the entire investigation ended with a small fix - correcting one line of code. Such apps taught me well how to find faster ways of understanding how an app executes, and, with time I made my work a lot more efficient.

In this book, I describe the techniques I learned throughout many years of experience, sprinkling valuable notes and tips throughout. By learning these approaches, you'll save valuable time and become more efficient in both solving app problems and implementing new app capabilities.

By the end of the book, you will have covered the following topics:

- Using a debugger efficiently by embracing various techniques that'll help you understand code faster.
- Correctly implementing and using application logs.
- Finding places in an app's execution that are slow and can be optimized.
- Profiling to evaluate the SQL queries an app uses to manage data in a database.
- Evaluating heap dumps to find memory leaks and make an app's memory management more efficient.
- Using thread dumps to identify root causes of deadlocks, and to find ways to solve them.
- Using advanced visualization techniques of an app's execution, such as call graphs and flame graphs.
- Using tools and techniques to easily follow and comprehend the execution of a service-oriented or microservices system.

The book is primarily for developers using JVM languages such as Java, Scala, or Kotlin; the examples provided with the book are in Java. However, many of the things discussed (especially in parts 1 and 3) are valuable for developers using different languages.

Thank you again for your interest and for purchasing the MEAP!

If you have any questions, comments, or suggestions, please share them in Manning's [liveBook Discussion Forum](#).

—Laurențiu Spilcă

brief contents

PART 1 – THE BASICS OF INVESTIGATING A CODE BASE

- 1 Revealing app's obscurities*
- 2 Understanding your app's logic through debugging techniques*
- 3 Finding problem root causes using advanced debugging techniques*
- 4 Finding issues' root causes in apps running in remote environments*
- 5 Making the most of logs: Auditing app's behavior*

PART 2 – DEEP DIAGNOSTICING AN APP'S EXECUTION

- 6 Identifying resource consumption problems using profiling techniques*
- 7 Finding hidden issues using profiling techniques*
- 8 Using advanced visualization tools for profiled data*
- 9 Investigating locks in multithreaded architectures*
- 10 Investigating deadlocks with thread dumps*
- 11 Finding memory-related issues in an app's execution*

PART 3 – FINDING PROBLEMS IN LARGE SYSTEMS

- 12 Investigating apps' behavior in large systems*

APPENDIXES

- A Tools*
- B Opening a project*
- C Recommended further study*
- D Threads*
- E Memory of a Java app*

1

Revealing app's obscurities

This chapter covers

- What is a code investigation technique?
- What code investigation techniques we use to understand Java apps

A software developer has various responsibilities. Most of these responsibilities depend on how they understand the code they work with. Most likely, analyzing code to find out how to correct issues, implement new capabilities, and even learn new technologies is an activity that takes a software developer most time. And time is precious. So you need efficient investigation techniques to be a productive software developer. Learning how to be efficient in understanding your code is the main topic of this book.

NOTE Software developers generally spend more time understanding how the software works rather than writing code to implement new features or correct errors.



Often, software developers use the word "debugging" for any investigation technique. Actually, "debugging" is only one of the various available techniques for examining logic implemented as code.

Be aware that you'll often hear the term "debugging" used by developers. While this term should mean only "finding out issues to solve them", developers use it to name different purposes for investigating how code works:

- Learning a new framework
- Finding the root cause of a problem

- Understanding existing logic to extend it with new capabilities

1.1 How to more easily understand your app

I want you first to understand what investigating code is and how developers understand this activity. In the next section, we look at several commonly encountered scenarios where you can apply the techniques you'll learn in this book.

Briefly, I describe investigating code as being the process of analyzing a software capability's specific behavior. You might wonder, Why such a generic definition? Which is the investigation purpose? At the beginning of software development, looking through code had one precise purpose: finding and correcting software errors (which we name bugs). This is actually the reason why many developers still use the term "debugging" to name any of these techniques.

Just take a look at the way the word "debug" is formed:

de-bug = take out bugs, eliminate errors

In many cases today, we still debug apps to find and correct errors. But unlike the early days of software development, apps today are more complex. In many cases, developers find themselves investigating how a particular software capability works, simply for learning a specific technology or library. Debugging is no longer only about finding a particular issue but for correctly understanding its behavior (figure 1.1).

Why do we analyze code in apps?

- To find a particular issue
- To understand how a particular software capability works so we can enhance it
- To learn a specific technology or library



Figure 1.1 Code investigation is not only about finding problems in software. Today, apps are complex. We often use investigation techniques to understand an app's behavior or simply to learn new technologies.

Of course, maybe not all of us, but many developers also investigate code for leisure. Exploring how code works is fun, can sometimes become frustrating as well, but nothing compares with the feeling of finding the root cause of an issue or finally getting how things work (figure 1.2).



Figure 1.2 It maybe doesn't imply so much physical effort, but debugging sometimes makes you feel like Lara Croft or Indiana Jones. Many developers enjoy the unique sensation of solving the puzzle of a software issue.

Furthermore, we have various investigation techniques we apply to investigate software's behavior (figure 1.3). As we'll discuss later in the chapter, developers (and especially beginners) often wrongly consider that "debugging" is the action of using a debugger tool. The debugger is a software program you use to read and understand easier the source code of an application usually by pausing the execution on specific instructions and running the code step by step. Using a debugger is a common technique used to investigate software's behavior (and usually the first one a developer learns). But using a debugger is not the only technique you can use, and it doesn't help you in every scenario. We'll discuss the standard and the more advanced ways of using a debugger in chapters 2 and 3.

Figure 1.3 presents the various investigation techniques you'll learn throughout this book.

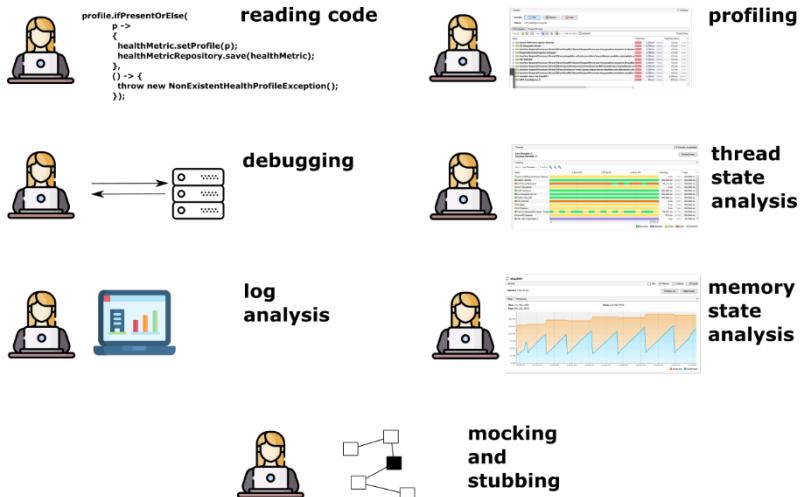


Figure 1.3 Code investigation techniques. Depending on the case, a developer would choose one or more of these techniques to understand how a certain capability works.

Software developers generally spend more time understanding how the software works rather than writing code to implement new features or correct errors.

When a developer solves a bug, they spend the most time on understanding a particular feature. The change they end up making sometimes reduces to one line of code – a missing condition, a missing instruction, or a misused operator. It's not writing the code that spends a developer's time. Understanding how the app works occupies most of a developer's time.

In most cases, simply reading the code is enough to understand it. Anyway, reading code is not like reading a book. When we read code, we don't read nice short paragraphs written in a logical order from top to bottom. Instead, we step from one method to another, from one file to another – we sometimes feel like we advance in a vast labyrinth and get lost. As a side-read on this subject, I recommend the excellent book written by Felienne Hermans, *The Programmer's Brain* (Manning, 2021).

In many cases, the source code is written in a way that doesn't make it easy to read. Yes, I know what you are thinking: It should be. And I agree with you. Today, we learn many patterns and principles for code design and avoiding code smells, but let's be honest: developers still don't use these principles properly in too many cases. Moreover, legacy apps usually don't follow these principles, simply because the principles didn't exist many years ago when those capabilities were written. You still need to be able to investigate such code.

Take a look at listing 1.1. Suppose you found this piece of code while trying to identify the root cause of a problem in the app you're working on. This code definitely needs refactoring. But before you can refactor it, you need to understand what's doing. I know there are developers out there who can just read through this code and immediately understand what it does, but I'm not one of them.

To easily understand the logic in listing 1.1, I use a debugger to go through each line to observe how it works with the given input (as we'll discuss in chapter 2). (A debugger is a

tool that allows you to stop the execution on specific lines and manually execute each instruction while observing how the data is changed.) With a bit of experience and some tricks we'll discuss in chapters 2 and 3, you can find out in a couple of times of parsing this code that it calculates the maximum between the given inputs. In chapters 2 and 3, we'll take some of these examples where using a debugger tool help you understand the code faster and discuss this technique more.

You find the code presented in listing 1.1 as part of the project da-ch1-ex1 provided with the book.

Listing 1.1 A hard-to-read piece of logic for which you'd use a debugger tool to understand it

```
public int m(int f, int g) {
    try {
        int[] far = new int[f];
        far[g] = 1;
        return f;
    } catch(NegativeArraySizeException e) {
        f = -f;
        g = -g;
        return (-m(f, g) == -f) ? -g : -f;
    } catch(IndexOutOfBoundsException e) {
        return (m(g, 0) == 0) ? f : g;
    }
}
```

Some scenarios don't allow you to navigate through the code or make it more challenging to do so. Today most apps rely on dependencies – libraries or frameworks. In most cases, even where you have access to the source code (you use an open-source dependency), it's still difficult to follow the source code that defines a framework logic. Sometimes, you don't even know where to start. In such cases, you must use different techniques to understand the app. For example, you could use a profiler tool (as you'll learn in chapters 6 through 9) to identify what code executes before deciding where to start the investigation.

Other scenarios will not give you the chance to have a running app. In some cases, you'll have to investigate a problem that made the app crash. If the application that encountered problems and stopped is a production service, you need to make it available again fast. So you need to collect the details that help you debug the problem and use these details to identify the problem and improve your app to avoid the problem in the future. This investigation, which relies on collected data after the app crashed, is called "postmortem investigation." For such cases, you'd use logs, heap dumps, or thread dumps. All these are troubleshooting instruments that we'll discuss in chapters 10 and 11.

1.2 Typical scenarios for using investigation techniques

Let's discuss in this section some common scenarios for using code investigation approaches. We must take some examples of typical cases from real-world apps and analyze them to emphasize the importance of what we're discussing in the book. We'll consider the following common situations a developer faces when they need to investigate how an app works:

- Understand why a particular piece of code or software capability provides a different result than the expected one.
- Learn how technologies that the app uses as dependencies work.
- Identify causes for performance issues such as app slowness.
- Finding out root causes for cases in which an app suddenly stops.

For each presented case, you'll find one or more techniques helpful in investigating the app's logic. In the rest of the book, we'll dive into these techniques, and we'll demonstrate with examples how to use them.

1.2.1 Demystifying the unexpected output

The most frequent scenario where you'll need to analyze code is when some logic ends up with a different output than you expected. This case might sound simple, but scenarios in which you need to investigate the root cause of a different output than the one expected could be more or less easy to solve.

First, let's define "output". This term might have many definitions for an app. "Output" could be some text in the app's console, or it could be some records changed in a database. We can consider "output" to be an HTTP request the app sends to a different system, or it might be some data sent in the HTTP response to a client's request.

OUTPUT Any result of executing a piece of logic that might result in data change, exchange of information, or action against a different component or system.

How do we investigate a case in which a specific part of the app doesn't have an expected execution result? Choosing the proper technique depends on what the expected output is. Let's take some examples and analyze them.

SCENARIO 1 - THE SIMPLE CASE

Suppose the app should insert some records into a database. You observe that the app adds only a part of the records. You expected to find more data in the database than the app actually produces.

The simplest way would be to use a debugger tool to follow the code execution and understand how it works (figure 1.4). You'll learn all about the main features of a debugger in chapters 2 and 3. The debugger pauses the app execution at a specific instruction you choose, and then it allows you to continue the execution manually. You run each code instruction one by one while you can see the values variables hold and evaluate expressions on the fly.

You can mark an instruction with a breakpoint to tell the debugger to pause the execution before executing that specific instruction.

```

10 public int m(int f, int g) {    f: 10    g: 5
11     try {
12         int[] far = new int[f];    f: 10
13         far[g] = 1;
14         return f;
15     } catch(NegativeArraySizeException e) {
16         f = -f;
17         g = -g;
18     } catch(IndexOutOfBoundsException e) {
19         return (-m(f, g) == -f) ? -g : -f;
20     } catch(NullPointerException e) {
21         return (m(g, g: 0) == 0) ? f : g;
22     }

```

The debugger shows the value in each variable and you can use this to understand how the app execution changes the data.

Figure 1.4 Using a debugger, you can pause the execution before particular instruction and then observe how the app's logic changes the date by manually running the instructions step by step.

This scenario is the simplest, and by learning how to use all the relevant debugger features properly, you find solutions to such issues in no time. Unfortunately, other cases are more complex, and a debugger tool isn't enough to solve the puzzle and find the cause of the problem.



TIP In many cases, one investigation technique isn't enough to understand the app's behavior. You'll need to combine the use of various approaches to faster understand more complex behavior.

SCENARIO 2 – THE WHERE-SHOULD-I-START-DEBUGGING?

Some cases don't allow the use of a debugger directly simply because you don't know what to debug. Suppose your app is a complex service with a large number of lines of code. You investigate an issue where the app doesn't store the expected records in a database. It's definitely a problem of output, but out of the thousands of lines of code defining your app, you don't know what part of the app implements the capability whose behavior you investigate.

I remember the exclamation of a colleague of mine who was investigating such a problem. Stressed for not being able to find out where to start, he exclaimed: "I wish debuggers had a way in which you can add a breakpoint on all the lines of an app. So you could see what it actually uses."

My colleague's statement was funny, but having such a feature in a debugger wouldn't be a solution. We have different alternatives to approach this problem. You would most likely narrow the possibilities of lines for adding a breakpoint by priorly using a profiler for such a case.

A profiler is a tool that you can use to identify what code executes while the app is running (figure 1.5). This is an excellent option for our scenario because it would give you an idea of where to start the investigation with a debugger. We'll discuss using a profiler in chapters 6 through 9, where you'll learn that it gives you more options than just more easily observing the code in execution.

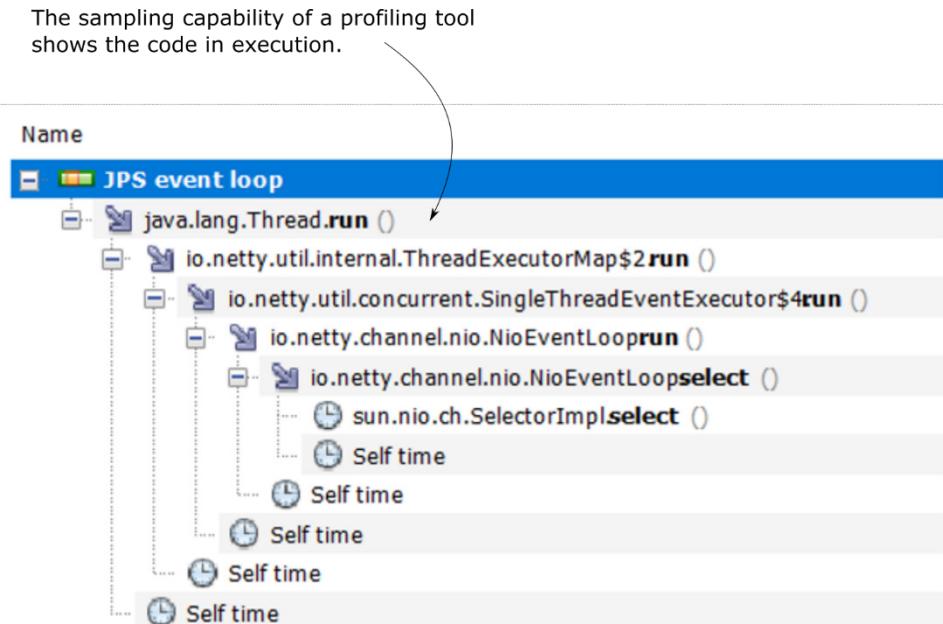


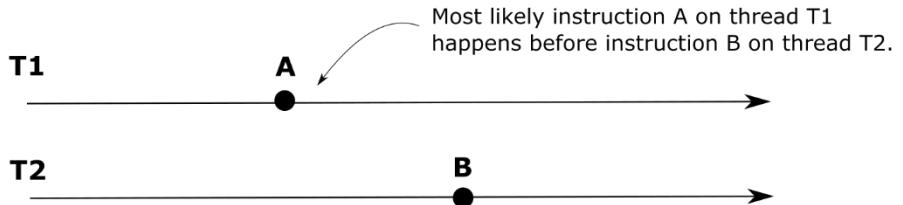
Figure 1.5 Identifying code in execution with a profiler. A profiler is a tool you can use to analyze the code that's executing. In case you don't know where to start debugging, the profiler can help you identify the code that is running and give you an idea of where you can use the debugger.

SCENARIO 3 – A MULTI-THREADED APP

Situations become even more complicated when dealing with logic implemented through multiple threads – a multi-threaded architecture. In most such cases, using a debugger is not an option at all because multi-threaded architectures tend to be sensitive to interference.

In other words, the way the app behaves is different when you use the debugger. Developers call this characteristic a "Heisenberg execution" or "Heisenbug", if we refer to an issue that doesn't happen or acts differently when the debugger (or another tool) interferes with the execution (figure 1.6). The name comes from the 20th-century physicist Werner Heisenberg who formulated the "uncertainty principle". This principle states that once you interfere with a particle, it behaves differently, so you cannot accurately predict both its velocity and position simultaneously (here's an article on this subject: <https://plato.stanford.edu/entries/qt-uncertainty/>). A multi-threaded architecture might change the way it behaves if you interfere with it just like you would interfere with a quantum mechanics particle, so this is why we compare these execution scenarios with Heisenberg's principle in quantum mechanics.

When nothing interferes with the app



When a debugger interferes with the app

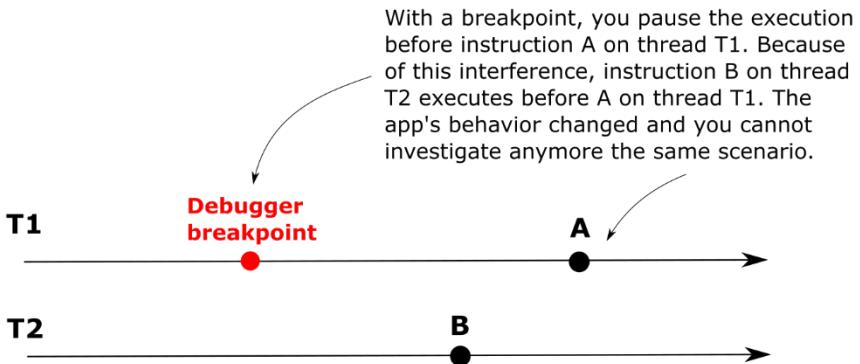


Figure 1.6 A Heisenberg execution. In a multi-threaded app, when a debugger interferes with the app's execution, it might change how the app behaves. This change doesn't allow you to correctly investigate the initial app behavior that you wanted to research.

For the area of multi-threaded functionality, we have a large variety of cases. That's what makes such scenarios, in my opinion, the most difficult to test. Sometimes, a profiler is a good option, but even the profiler might interfere with the app's execution in some cases, so it might not sometimes work either. Another alternative is to use logging (which we discuss in chapter 5) in the app. For certain issues, you can find a way to reduce the number of threads to one so that you can use a debugger for the investigation.

SCENARIO 4 – SENDING WRONG CALLS TO A GIVEN SERVICE

You could need to investigate a different scenario when the app doesn't interact correctly with another system component or an external system. Suppose your app sends HTTP requests to another app. You get notified by the maintainers of the second app that the HTTP requests don't have the right format (maybe a header is missing or the request body contains wrong data). Figure 1.7 visually presents this case.

You have to investigate why
the app sends an HTTP request
with incorrect data to another
system component.

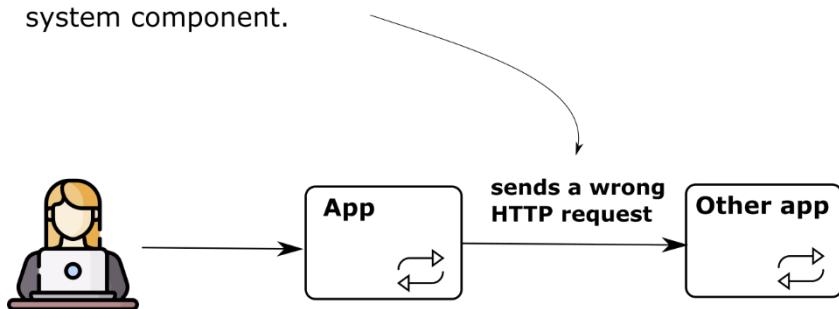
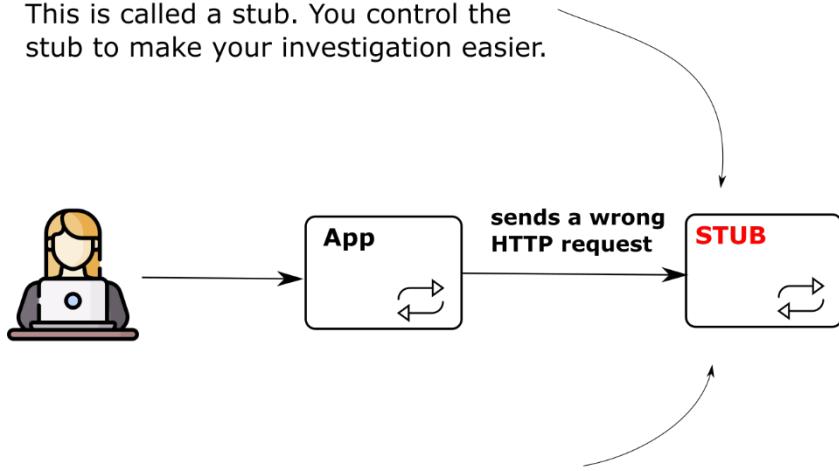


Figure 1.7 A wrong output can be your app sending erroneous requests to another system component. You could be asked to investigate such a wrong behavior and find its root cause.

This situation is still a "wrong output" scenario. How could you approach it? First, you need to identify what part of the code sends the requests. If you already know, you can further use a debugger to investigate how the app creates the request and identify what goes wrong. If you need to find out what part of the app sends a request, you might need to use a profiler, as you'll learn in chapters 6 through 9. You can use a profiler to determine what code acts at a given time in an executing process.

Here's a trick I always use when I have to deal with a complex case like this one where, for some reason, I can't identify straightforwardly where the app sends the request to/from. I replace the other app (the one my app wrongly sends requests to) with a stub. A stub is a fake application that I can control to help me identify the issue. For example, to determine what part of the code in my app sends the requests, I can make my stub block the request, so my app indefinitely waits for a response. Then, I simply use a profiler on my app to determine what code is being stuck by the stub. Figure 1.8 shows the usage of a stub. Compare this figure with figure 1.7 to understand how the stub replaced the real app.

You can create a fake app to replace the component your app calls.
This is called a stub. You control the stub to make your investigation easier.



For example, you can make the stub block indefinitely the HTTP request. In such a case, your app will remain blocked right on the instruction that sends the HTTP request. You can now easily use a profiler to identify that instruction.

Figure 1.8 You can replace the system component your app calls with a stub. You control the stub to allow to determine where your app sends the request from more quickly. Also, you can use the stub to test your solution after you correct the issue.

1.2.2 Learning certain technologies

Another use of techniques for analyzing code discussed in this section is for learning how certain technologies work. Some joke says that six hours of debugging can save five minutes of reading the documentation. While it's true that reading documentation is also essential when learning something new, some technologies are too complex to learn only from books and by reading the specifications. I always advise my students to "dive deeper" into a specific framework or library to understand it properly.



TIP For any technology (framework or library) you learn, spend some time reviewing the code you write. Always try to go deeper and debug on the framework's code.

I'll start with my favorite, Spring Security. At first glance, Spring Security might seem trivial. It's just implementing authentication and authorization, isn't it? Until you find out about the variety of ways to configure these two capabilities into your app. You mix them wrong – you might get in trouble. When things don't work, you have to deal yourself with what doesn't work. And the best choice to understand is debugging on Spring Security's code.

More than anything else, debugging helped me understand Spring Security. To help others, I put my experience and knowledge into a book, *Spring Security in Action* (Manning, 2020). If you read the book, you'll find I provide more than 70 projects. Not only for you to recreate them and run them, but also to debug them. I invite you to debug all examples provided with books you read to learn various technologies.

The second example of technology I mostly learned through debugging is Hibernate. Hibernate is a high-level framework used for implementing the app capabilities of working with a SQL database. Hibernate is one of the most known and used frameworks in the Java world, so it's a must-learn for any Java developer.

Learning Hibernate's basics is easy, and you can use books for this purpose. But in the real world, using Hibernate (the how and the where) is so much more than the basics. And for me, without digging deep into Hibernate's code, I definitely wouldn't have learned as much about this framework as I know today.

My advice for you is simple: For any technology (framework or library) you learn, spend some time reviewing the code you write. Always try to go deeper and debug on the framework's code. This will also make you a better developer.

1.2.3 Clarifying slowness

Performance issues occur now and then in apps, and like any other problem, you need to investigate it before knowing how to solve it. Learning the proper use of different debugging techniques to identify the causes of performance issues is vital.

In my experience, the most frequent performance issues occurring in apps are related to how fast-responding an app is. However, before going further, I'd like to mention that, even if most developers consider slowness and performance equal, that's not the case. Slowness problems (situations in which an app responds slowly at a given trigger) are just one kind of performance issue.

For example, I once had to debug a mobile app that was consuming the device's battery too quickly. I had an Android app using a library that connected using Bluetooth to an

external device. For some reason, the library was creating lots of threads without closing them. Usually, these threads that remain open and run without purpose are called "zombie threads". Zombie threads usually cause performance and memory issues. They are also usually challenging to investigate. In my case, I used profiling techniques which we'll discuss in chapters 6 through 9.

However, having the device's battery consuming fast because of an app is also an app's performance issue. An app using too much network bandwidth while transferring data over the network is another good example of a performance issue.

Let's stick to slowness problems, which is likely the most often encountered scenario. Many developers fear slowness problems. Usually, that's not because those problems are difficult to identify, but they can be challenging to solve in most cases. Finding the cause of a performance problem is usually an easy job with a profiler, as you'll learn in chapters 6 through 9. Besides showing you which code executes, as discussed earlier in section 1.2.1, a profiler also shows you the time the app spends for each instruction. Figure 1.9 shows how a profiler displays the execution time.

A profiler shows you the execution time for each instruction, which allows you to easily identify where a slowness problem comes from.

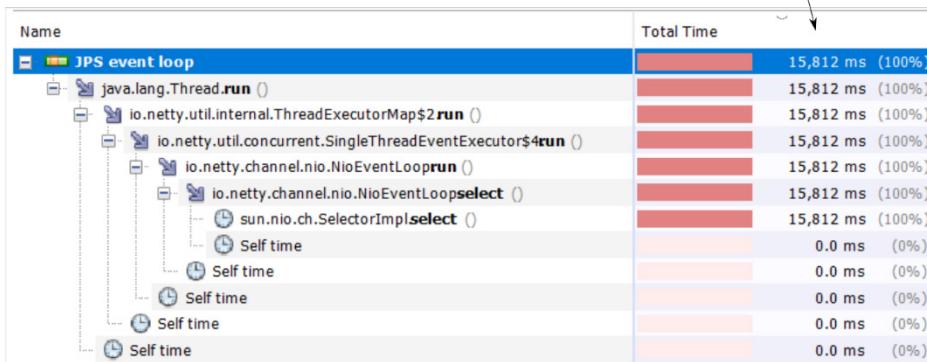


Figure 1.9 Investigating slowness problems with a profiler. The profiler shows you the time spent by each instruction during code execution. This profiler feature is excellent for identifying the root causes of performance problems.

In many cases, slowness problems are caused by I/O calls, such as reading or writing from a file or a database or sending data over the network. For this reason, developers often act empirically to find the problem cause. If you know what capability is affected, you try to focus on what I/O calls that capability executes. This approach also helps in minimizing the

scope of the problem to find the cause faster. But usually, you still need a tool to prove exactly where the problem comes from.

1.2.4 Understanding app crashes

Sometimes, apps completely stop responding due to various reasons. These kinds of problems are usually considered by developers more challenging to investigate than others. In many cases, the problems occur only in specific conditions, so you can't reproduce them in the local environment. Reproducing a problem is to make it happen on purpose in an environment where you can investigate it to find its root cause and solve it.

Every time you investigate a problem, you should first try reproducing it in an environment where you can study the problem. This approach gives you more flexibility with the investigation and also helps you confirm your solution after solving the problem. However, we're not always lucky to be able to reproduce a problem. App crashes are particularly a type of issue usually less often easy to reproduce.

We find app crashes scenarios in two main flavors:

1. The app completely stops
2. The app still runs but doesn't respond anymore to requests

When the app completely stops, it usually encountered an error from which it couldn't recover. Most often, a memory error causes such behavior. For a Java app, the situation where the heap memory fills and the app can't work anymore is represented by the `OutOfMemoryError`.

To investigate heap memory issues, we use heap dumps. A heap dump is a statistic of what the heap memory contains at a specific time. It is like a snapshot of the heap memory. You can configure a Java process to automatically generate such a snapshot when an `OutOfMemoryError` occurs, and the app crashes.

Heap dumps are powerful tools that give you plenty of details about how an app internally processes the data. We'll discuss all about how to use them in chapter 11. But let's see a short example now before we get diving deep into heap dumps so you have a better picture of what I'm talking about.

Listing 1.2 shows you a small code snippet that fills the memory with instances of a class named `Product`. You find this app in project `da-ch1-ex2` provided with the book. The app continuously adds `Product` instances to a list causing this way an intended `OutOfMemoryError`.

Listing 1.2 An app example causing an OutOfMemoryError

```
public class Main {  
  
    private static List<Product> products =      #A  
        new ArrayList<>();  
  
    public static void main(String[] args) {  
        while (true) {  
            products.add(      #B  
                new Product(UUID.randomUUID().toString()));      #C  
        }  
    }  
}
```

#A We declare a list that stores references of Product objects.

#B We continuously add Product instances to the list until the HEAP memory completely fills.

#C Each Product instance has a String attribute. We use a unique random identifier as its value.

Figure 1.10 shows a heap dump created for one execution of this app. Observe that you can easily find out that `Product` and `String` instances fill most of the HEAP memory. A heap dump is like a map of the memory. It gives you many details, including the relationships between instances as well as values. For example, even if you don't see the code, you can still observe a connection between the `Product` and the `String` instances based on how close the numbers of these instances are. Don't worry if these aspects look complex for now. We'll discuss in detail everything you need to know about using heap dumps in chapter 11. In this section, I just try to present what and why you'll learn further in the book.

Most of the memory is filled with String and Product objects.

The number of String instances is close to the number of Product instances, so a relationship is possible between them.

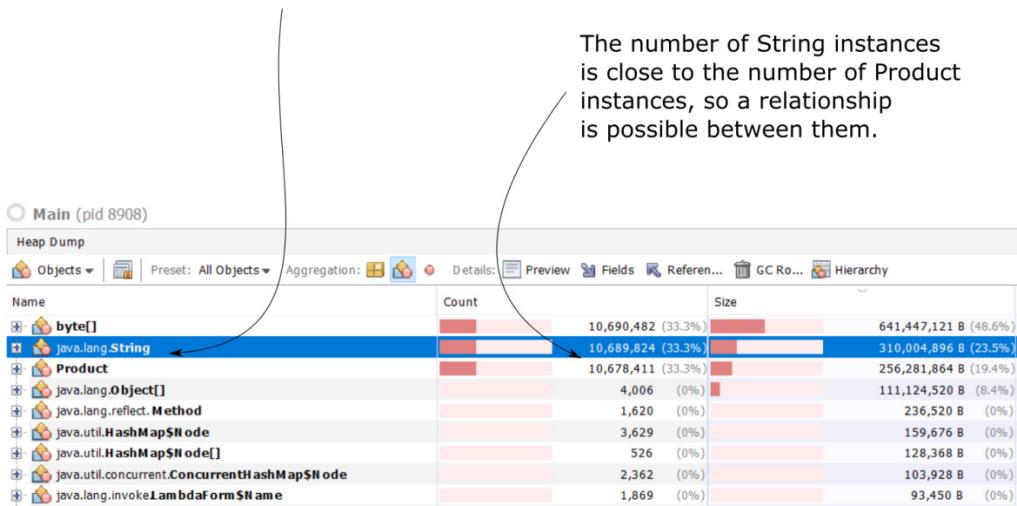


Figure 1.10 A heap dump is like a heap memory map. If you learn how to read it, it gives you invaluable clues of how the app internally processes data. A heap dump helps you investigate memory problems or performance issues. In this example, you see how easy you find out which object fills most of the app's memory and the fact that Product and String instances are related.

If the app still runs but doesn't respond anymore to requests, then a thread dump is the best tool to analyze what happens. A thread dump shows you the state of each app thread and what it was doing when the dump was taken. Figure 1.11 shows you an example of thread dump and some of the details you can easily observe with this tool.



Figure 1.11 A thread dump shows you details about the threads that were running when the dump was taken. In a thread dump, you find the thread states and the stack traces telling you what they were executing or what blocked them. These details are really valuable in investigating why an app is stuck or performance problems.

In chapter 10, we discuss generating and analyzing thread dumps to use them in debugging.

1.3 What you will learn in this book

This book is for Java developers with various levels of experience, from beginners to experts. In this book, you'll learn various code investigation techniques, scenarios in which to apply them, and how to apply them to bring you benefit from saving troubleshooting and investigation time.

If you are a junior developer, you'll most likely find many things to learn from this book. Some developers master all these techniques only after years of experience, others never master them. If you are already an expert, then you might find many things you already

know, but you still have a good chance to find new and exciting approaches you might not have had the opportunity to encounter yet.

When you finish the book, you'll have learned the following skills:

- Apply different approaches to using a debugger to understand an app's logic or find an issue.
- Investigate hidden functionality with a profiler to better understand how your app or a specific dependency of your app works.
- Use analyzing code techniques to determine whether your app or one of its dependencies causes a certain problem.
- Investigate data in an app's memory snapshot to identify potential problems with how the app processes data.
- Use logging to identify problems in an app's behavior or security breaches.
- Use remote debugging to identify problems that you can't reproduce in a different environment than the one the app's running in.
- Correctly choose what app investigation technique to use for specific cases to make your investigation faster.

1.4 Summary

- You can use various investigation techniques to analyze software behavior.
- Depending on your investigation scenario, one investigation technique works better than another. You need to know how to choose the correct approach to make your investigation faster.
- For some scenarios, using a combination of techniques helps you identify a problem faster. Learning how each analyzing technique works gives you an excellent advantage in dealing with complex scenarios.
- In many cases, developers use investigation techniques for learning new things rather than investigating issues. When learning complex frameworks such as Spring Security or Hibernate, just reading books or the documentation isn't enough. An excellent way to accelerate your learning is to debug examples that use a technology you want to understand better.
- A situation is easier to investigate if you can reproduce it in an environment where you can easily study it. Reproducing a problem not only helps you find its root cause easier, but it also helps you confirm that a solution works and solves the problem after applying it. You should always try first to reproduce the issue in an environment where you can investigate it.

2

Understanding your app's logic through debugging techniques

This chapter covers

- Using a debugger to investigate code
- When choosing to use a debugger and when avoiding it

Not long ago, during one of my piano lessons, I shared a music sheet of one of the songs I wanted to learn with my piano teacher. I was so impressed when I saw him just play the song while reading the music sheet for the first time. "How cool is that?" I thought. "How could someone get this skill?"

Then, I remembered some years ago, I was in a peer-programming session with one of the newly hired juniors in the company I was working for. It was my turn at the keyboard, and we were investigating a relatively large and complex piece of code using the debugger. I started navigating through the code using the debugger, pressing relatively fast on the keyboard keys that would allow me to step over, into, and out of specific lines of code. While I was focusing on the code I was investigating, but quite calm and relaxed, almost forgetting I had someone near me (rude of me), I heard her saying, "Wow, stop a bit. You're too fast. Can you even read that code?"

I realized that situation was similar to the one in the story with my piano teacher. How can you get to have this skill? The answer is easier than you thought: Work hard and gain experience. But, while practicing is invaluable and takes a lot of time to learn how to play the piano, for debugging, I have some tips to share with you that will help you improve your technique much faster.

In this chapter, we discuss one of the most important tools used in understanding code: **the debugger**.



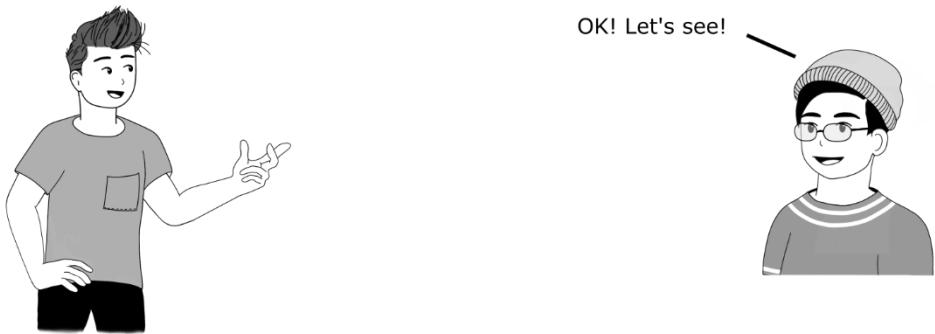
A **debugger** is a tool that allows you to pause the execution on specific lines and manually execute each instruction while observing how the data is changed.

A debugger is like Google Maps navigation - helping you find your way through complex logic implemented in your code. It's most likely also the most used tool for understanding code.

The debugger is usually the first tool a developer learns to use to help them understand what code does. Fortunately, all IDEs come with a debugger, so you don't have to do anything special to have one. In this book, I use IntelliJ IDEA Community in my examples, but any other IDE is quite similar, offering (sometimes with a different look) the same options we'll discuss. Although a debugger seems to be a tool most developers know how to use, you might find out in this chapter and in chapter 3 some techniques of using a debugger you might not know yet.

We'll start in section 2.1 by discussing how developers read code and why in many cases, just reading the code isn't enough to understand it easily. Because only reading the code is often not enough, we need to use various tools in our investigation, such as a debugger or a profiler (which we discuss later, in chapters 6 through 9). In section 2.2, we continue the discussion by applying the most simple techniques for using a debugger with an example.

If you are an experienced developer, you might already know these techniques. But, you may still find it useful reading through the chapter as a refresher, or you could skip and directly read the more advanced techniques for using a debugger that we'll discuss in chapter 3.



2.1 When analyzing code is not enough...

Let's start by discussing how to read code and why sometimes only reading the logic isn't enough to understand it. In this section, I'll tell you how reading code works and how it is different from reading something else like a story or poetry. To observe this difference and understand what causes the complexity in reading code, we'll use a code snippet that implements a short piece of logic. Understanding what's behind the way our brain interprets code helps you realize the need for tools such as a debugger.

Any code investigation scene starts with reading the code. But reading code is different than reading poetry. When reading a verse, you read the text line by line in a given linear order letting your brain assemble and picture the meaning. You read the same verse twice; you might understand different things.

With code, however, that's opposite. First of all, code is not linear. When reading code, you don't simply read it line by line. Instead, you jump in and out of instructions while understanding how they affect the data they process. Reading code is more like a maze rather than a straight road. If you're not attentive, you might get lost and forget where you started from. Secondly, unlike a poem, the code always and for everyone "means" the same thing. That meaning is your investigation's objective.

Same as you'd use a compass to find your path, a debugger helps you easier identify what your code does. We'll use as an example the `decode(List<Integer> input)` method presented in listing 2.1. You also find this code in project da-ch2-ex1 provided with the book.

Listing 2.1 An example of a method to debug

```
public class Decoder {
    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }
        return total;
    }
}
```

Going from the top line to the bottom line in a code snippet to understand it, you find yourself in situations where you have to assume how something works (figure 2.1). Are those instructions really doing what you think they're doing? When you are not sure, you have to dive into these instructions and observe what they actually do – analyze the logic behind them. Figure 2.1 point out two of the uncertainties in the given code snippet:

- What does the `StringDigitExtractor()` constructor do? It might only create an object, or it might also do something else. It could be that it somehow changes the value of the given parameter.
- What is the result of calling the `extractDigits()` method? Does it return a list of digits? Does it also change inside the object the parameter we used when creating the `StringDigitsExtractor` constructor?

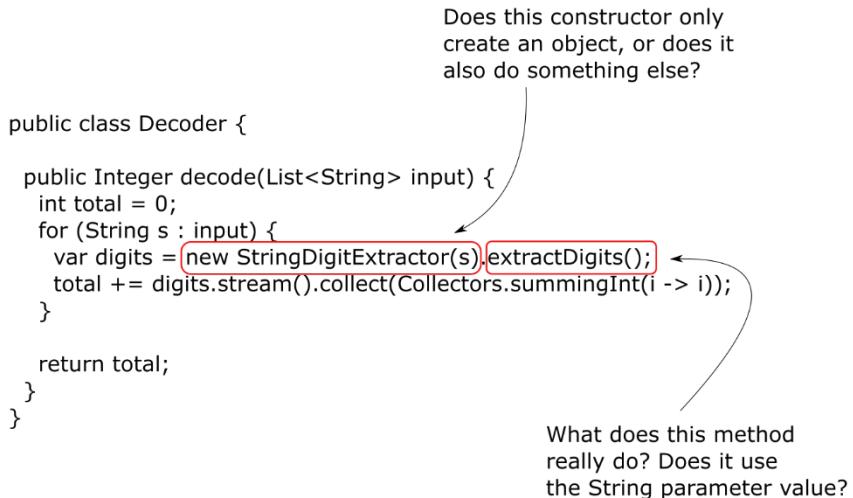


Figure 2.1 When reading a piece of code, you often need to figure out what happens behind the scenes in some of the instructions composing that logic. The method names are not always suggestive enough, and you can't totally rely on them. Instead, you need to go deeper into what these methods do and analyze them first to understand what they do.

So even with a small piece of code, you might have to dive more deeply into some instructions. Each new code instruction you investigate further creates a new investigation plan and adds to the cognitive complexity of the investigation (figures 2.2 and 2.3). As you go deeper into the logic and open more plans, the more complex the process becomes.

You take a piece of stone,
chisel it with blood,
grind it with Homer's eye,
burnish it with beams
until the cube comes out perfect.

Next you endlessly kiss the cube
with your mouth, with others' mouths,
and, most important,
with infanta's mouth.

Then you take a hammer
and suddenly knock a corner off.

All, indeed absolutely all will say
what a perfect cube
this would have been
if not for the broken corner.

Reading poetry is linear.
You read each verse
one by one, from top
to bottom.



↓ (Lesson about the cube, N. Stănescu)

Figure 2.2 Compare how you read poetry with how you read code. You read poetry line by line, but when you read code, you jump around, as shown in figure 2.3.

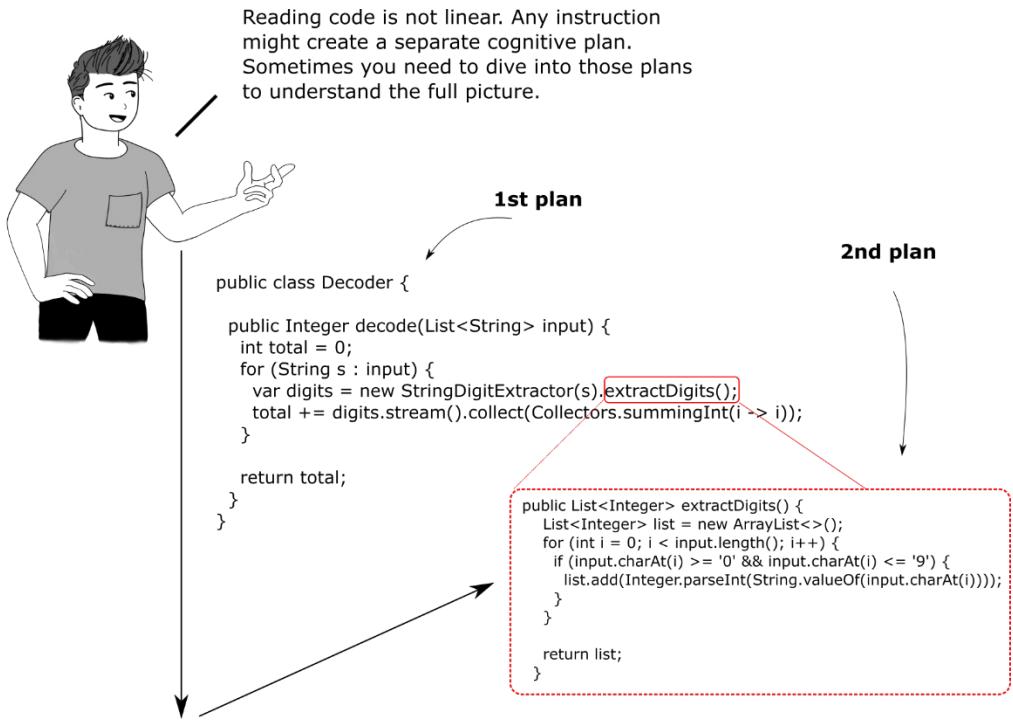


Figure 2.3 Reading code is different than reading poetry and adds a lot of complexity. You can imagine reading code as reading in two dimensions. One dimension is reading a current piece of code up to down. The second dimension is going into a specific instruction to understand it in detail. Remembering how things work for each plan and how they assemble makes understanding code solely by reading it very difficult.

Reading poetry always has one path. Instead, code analysis creates many paths through the same piece of logic. The fewer new plans you open, the less complex the process is. The compromise is, skipping over a certain instruction, making the overall investigation process simpler, or going into detail to better understand each individual instruction and raise the process complexity.



TIP Always try to shorten the reading path by minimizing the number of plans you open for investigation. Use a debugger to help you navigate the code easier, keep track of where you are, and observe how the app changes the data while executing.

2.2 Investigating code with a debugger

In this section, we discuss a tool that helps us minimize the cognitive effort of reading code to understand how it works – a debugger. All IDEs provide a debugger, and even if the interface might look slightly different from one IDE to another, the options you have are generally the same. For the examples in this book, I'll use IntelliJ IDEA Community, but I encourage you to use your favorite IDE and compare it with the examples in the book. You'll find they are pretty similar.

A debugger simplifies the investigation process by:

- Providing you with a means to pause the execution at a particular step and execute each instruction manually at your own pace.
- Showing you where you are and where you came from in the code reading path. This way, the debugger works as a map releasing you from having to remember all the details.
- Showing you the values that variables hold, making the investigation more visual and easier to process.

Let's take the example in project da-ch2-ex1 again and use the most straightforward debugger capabilities to understand the code. Listing 2.2 repeats the code we discussed in section 2.1 so that you don't have to flip back pages.

Listing 2.2 A piece of code we want to understand

```
public class Decoder {
    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }
        return total;
    }
}
```

I'm sure you now wonder: "How do I know when to use a debugger?". This is a fair question I want to answer before teaching you how to use the debugger. The main prerequisite for

using a debugger is knowing what piece of logic you want to investigate. As you'll learn further in this section, the first step for using a debugger is selecting an instruction where you want the execution to pause.



NOTE Unless you already know which instruction you need to start investigating the execution from, you can't use a debugger.

You'll find cases where you don't know up-front a specific piece of logic you want to investigate in real-world scenarios. In this case, before using a debugger, you'll need to apply different techniques to find out which is the part of the code you want to investigate using the debugger. But don't worry, in the rest of the book, we'll also discuss how to find out which is the part of code you need to investigate in such cases where you don't know it upfront. In this chapter, and chapter 3, we'll focus only on using the debugger, so we'll assume somehow we found the piece of code we want to understand.

Going back to our example, where do we start from? First of all, we need to read the code and figure out what we understand and don't understand. Once we figure out where the logic becomes unclear, we will be able to execute the app and "tell" the debugger to pause the execution. We pause the execution on those lines of code that are not clear to observe how they change the data. To "tell" the debugger where to pause the app's execution, we use breakpoints.



A **breakpoint** is a marker we use on lines where we want the debugger to pause the execution so we can investigate the implemented logic.

In figure 2.4, I shaded differently the code that we usually understand straightforwardly (considering you know the language fundamentals). It's usually pretty fast to observe that this code takes a list as an input, parses the list and processes each item in it, and somehow calculates an integer that the method returns in the end. Moreover, the process the method implements is easy to observe without the need for a debugger.

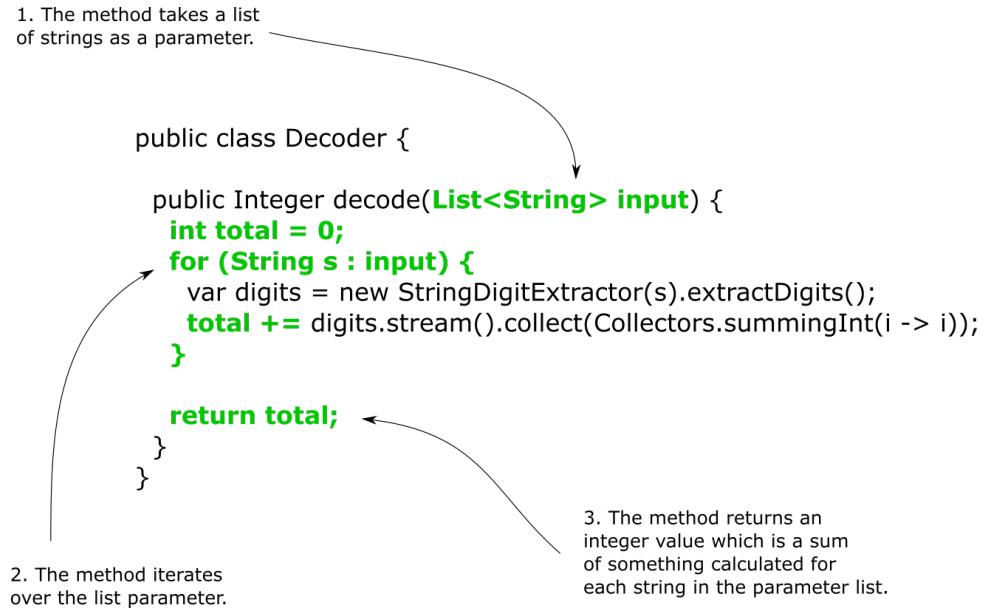


Figure 2.4 Assuming you know the language fundamentals, you can easily observe that this code takes a collection as input and parses the collection to calculate an integer.

In figure 2.5, I shaded differently the lines that usually cause difficulties in understanding what the method does. These lines of code are more challenging to understand because they hide their own implemented logic. Some will find the `digits.stream().collect(Collectors.summingInt(i -> i))` understandable since it's part of the Stream API provided with the JDK since Java 8. So depending on how deep your Java knowledge is, this line might be straightforward to understand as well. But for the new `StringDigitExtractor(s).extractDigits()` we can't say the same thing. Since this is part of the app we investigate, this instruction might do anything.

Additional complexity might be added by the way the developer chooses to write the code. For example, starting with Java 10, developers can infer the type of a local variable using `var`. Inferring the variable type is not always a wise choice because it might make the code even more difficult to read (figure 2.5), bringing one more scenario in which using the debugger would be useful.



TIP When investigating code with a debugger, start directly from the first line of code that you can't figure out straightforwardly what it does.

Training junior developers and students in the past many years, I observed that they start debugging on the first line of a specific code block in many cases. While you certainly can do so, it's more efficient if you first read the code without the debugger at all and try to figure out whether you can understand the code. Then, start debugging directly from the point that causes more difficulties. This approach will save you time since you might find out you don't need the debugger to understand what happens in a specific piece of logic. After all, even if you need to use the debugger, you go only over the code you need to understand.

There can be cases where you add a breakpoint on the only line you know it certainly executes. Sometimes, for example, your app throws an exception, you see that in the logs and you don't know which is the exact previous line that causes the problem. So what you can do is add a breakpoint to pause the app's execution just before it throws the exception. But the idea stays the same – avoid pausing the execution of the instructions you understand – instead, use breakpoints for the lines of code you want to focus on.

What happens for every string in the list? How is the String turned into a number?

```
public class Decoder {  
  
    public Integer decode(List<String> input) {  
        int total = 0;  
        for (String s : input) {  
            var digits = new StringDigitExtractor(s).extractDigits();  
            total += digits.stream().collect(Collectors.summingInt(i -> i));  
        }  
  
        return total;  
    }  
}
```

Figure 2.5 In this piece of code, we shaded differently the lines of code that we most likely find more difficult to understand. When starting to use the debugger, you add the first breakpoint on the first line that makes the code more challenging to understand.

For this example, we start by adding a breakpoint on line 11 presented in figure 2.6:

```
var digits = new StringDigitExtractor(s).extractDigits();
```

Generally, to add a breakpoint on a line in any IDE, you click on or near the line number (or even better, use a keyboard shortcut – for IntelliJ you can use Ctrl-F8 for Windows/Linux, or Command-F8 for a MacOS). The breakpoint will be displayed with a circle, as presented in figure 2.6. Make sure you run your application with the debugger. In IntelliJ, you find a button represented as a small bug icon just near the one you mainly use to start the app. You can also right-click on the main class file and use the “Debug” button in the context menu. When the execution reaches the line you marked with a breakpoint, it pauses, allowing you to navigate further.

1. You add a breakpoint on the line where you wish the debugger to stop the execution. This line should usually be the first instruction that creates concerns.

2. You run the app with the debugger.

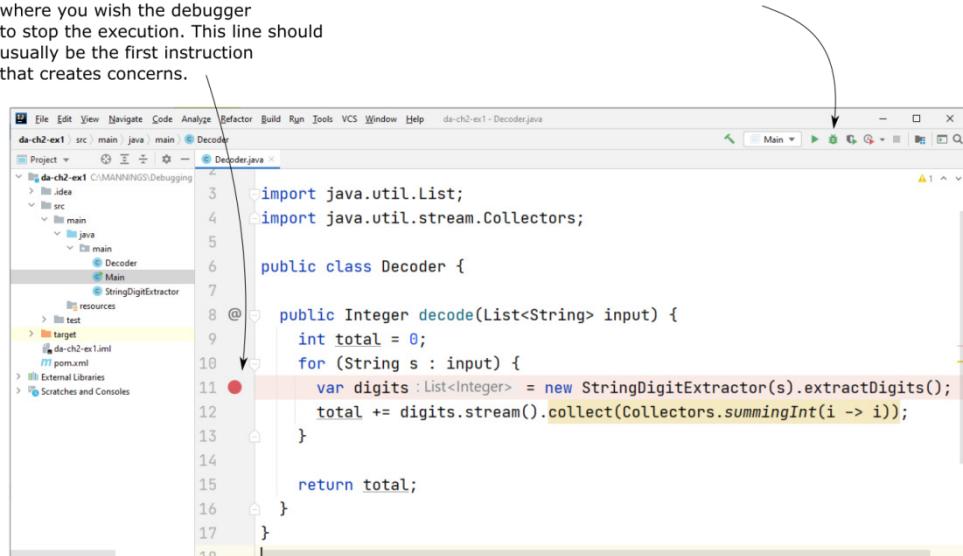


Figure 2.6 Click near the line number to add a breakpoint on a specific line. Then, run the app with the debugger. The execution pauses on the line you marked with a breakpoint and allows your control further.

NOTE Remember, you always need to execute the app using the “Debug” option to have an active debugger. If you use the “Run” option, the breakpoints won’t be considered since the IDE doesn’t attach the debugger to the running process. Some IDEs may run by default your app and also attach the debugger, but if that’s not the case (like for IntelliJ or Eclipse), then the app execution won’t pause at the breakpoints you define.

When the debugger paused the code execution on a specific instruction from the line you marked with a breakpoint, you can already use valuable information the IDE displays. In figure 2.7, you can see how my IDE displays two essential pieces of information:

1. *The value of all the variables in scope* – knowing all the variables in scope and their values help you understand what data is processed and how the logic you investigate processes the data. Remember that the execution is paused before the execution of the line marked with a breakpoint. So data state is also as before the execution of the instruction marked with the breakpoint.
2. *The execution stack trace* – shows you how the app got to execute the line of code where the debugger paused the execution. Each line in the stack trace is a method involved in the calling chain. The execution stack trace helps you visualize the execution path and helps you to avoid needing to remember how you got at a specific instruction when using the debugger to navigate through code.



TIP You can add as many breakpoints as you want, but the best is to only use a limited number at a time and focus only on those lines of code you marked with the breakpoints. I usually use no more than three breakpoints at the same time. I often see developers adding too many breakpoints, forgetting them, and getting lost in the investigated code.

The execution paused on the line you marked with a breakpoint.

```

public class Decoder {
    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }
        return total;
    }
}

```

The debugger also shows you the stack trace. The stack trace shows you the execution path so you can easily see who called the method you investigate.

When the debugger pauses the app execution on a specific line, you can see the values of all the variables in the scope.

Figure 2.7 When the execution is paused on a given line of code, you can see all the variables in scope and their values. You can also use the execution stack trace to remember where you are navigating through the lines of code.

Generally, observing the values of the variables in scope is something easy to get. But depending on your experience, you might be aware of what the execution stack trace is. So, let's continue in section 2.2.1 with what the execution stack trace is and why this tool is essential. We'll then discuss in section 2.2.2 how to navigate the code using the essential operations such as the step-over, step-into, and step-out. You can skip section 2.2.1 and go directly to 2.2.2 if you are already familiar with the execution stack trace.

2.2.1 What is the execution stack trace, and how to use it?

The execution stack trace is a valuable tool you use to understand the code while debugging. Same as a map, the execution stack trace shows you how the execution got to the specific line of code where the debugger paused it and helps you decide where to navigate further.

Figure 2.8 shows a comparison between the execution stack trace and the execution seen in a tree format. The stack trace shows you how methods called one another up to the point where the debugger paused the execution. In the stack trace, you find the method names, the class names, and the lines that caused the calls.

We read the execution stack from bottom to top.
 The bottom layer in the stack is the first layer.
 The first layer is the one where the execution began.
 The top layer in the stack (the last layer) is the method where the execution is currently paused.

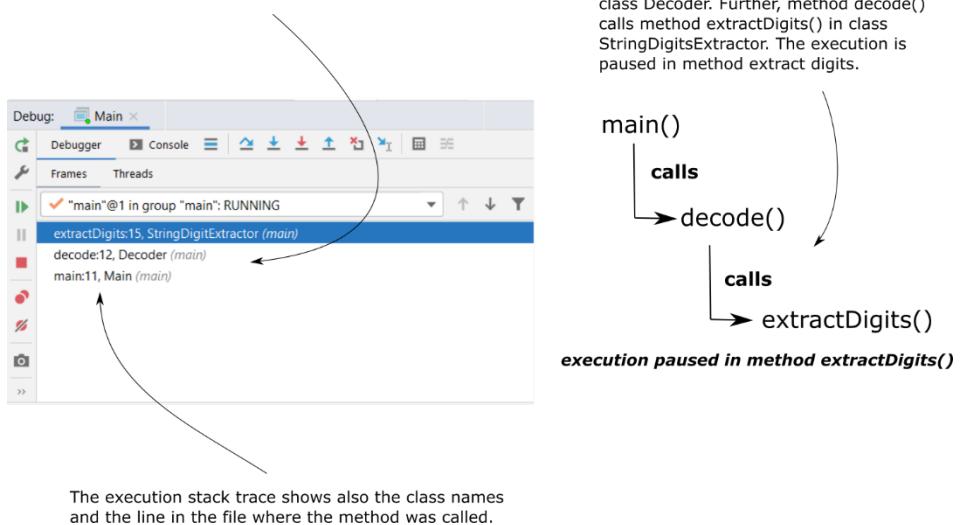


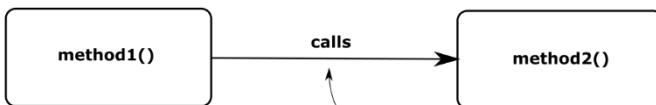
Figure 2.8 The top layer of the execution stack trace is the method where the debugger paused the execution. Each other layer in the execution stack trace is where the method represented by the above layer has been called. The bottom layer of the stack trace (also named the „first“ layer) is where the execution of the current thread began.

One of my favorite usages of the execution stack trace is finding hidden logic in the execution path. In most cases, developers use the execution stack trace simply to understand where a certain method has been called from. But another essential thing you need to consider is that real-world apps which use frameworks (such as Spring, Hibernate, and so on) sometimes alter the execution chain of the method.

For example, Spring apps often use code that is decoupled in what we call “aspects” (in Java/Jakarta EE terminology, they are named “interceptors”). These aspects implement logic that the framework uses to augment the execution of specific methods in certain conditions. Unfortunately, such logic is often difficult to observe since you can’t see the aspect code

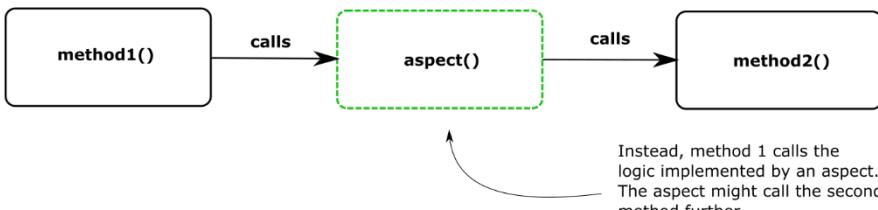
directly in the call chain when reading the code (figure 2.9). This characteristic of the aspects makes a developer's life challenging to investigate a given capability.

The apparent flow of method execution



Reading the code, it looks like method 1 directly calls method 2.

How the code really executes



Instead, method 1 calls the logic implemented by an aspect. The aspect might call the second method further.

Figure 2.9 An aspect logic is completely decoupled from the code. For this reason, when reading the code is difficult to realize there's more logic that will execute than you actually see. Such cases where you have hidden logic executing can be confusing when investigating a certain capability.

Let's take a code example to prove this behavior and how the execution stack trace is helpful in such cases. You find this example in project da-ch2-ex2 provided with the book (appendix B is a refresher for opening the project and starting the app). The project is a small Spring app that prints the value of the parameter in the console.

Listings 2.3, 2.4, and 2.5 show the implementation of these three classes. As presented in listing 2.3, the `main()` method calls `ProductController's saveProduct()` method sending the parameter value "Beer".

Listing 2.3 The apps' main class calls the `ProductController's saveProduct()` method

```

public class Main {
    public static void main(String[] args) {
        try (var c =
            new AnnotationConfigApplicationContext(ProjectConfig.class)) {
            c.getBean(ProductController.class).saveProduct("Beer"); #A
        }
    }
}
  
```

#A We call the `saveProduct()` method with the parameter value "Beer"

In listing 2.4, you observe that `ProductController's saveProduct()` method simply calls further the `ProductService's saveProduct()` method with the received parameter value.

Listing 2.4 The ProductController calls ProductService, sending the parameter value

```
@Component
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    public void saveProduct(String name) {
        productService.saveProduct(name);      #A
    }
}
```

#A ProductController calls the service method sending further the parameter value.

Listing 2.5 shows the `ProductService's saveProduct()` method that prints the parameter value in the console.

Listing 2.5 The ProductService prints in the console the value of the parameter

```
@Component
public class ProductService {

    public void saveProduct(String name) {
        System.out.println("Saving product " + name);      #A
    }
}
```

#A Printing the parameter value in the console

As presented in figure 2.10, the flow is quite simple:

1. The `main()` method calls `saveProduct()` method of a bean named `ProductController`, sending the value "Beer" as a parameter.
2. Then, the `ProductController saveProduct()` method calls the `saveProduct()` method of another bean `ProductService`.
3. The `ProductService` bean prints the value of the parameter in the console.

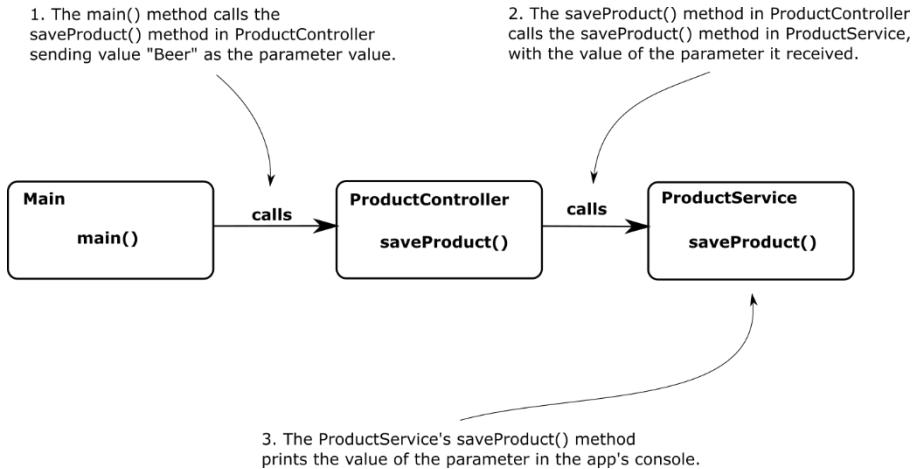


Figure 2.10 Method `main()` calls `saveProduct()` of bean `ProductController` sending the value “Beer” as the parameter value. The `ProductController’s saveProduct()` method calls the `ProductService` bean sending the same parameter value as the one it receives. The `ProductService` bean prints the parameter value in the console. The expectation is that “Beer” will be printed in the console.

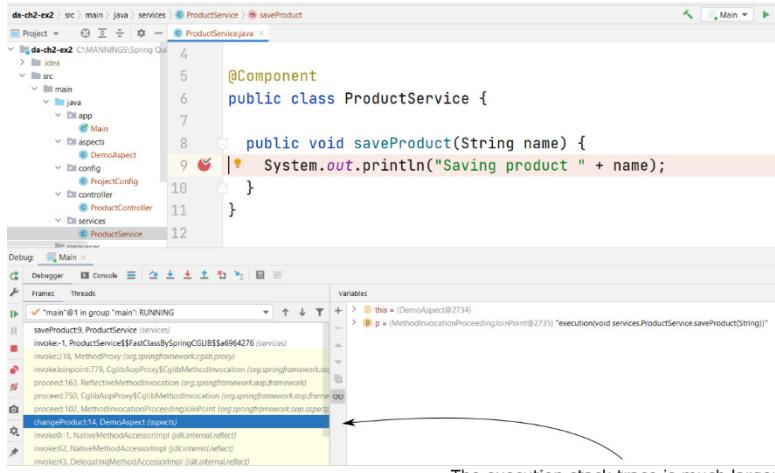
Naturally, you would assume the following message is printed when running the app:

```
Saving product Beer
```

However, when you run the project, you observe that the message is a different one:

```
Saving product Chocolate
```

How is that possible? To answer this question, the first thing is to use the execution stack trace to find out who changes the parameter value. Add a breakpoint on the line that prints a different value than you expect, run the app with the debugger, and observe the execution stack trace (figure 2.11). Instead of having the `ProductService’s saveProduct()` method from the `ProductController` bean, you find out that an aspect alters the execution. You check out the aspect class and observe that indeed the aspect is responsible for replacing the “Beer” with “Chocolate” (listing 2.6).



The execution stack trace is much larger than you can observe when reading the code. You can clearly see in the execution stack trace that `ProductService`'s `saveProduct()` method is not called directly from `ProductController`. Somehow, an aspect executes in between the two methods.

Figure 2.11 The execution stack trace shows that an aspect has altered the execution. This aspect is the reason why the value of the parameter changes. Without using the stack trace, finding why the app has a different behavior than expected would have been more difficult.

Listing 2.6 shows the code of the aspect that alters the execution, replacing the value `ProductController` sends to `ProductService`.

Listing 2.6 The aspect logic that alters the execution

```
@Aspect
@Component
public class DemoAspect {

    @Around("execution(* services.ProductService.saveProduct(..))")
    public void changeProduct(ProceedingJoinPoint p) throws Throwable {
        p.proceed(new Object[] {"Chocolate"});
    }
}
```

Aspects are quite a fascinating and useful feature in Java application frameworks today. But if you don't use them properly, they can lead to apps being difficult to understand and maintain. Of course, in this book, we discuss relevant techniques that will help you identify and understand the code even in such cases. But trust me, if you need to use this technique for an application, it means the application is not easily maintainable. A clean coded app (without technical debt) is always a better choice than an app for which you invest effort in debugging later. If you're interested in understanding better how aspects work in Spring, I recommend you to read chapter 6 of another book I wrote, *Spring Start Here* (Manning, 2021).

2.2.2 Navigating code with the debugger

In this section, we discuss the basic ways you navigate code with a debugger. You'll learn how to use three fundamental navigation operations:

1. **Step over** – you continue the execution with the next line of code in the same method.
2. **Step into** – you continue the execution inside one of the methods called on the current line.
3. **Step out** – you return the execution to the method that called the one you investigate now.

The investigation process starts with identifying the first line of code where you want the debugger to pause the execution. To understand the logic, you need to navigate through the lines of code and observe how the data changes when different instructions execute.

You have some buttons on the GUI and keyboard shortcuts to use the navigation operations in any IDE. Figure 2.12 shows you how these buttons appear in the IntelliJ IDEA Community GUI, the IDE I use.

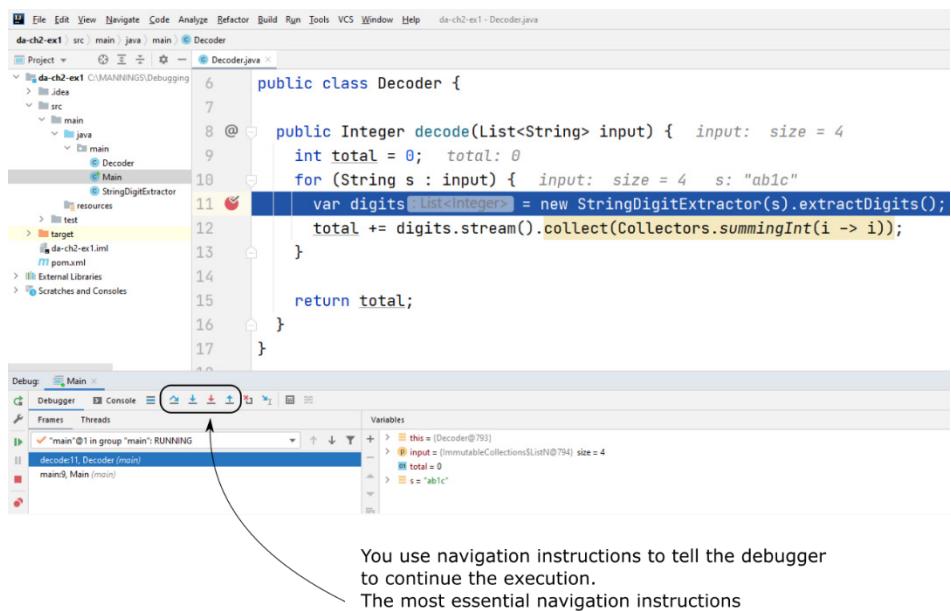


Figure 2.12 The navigation operations help you “walk” through the app logic in a controlled way to identify how the code works. To navigate through code, you can use the buttons on the IDE’s GUI (as presented in the figure) or use the keyboard shortcuts associated with these operations.



TIP Even if at the beginning you might find it easier to use the buttons on the IDE's GUI, I recommend you use the keyboard shortcuts instead. If you get comfortable using the keyboard shortcuts, you'll observe you can use them much faster than if you use a mouse.

Figure 2.13 visually describes the navigation operations. You can use the "step over" operation to go to the next line in the same method. Generally, this is the most commonly used navigation operation.

Now and then, you need to understand better what happens with a particular instruction. In our example, you could need to enter the `extractDigits()` method to understand clearly what it does. For such a case, you use the "step into" operation. When you want to return to the `decode()` method, you can use "step out".

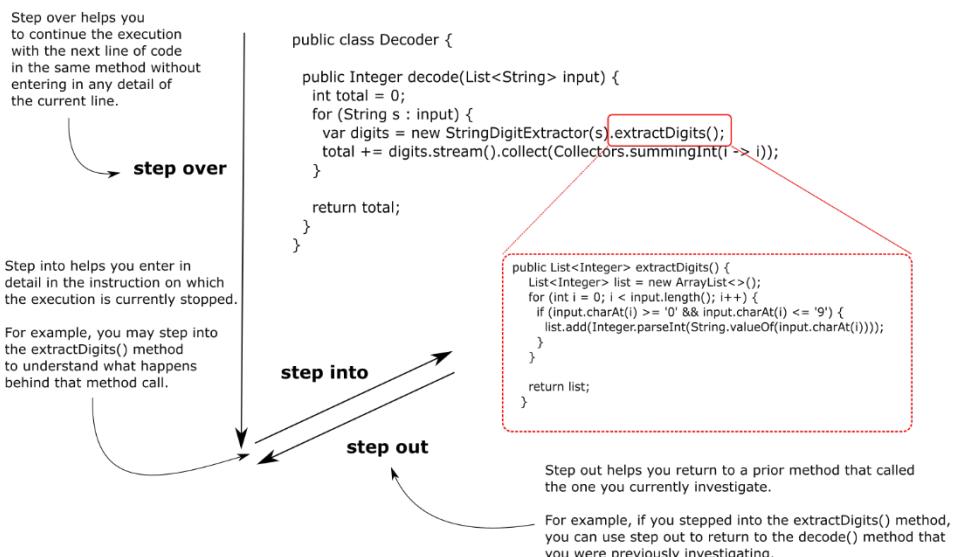


Figure 2.13 Navigation operations. Stepping over helps you go to the next instruction in the same method. When you want to start a new investigation plan and go into details in a specific instruction, you can use the step into operations. You can go back to the previous investigation plan with the step-out operation.

You can also visualize the operations on the execution stack trace as presented in figure 2.14.

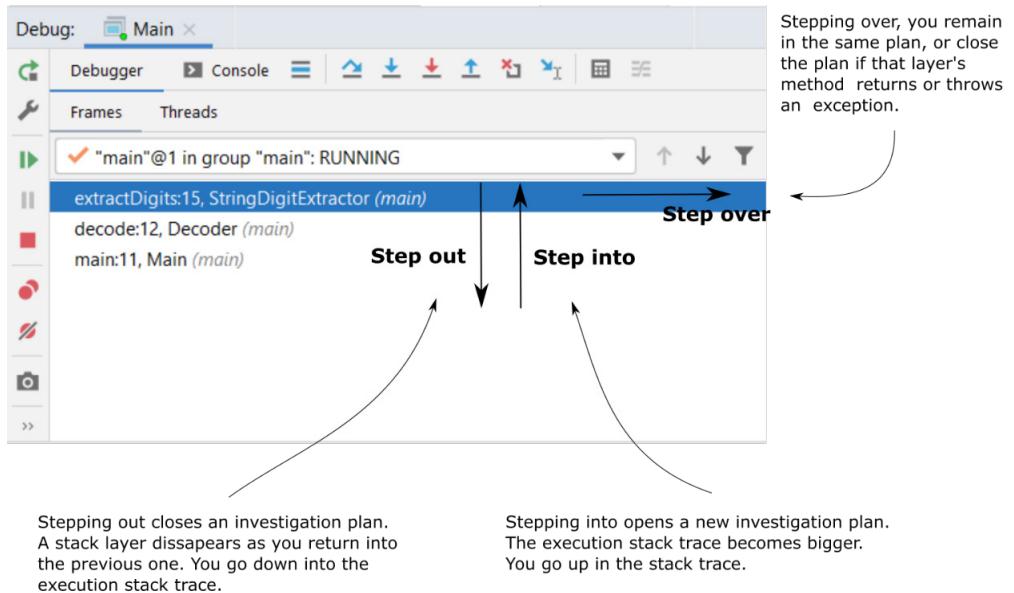


Figure 2.14 Navigation operation as seen from the execution stack trace point of view. When you step out, you go down in the stack trace and close an investigation plan. When you step into, you open a new investigation plan so you go up in the stack trace and the stack trace becomes bigger. Stepping over, you remain in the same investigation plan. If the method ends (returns or throws an exception) stepping over closes the investigation plan, and you go down in the stack trace same as when you step out.

Ideally, you start with using the “step over” operation as much as possible when trying to understand how a piece of code works. The more you step into, the more investigation plans you open, so the more complex the investigation process becomes (figure 2.15). In many cases, you can deduce what a specific line of code does only by stepping over it and observing the output.



Figure 2.15 The movie *Inception* (2010) portrays the idea of dreaming in a dream. The more layers deep you dream, the longer you stay there. You can compare this idea with stepping into a method and opening a new investigation layer. The deeper you step in, the more time you'll spend investigating the code.

Figure 2.16 shows you the result of using the step over navigation operation. The execution pauses now on line 12, one line below the one where we initially paused the debugger with the breakpoint. The `digits` variable is now initialized as well, so you can see its value.

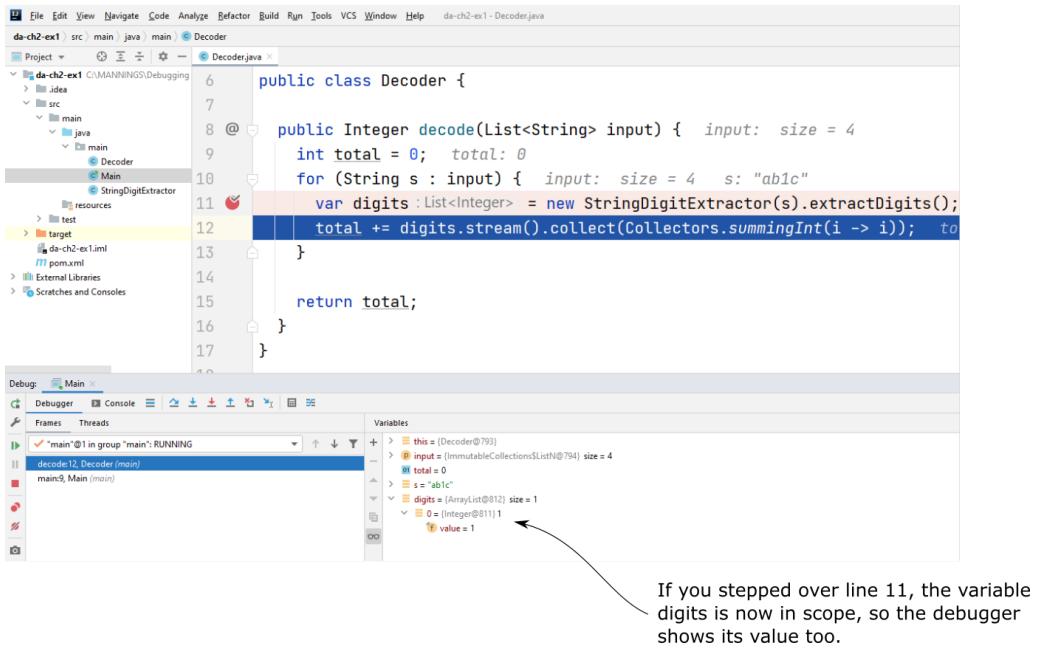


Figure 2.16 When you step over a line, the execution continues in the same method. In our case, the execution paused next on line 12, and you can see the value of the digits variable that was initialized by line 11. You can use this value to deduce what line 11 does without having to go into more detail about what this instruction does.

Try continuing the execution multiple times. You'll observe that, on line 11, for each string input, the result is a list that contains all the digits in the given string. Often, the logic is easy enough to understand just by analyzing the outputs for a few executions. But what if we don't figure out what a line does just by executing it?

If you don't figure out what happens, it means you need to go into more detail on that line. Going into more detail on a line should be your last option since you open a new investigation plan that complicates your process. But, when you have no other choice, you use this option to get more details on what the code does. Figure 2.17 shows you the result of stepping into line 11 of the `Decoder` class:

```
var digits = new StringDigitExtractor(s).extractDigits();
```

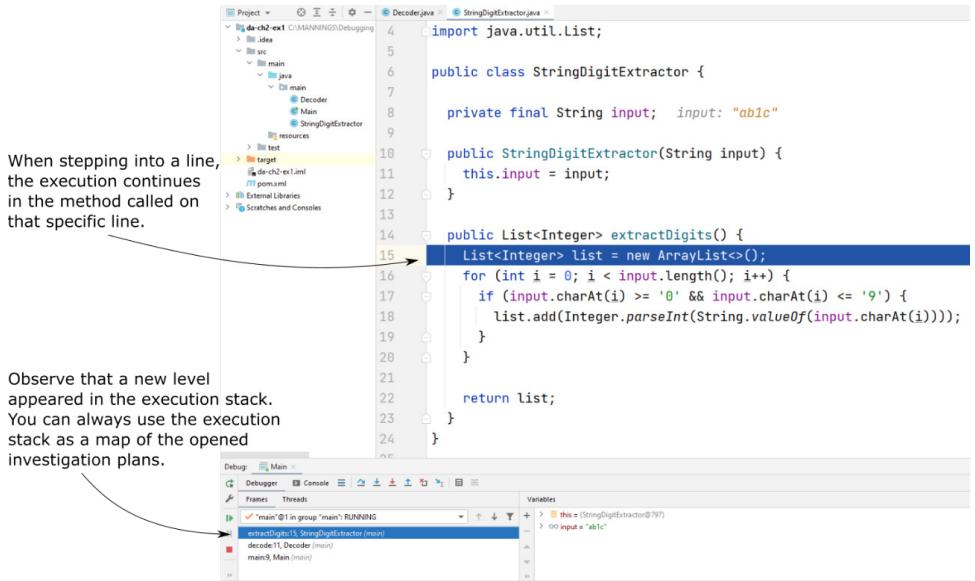


Figure 2.17 Stepping into allows you to observe the entire execution of the current instruction. This opens a new investigation plan, allowing you to parse the logic behind that particular instruction you stepped into. You can use the execution stack trace to retrace the execution flow.

If you stepped into an instruction, take your time to first read what's behind that code line. In many cases, it's enough to throw a look at the code to spot what happens, then, you can go back to where you were before stepping into. I often observe students rushing into debugging the method they stepped into without taking a breath first and reading that piece of code. Why is it important to read the code first? Because stepping into a method is in fact, opening another investigation plan, so, if you want to be efficient, you have to retake the investigation steps:

1. Read the method and find out which is the first line of code you don't understand.
2. Add a breakpoint on that line of code, and start the investigation from there.

But, often, you'll observe that stopping a bit and reading the code shows you that you don't need to continue the investigation in that plan. If you already get what happens, you need just to return to where you were previously. And you can do so using the "step out" operation. Figure 2.18 shows you what happens when using step out from the `extractDigits()` method: the execution returns to the previous investigation plan in the `decode(List<String> input)` method.



TIP The step out operation can save you time. Just remember it exists! When entering a new investigation plan (by stepping into a code line), first read the new piece of code. Step out from the new investigation plan you stepped into once you understand what it does.

```

package main;

import java.util.List;
import java.util.stream.Collectors;

public class Decoder {

    public Integer decode(List<String> input) {
        int total = 0;
        for (String s : input) {
            var digits = new StringDigitExtractor(s).extractDigits();
            total += digits.stream().collect(Collectors.summingInt(i -> i));
        }
        return total;
    }
}

```

When you step out from the extractDigits() method, the execution returns to the previous investigation plan.

You can also observe in the execution stack trace that the execution plan of the extractDigits() method was closed and the execution returned to the decode() method.

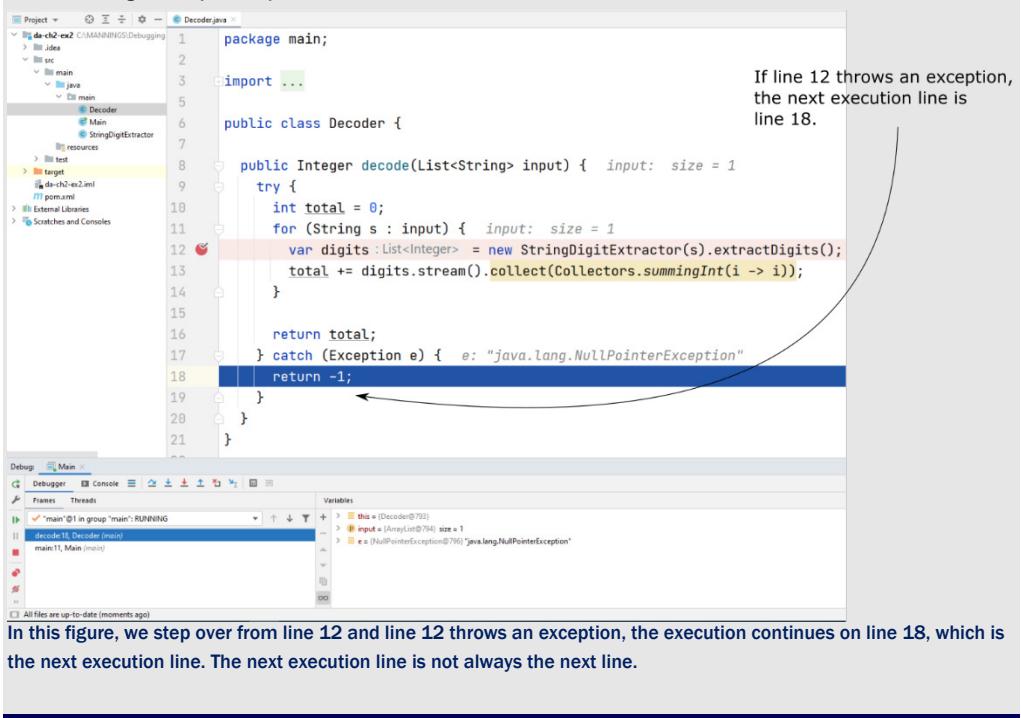
Figure 2.18 The step out operation helps you close an investigation plan and return to the previous in the execution stack trace. Using the step out operation is helpful to save time since you don't have to step over each instruction until the current execution plan closes by itself. Stepping out offers you a shortcut to return to the previous execution plan you were investigating.

Why is the next execution line not always the next line?

When discussing code navigation with the debugger, I often talked about the “next execution line”. I want to make sure I’m clear about the difference between the “next line” and the “next execution line”:

The next execution line is the line of code the app executes next. When we say the debugger paused the execution on line 12, the **next line** is always line 13, but the **next execution line** can be different. For example, if line 12 doesn’t throw an exception in the next figure, the next execution line will be 13, but if line 12 throws an exception, the next execution line is line 18. You find this example in project da-ch2-ex3.

When using the step over operation, the execution will continue to **the next execution line**.



2.3 When using the debugger might not be enough?

The debugger is an excellent tool that helps us analyze the code by navigating through the code to understand how it works with data. But not all the scenarios can be investigated with a debugger. In this section, we discuss some scenarios when using a debugger is not possible or not enough. You need to be aware upfront about the cases when using a debugger is not enough, so you don’t lose time trying it, and, instead, directly start with another technique that would help you in that case.

Here are some of the most often encountered investigation scenarios where using a debugger (or only a debugger) is usually not the right approach:

- Investigating output problems when you don’t know which part of the code creates the output

- Investigating performance problems
- Investigating crashes where the entire app fails
- Investigating multithreaded implementations



Remember that a critical prerequisite for using a debugger is knowing where to pause the execution for starting your investigation.

If you don't know which part of the code generates the wrong output, you need first to find that out before starting debugging it. Depending on the app, it might be more or less easy to find where something happens in the implemented logic. If the app has a clean class design, you will find the part of the app responsible for the output relatively easy. If the app lacks a class design, you might find it challenging to discover where things happen to use the debugger further. In the next chapters, you'll learn several other techniques. Some of these techniques, such as profiling the app or using stubs, will help you identify where to start the investigation with a debugger.

Performance problems are also a particular set of issues you can't usually investigate with a debugger. Cases in which the application is slow or stuck completely are examples of performance problems that happen often. In most cases, profiling and logging techniques (that we'll discuss in chapters 5 through 9) will help you troubleshoot such scenarios. For the particular instances in which the app blocks entirely, in most cases, getting and analyzing a thread dump is the most straightforward investigation path. We'll discuss analyzing thread dumps in chapter 10.

If the app encountered an issue and the execution stopped (*the app crashed*), then you cannot use a debugger on the code. Using a debugger is all about observing the app in execution. If the application doesn't execute anymore, then you clearly have nothing to do with a debugger. Depending on what happened, you might need to audit logs, as we'll discuss in chapter 5, or investigate thread or heap dumps, as you'll learn in chapters 10 and 11.

Most developers find, however, the *multithreaded implementations* to be the most challenging to investigate. Such implementations can be easily influenced by your interference with tools such as the debugger. This interference creates a Heisenberg effect (discussed in chapter 1): the app behaves differently when you use the debugger than when you don't interfere with it. As you'll learn, you can sometimes isolate the investigation to one thread and use the debugger. But in most cases, you'll have to apply a set of techniques that include debugging, mocking and stubbing, and profiling to understand the app's behavior in the most complex scenarios.

2.4 Summary

- Every time you open a new piece of logic you investigate (for example, entering a new method that defines its own logic) we say that you open a new investigation plan.
- Unlike reading a text paragraph, reading code is not linear. Each instruction might create a new plan you need to investigate. The more complex the logic you explore, the more plans you need to open for investigation. The more plans you open, the more complex the process becomes. One of the tricks to speed up a code investigation process is to open as few plans as possible.
- A debugger is a tool that helps you pause the app's execution on a specific line so that you can observe the app's execution step by step and the way it manages data. Using a debugger helps you take out some of the cognitive load of reading code.
- You mark the lines of code where you want the debugger to pause the app execution with breakpoints. When the app execution is paused on a specific line, you can evaluate the values of all the variables in the scope.
- You can step over a line; that means continuing the execution to the next execution line in the same execution plane or step into a line, which means going into details on the instruction on which the debugger paused the execution. You should minimize the number of times you step into a line and rely more on stepping over. Every time you step into a line, the investigation path gets longer and the process more time-consuming.
- Even though at the beginning, using the mouse and the IDE's GUI to navigate through the code seems more comfortable, learning and using the key shortcuts for these operations will help you debug navigate faster. I recommend you learn the key shortcuts of your favorite IDE and use them instead of triggering the navigation with the mouse.
- After stepping into a line, first read the code and try to understand it. If you have already figured out what happens, use the step-out operation to return to the previous investigation plan. If you don't figure out what happens, identify the first unclear instruction, add a breakpoint, and start debugging from there.

3

Finding problem root causes using advanced debugging techniques

This chapter covers

- Using conditional breakpoints to investigate specific scenarios
- Using breakpoints to log debug messages in the console
- Changing data while debugging to force the app to act in a specific way
- Rerunning a certain part of the code while debugging

In chapter 2, we started discussing the most common ways to use a debugger. When debugging a certain piece of implemented logic, developers often use code navigation operations such as stepping over a line, stepping into a line, and out of it. Knowing how to properly use these operations help you investigate a given piece of code to understand better or find a given issue.

But a debugger is a more powerful tool than many developers are aware of. Now and then, developers struggle a lot when debugging code using the basic navigation operations only. At the same time, developers could spend a lot less time using some of the other (less known) approaches a debugger offers.

In this chapter, you'll learn how to get the most out of the features the debugger offers:

- Conditional breakpoints
- Breakpoints as log events
- Modifying in-memory data
- Dropping execution frames

We'll discuss some beyond-basic ways to navigate code you investigate, and you'll learn how and when using these approaches help you. We'll use code examples to discuss these

investigation approaches so you can understand how you could use them to save time, and when to avoid them.

3.1 Minimizing the investigation time with conditional breakpoints

In this section, we discuss the use of conditional breakpoints to pause the app's execution on a given line of code only in specific conditions.



A **conditional breakpoint** is a breakpoint to which you associate a condition, such that the debugger only pauses the execution if the condition fulfills. Conditional breakpoints are helpful in investigation scenarios when you are only interested in how a part of the code works with given values — using conditional breakpoints where appropriate saves you time and helps you more easily understand how your app works.

Let's take an example to understand how conditional breakpoints work and typical cases where you'll need to use them. Listing 3.1 presents a method that returns the sum of the digits in a list of `String` values. You might already be familiar with this method from chapter 2. We'll use this piece of code here as well to discuss conditional breakpoints. We'll then compare this simplified example with similar situations you could encounter in real-world cases. To apply the approaches we discuss, you find this example in project `da-ch3-ex1` provided with the book.

Listing 3.1 A piece of code where you could use conditional breakpoints for investigation

```
public class Decoder {

    public Integer decode(List<String> input) {
        try {
            int total = 0;
            for (String s : input) {
                var digits = new StringDigitExtractor(s).extractDigits();
                var sum = digits.stream().collect(Collectors.summingInt(i -> i));
                total += sum;
            }

            return total;
        } catch (Exception e) {
            return -1;
        }
    }
}
```

Often, when debugging a piece of code, you are only interested in how the logic works for specific values. For example, say you suspect the implemented logic doesn't work well in a

given case (for example, some variable has a certain value), and you want to prove it. Or you simply want to understand what happens in a given situation to have a better overview of the whole functionality.

Suppose that, in this case, you only want to investigate why the variable sum is sometimes zero. How could you only work on this case? You could use the step-over operation to navigate the code until you observe that the method returned zero. This approach would likely be acceptable in a demo example such as this one (small enough). But in a real-world case, it might be that you would have to step over a lot of times until you reach the case you expect. It could even be that you don't even know when the specific case you want to investigate appears in a real-world scenario.

Using conditional breakpoints is better than navigating through code until you get to the conditions you desire to research. Figure 3.1 shows you how to apply a condition to a breakpoint in IntelliJ IDEA. Right-click the breakpoint you want to add the condition for and write the condition for which the breakpoint applies. The condition needs to be a Boolean expression (it should be something that can be evaluated with true or false). Using the `sum == 0` condition on the breakpoint, you tell the debugger to consider that endpoint and stop the execution only when it reaches a case where the variable sum is zero.

In IntelliJ, you right-click on the breakpoint to define its condition. In this example, the debugger only stops on this breakpoint when the variable "sum" is zero.

You can add a condition on certain breakpoints. The debugger considers these breakpoints only if their condition evaluates to "true".

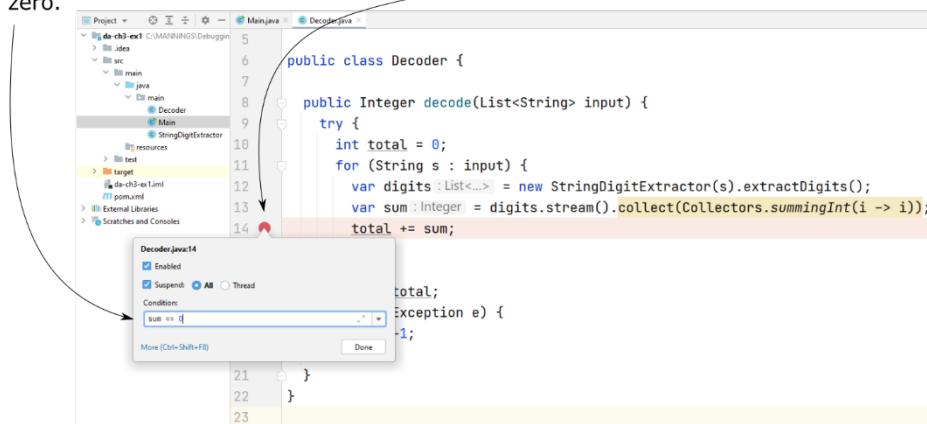


Figure 3.1 Using a conditional breakpoint to stop the execution just for specific cases. In this figure, we want to pause the execution on line 14 only if sum is zero. We can apply a condition on the breakpoint that instructs the debugger to consider that breakpoint only if the given state is true. This approach helps you get a scenario you want to investigate faster.

When you run the app with the debugger, the execution pauses only when the loop first iterates on a string that contains no digits, as you observe in figure 3.2. This situation causes the variable sum to be zero, so the condition on the breakpoint to be evaluated is true.

A conditional breakpoint saves you time since you don't have to search yourself for the suitable case you need to investigate. Instead, you allow the app to run and the debugger pauses the execution only to investigate it at the right time. Although using conditional breakpoints is easy, many developers seem to forget about this approach and lose a lot of time investigating scenarios that they could simplify a lot with conditional breakpoints.

When you run the app with the debugger, the debugger only stops the execution for the first element in the parameter list that doesn't contain digits (for which variable sum will be 0).

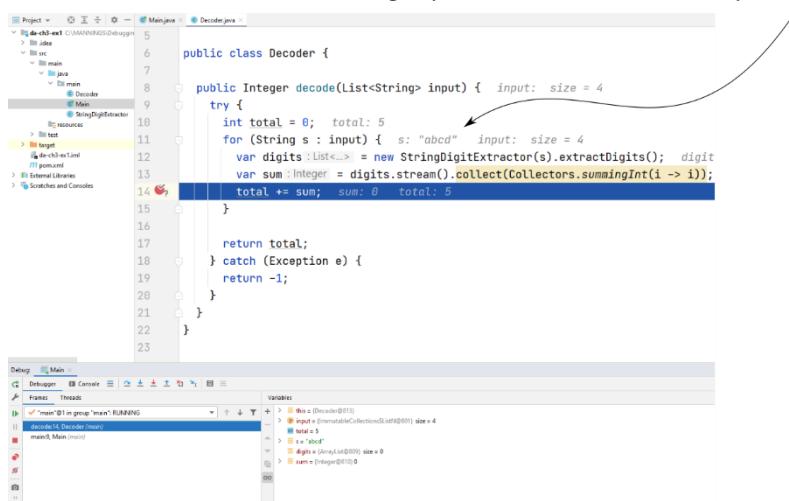


Figure 3.2 A conditional breakpoint. Line 14 in the figure was executed multiple times, but the debugger only paused the execution when the variable sum was zero. This way, we skipped over all the cases we were not interested in, and we'll start directly with the conditions relevant to our investigation.

Conditional breakpoints are excellent. However, mind that they also have their downsides. Conditional breakpoints can dramatically affect the performance of the execution since the debugger has to continuously intercept the values of the variables in the scope you use and evaluate the breakpoint conditions.



TIP Use a small number of conditional breakpoints.
Preferably, use only one conditional breakpoint at once
to avoid slowing down the execution too much.

Another way to use conditional breakpoints is to log specific execution details such as various expression values and the stack traces for particular conditions.

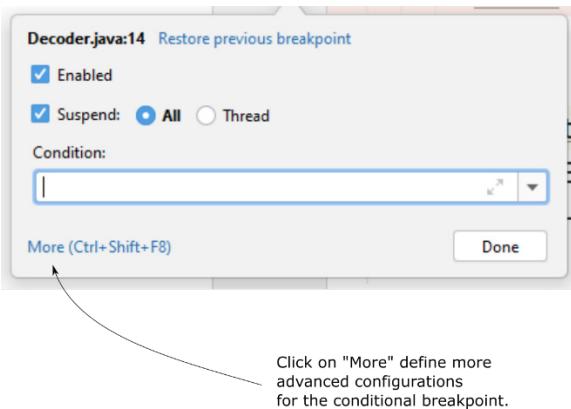
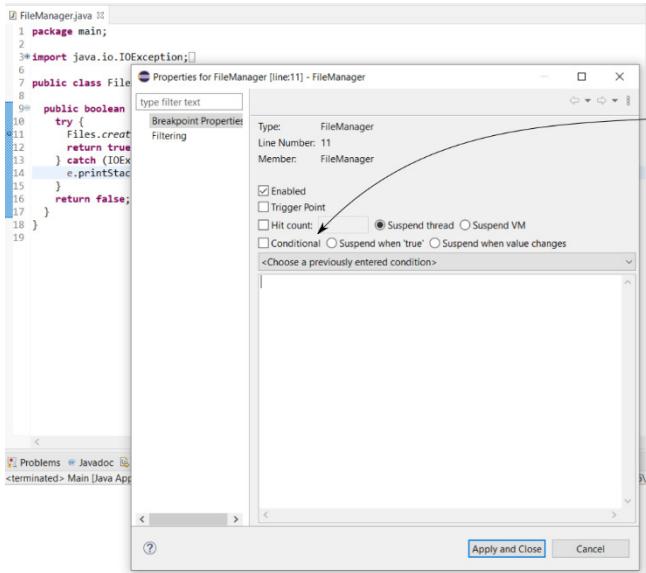


Figure 3.3 To apply advanced configuration on the breakpoint in IntelliJ, you can click on the More button as presented in the figure.

Unfortunately, this feature only works in certain IDEs. For example, even if in Eclipse you can use conditional breakpoints in the same way as described earlier in this chapter, you can't use breakpoints only for logging execution details.



In Eclipse IDE, you can define a conditional breakpoint. However, you can't log specific execution details instead of suspending the thread.

Figure 3.4 Not all IDEs offer the same debugging tools. All IDEs give you the basic operations, but some features, such as logging the execution details instead of pausing the execution, might be missing. In Eclipse IDE, you can define conditional breakpoints, but you can't use the logging feature.

Even if this feature is not part of all IDEs, I found it very helpful in particular scenarios, so let's discuss an example of how to use it in section 3.2.

You might ask yourself if you should use only IntelliJ IDEA like I do for these examples. Even if most examples in this book use IntelliJ IDEA, this doesn't mean this IDE is better than others. Throughout the time, I've used many IDEs with Java, such as Eclipse, Netbeans, or JDeveloper. My recommendation is you shouldn't become too comfortable with using one IDE. Instead, try to use various options so you can decide which one is a better fit for you and your team.

3.2 Using breakpoints that don't pause the execution

In this section, we discuss using breakpoints to log messages you can later use in investigating the code. One of my favorite ways to use breakpoints is to log details that can help me further understand what happened during the app's execution without pausing the execution. As you'll learn in chapter 5, logging is an excellent investigation practice in some cases. Many developers struggle adding log instructions when they need to use this approach, while in many cases, they could have simply used a conditional breakpoint.

Figure 3.5 shows you how to configure a conditional breakpoint that doesn't pause the execution. Instead, the debugger logs a message every time the line marked with the breakpoint is reached. In this case, the debugger logs the value of the `digits` variable and the execution stack trace.

You can use a breakpoint only to log certain details without suspending the execution.

Here, when variable sum is 0, the value of the digits variable and the stack trace is printed in the console

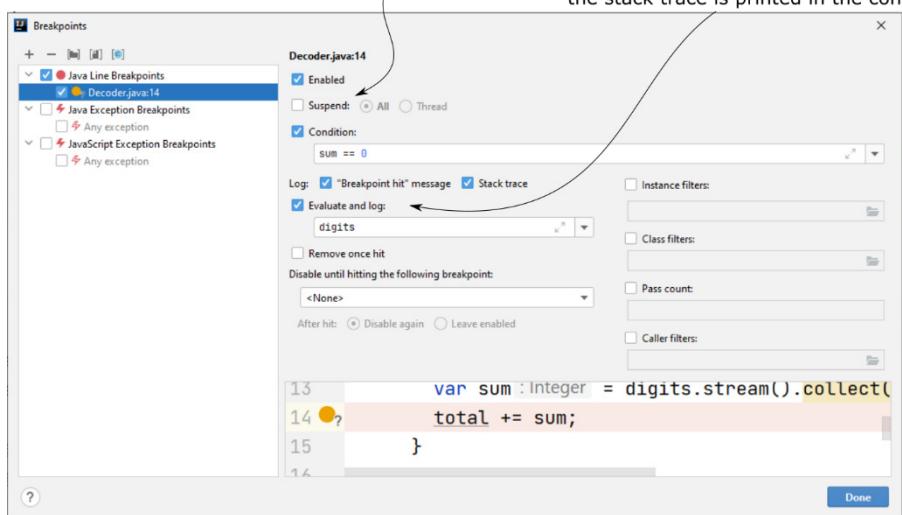


Figure 3.5 Conditional breakpoint advanced configuration. Besides specifying a condition for the breakpoint, you can instruct the debugger not to suspend the execution for the given breakpoint. Instead, you could simply log the data you need to understand your case.

Figure 3.6 shows the result of running the app with the conditional breakpoint configured, as presented in figure 3.5. Observe that the debugger logged in the console the execution stack trace and the value of the `digits` variable, which is an empty list: `[]`. This kind of information leads you to solve the puzzles of the code you investigate in a real-world scenario.

The screenshot shows an IDE interface with two tabs: `Main.java` and `Decoder.java`. In `Decoder.java`, line 14 is highlighted with a red rectangle and has a yellow circle at its start, indicating a breakpoint. The code implements a `Decoder` class with a `decode` method that processes a list of strings to calculate a total sum of digits. In the `Main.java` tab, there is a single line of code: `Decoder.main();`. The IDE's Debug view is open, showing the following log output:

```

Connected to the target VM, address: '127.0.0.1:53296', transport: 'socket'
Breakpoint reached at main.Decoder.decode(Decoder.java:14)
Breakpoint reached
    at main.Decoder.decode(Decoder.java:14)
    at main.Main.main(Main.java:9)
[]
15
Disconnected from the target VM, address: '127.0.0.1:53296', transport: 'socket'

Process finished with exit code 0

```

Using this conditional breakpoint, the debugger doesn't stop the execution anymore. Instead, it logs in the console the value of the digits variable and the execution stack trace.

Figure 3.6 Using breakpoints without pausing the execution. Instead of pausing the execution on the line marked with a breakpoint, the debugger logs a message when the line has been reached. The debugger also logs the value of the digits variable and the execution stack trace.

Execution stack trace – visual vs. text representation

Observe the way the stack trace is printed in the console. You'll often find the execution stack trace in a text format rather than a visual one. The advantage of the text representation of the execution stack trace is that it can be stored in any text format output, such as the console or a log file.

The figure shows you a comparison between the visual representation of the execution stack trace provided by the debugger and its text representation. As you observe, you are provided in both cases with the same essential details that help you understand how a specific line of code got to be executed.

In this particular case, the stack trace tells us that the execution started from the `main()` method of the `Main` class. Remember that the first layer of the stack trace is the bottom one. On line 9, the `main()` method called the `decode()` method in the `Decoder` class (layer 2), which further called the line we marked with the breakpoint.

Breakpoint reached

3 at main.StringDigitExtractor.extractDigits(StringDigitExtractor.java:16)
 2 at main.Decoder.decode(Decoder.java:12)
 1 at main.Main.main(Main.java:9)

A comparison between the visual representation of the execution stack trace in the debugger and its text representation. The stack trace shows you how a method got to be called and provides you enough details to understand the execution path.

3.3 Dynamically altering the investigation scenario

In this section, you'll learn another valuable technique that will make your code investigations easier: changing the values of the variables in scope while debugging. In some cases, this approach can save plenty of time. We'll begin with discussing these scenarios where changing variables values on the fly is the most valuable approach. Then I will demonstrate to you how to use this approach with an example.

Earlier in this chapter, we discussed conditional breakpoints. Conditional breakpoints help you tell the debugger to pause the execution in certain conditions only (for example, when a

given variable has a certain value). Often, we investigate logic that executes in a short time, and using conditional breakpoints is enough. For cases such as debugging a piece of logic called through a REST endpoint (especially if you have the right data to reproduce a problem in your environment), you would simply use a conditional breakpoint to pause the execution when appropriate. That's because you know it can't take a long time to execute something called through an endpoint. But suppose the following scenarios:

1. You investigate an issue with a process that takes a long time to execute. Say it's a scheduled process that sometimes takes even over an hour to finish its execution. You suspect that some given parameter values cause the wrong output, and you want to prove that this is the case before you decide how to correct the problem.
2. You have a piece of code that executes fast. But the problem is that you can't reproduce the issue in your environment. The issue appears only in the production environment where, due to security constraints, you don't have access. You believe the issue appears when certain parameters have specific values. You want to prove your theory is right.

In scenario 1, breakpoints (conditional or not) aren't so helpful anymore. Unless you investigate some logic that happens at the very beginning of the process, running the process and waiting for the execution to pause on a line marked with a breakpoint would take too much time (figure 3.7).



You, when the process starts.



You, when the execution finally reached the breakpoint.

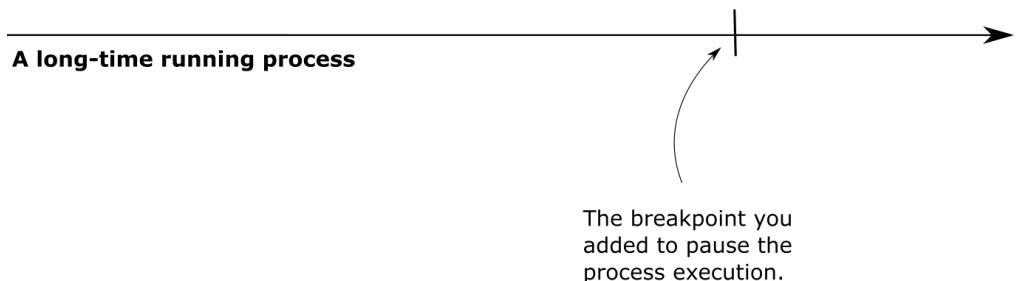
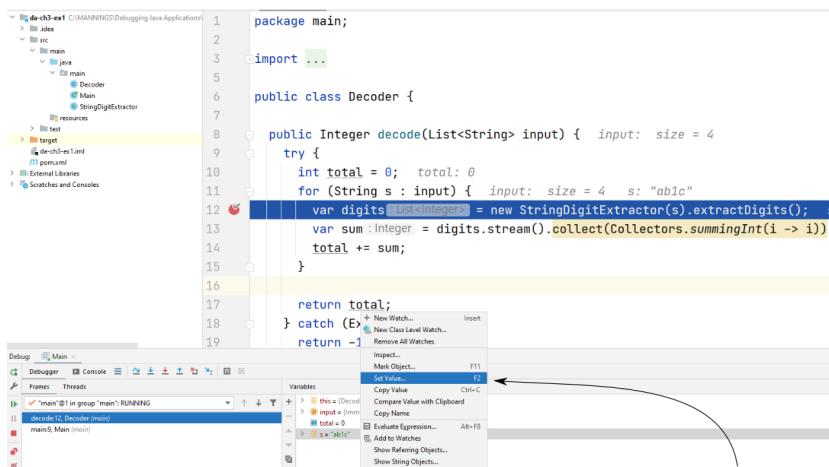


Figure 3.7 Usually, when investigating issues in a long-running process, using breakpoints is not really an option. This is because it can take a long time for the execution to reach the part of code you are investigating, and if you have to rerun the process a few times, you will definitely spend too much time.

For scenario 2, using breakpoints may sometimes be possible. In chapter 4, we'll discuss remote debugging, and you'll find out how and when remote debugging is a helpful investigation technique. But, let's assume for the moment (since we didn't discuss remote debugging yet) that you can't apply remote debugging in this case. Instead, if you have an idea of what causes the problem and you only need to prove it but don't have the right data, you could use on the fly change of values in variables.

Figure 3.8 shows you how to change the data in one of the variables in the scope when the debugger pauses the execution. In IntelliJ IDEA you just have to right-click on the variable whose value you want to change. You make this action in the frame where the debugger shows the current values of the variables in scope. To make the explanation more comfortable, we'll use the same example da-ch3-ex1 we also worked with previously in this chapter.



When the debugger pauses the execution on a line, you can set values in the variables in scope. This way, you can create your own investigation scene with the conditions you need for the case you are revising.

Figure 3.8 Setting a new value in a variable in scope. The debugger shows you the values for the variables in scope when it pauses the execution on a given line. Besides being able to see the variable values, you can also change them to create a new investigation case. This approach helps you, in some cases, to validate that you are right about what you suspect the code does.

Once you selected which variable you want to change, set the value as presented in figure 3.9. Remember that you have to use a value according to the variable's type. That means, if you change a `String` variable, you still need to use a `String` value; you cannot use a `long` or a `Boolean`.

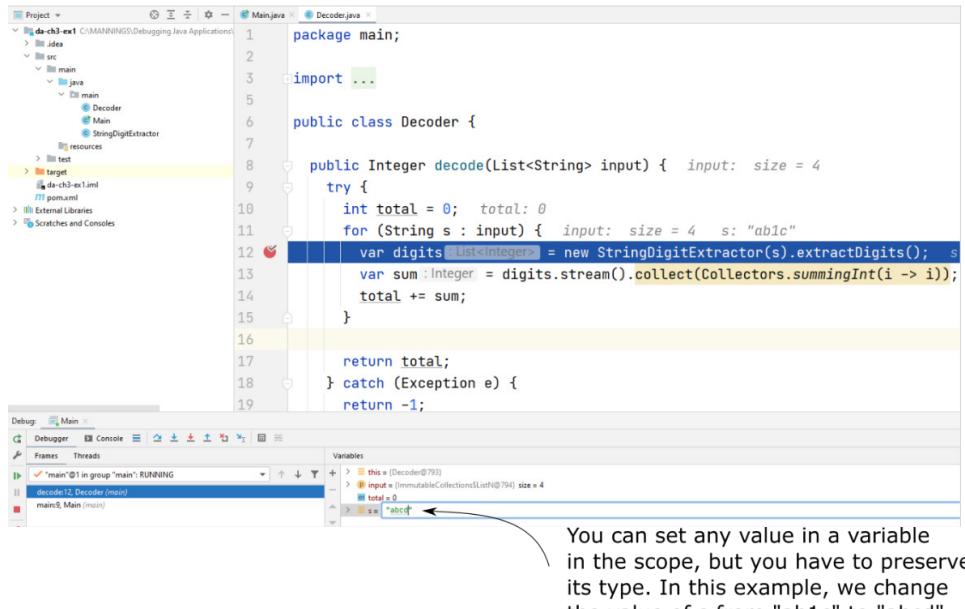


Figure 3.9 Change the variable's value to observe how the app's execution behaves in different conditions.

When you continue the execution, as presented in figure 3.10, you observe the app now uses the new value. Instead of calling `extractDigits()` for value "ab1c", the app used the value "abcd". The list the method returns is empty because the string "abcd" doesn't contain digits.

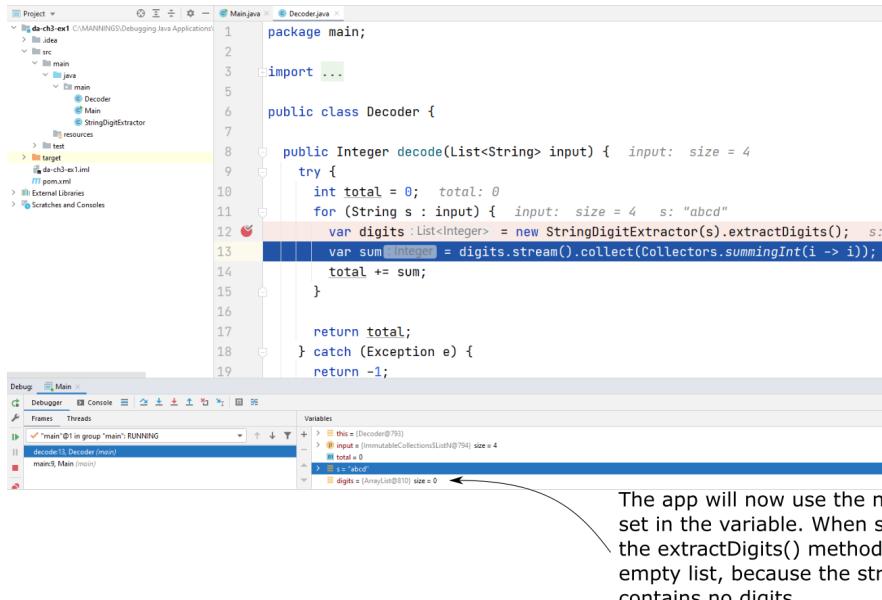


Figure 3.10 When using the step-over operation, the app now uses the new value you set to the “s” variable. Method extractDigits() returns an empty list because string “abcd” doesn’t contain digits. Setting values in variables on the fly helps you test different scenarios even if you don’t have the input data you need.

Let’s compare the conditional breakpoints approach we discussed in section 3.1 with the approach of changing data on the fly that we discuss in this section. In both cases, you need to first have an idea of the part of the code that potentially causes the problem.

You use conditional breakpoints if

- You have the right data that causes the scenario you want to investigate. In our example, we need to have the value for which we want to execute the behavior in the provided list.
- It doesn’t take too long to execute the code you investigate. For example, suppose we had a list with many elements, and for each element, it would take several seconds for the app to process it. In this case, using a conditional breakpoint might assume you have to wait a long time to investigate your case.

You use the approach of changing the variable values if

- You don’t have the right data to cause the scenario you want to investigate.
- Executing the code takes too long.

All right! I know what you are thinking now: “Why are we using conditional breakpoints at all?”. From our discussion, it might look like you’d better avoid using conditional breakpoints at all since you can just create any environment you need to investigate by changing the variable’s values on the fly.

However, I don't want you to understand this is the case, since both techniques have advantages and disadvantages. Changing the values of the variables might look like an excellent approach until you have to change more than a couple of values. You'll observe that creating your environment while debugging is easy when you change one or two values. Still, when you go with more than these, the complexity of the scenario you investigate becomes more and more challenging to manage.

3.4 Rewinding the investigation case

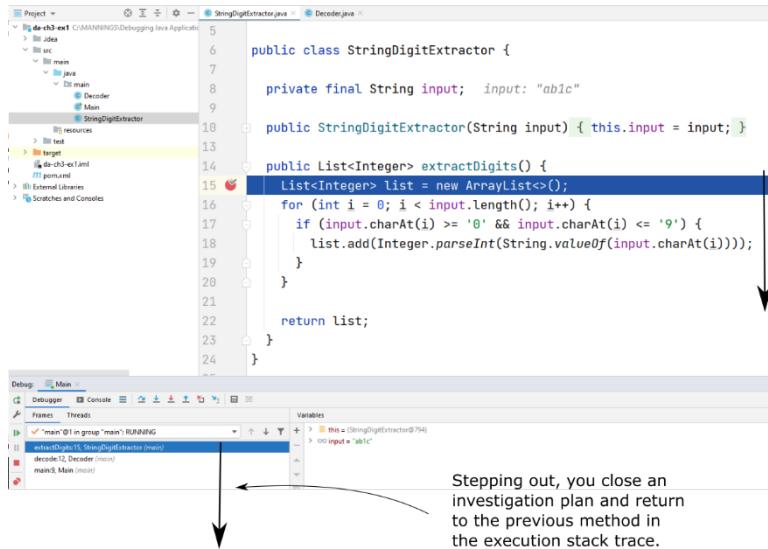
Going back in time is not something we're familiar with or used to. However, in what concerns debugging, turning the time arrow is sometimes possible. In this section, we discuss when and how we can "go back in time" while investigating code with a debugger. We name this approach "dropping frames" or "dropping execution frames", or "quitting execution frames".

We'll use an example where we'll demonstrate this approach using IntelliJ IDEA. Based on this example, we'll compare this approach with the ones we discussed in the previous sections of this chapter, and then we'll also determine when this technique can't be used.

Dropping an execution frame is in fact going back one layer in the execution stack trace. For example, suppose you stepped into a method and want to go back: you can drop the execution frame to return to where the method was called.

Many developers confuse the "dropping a frame" with "stepping out," most likely because the current investigation plan closes in both cases, and the execution returns to where the method is called. However, there's a big difference. When you step out of a method, the execution continues in the current plan until the method returns or throws an exception. Then, the debugger pauses the execution right after the current method exits.

Figure 3.11 shows you how step-out works using the example you find in project da-ch3-ex1. You are in the `extractDigits()` method, which, as you can see from the execution stack trace, has been called from the `decode()` method in the `Decoder` class. If you use the step-out operation, the execution continues in the method that called `extractDigits()` until the method returns. Then, the debugger pauses the execution in the `decode()` method. ***In other words, stepping out is like fast-forwarding this execution plan to close it and return to the previous one.***

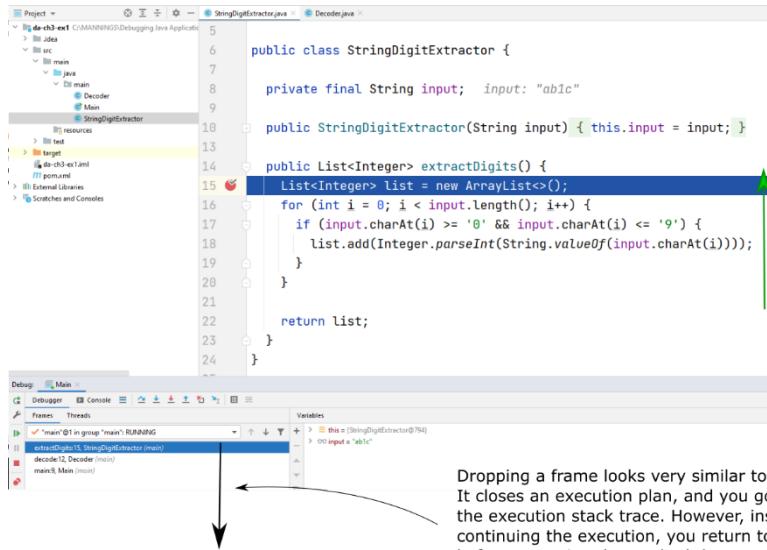


When you step out, you let the current method execute. You return to the previous method one line after the one that created the current investigation plan.

Stepping out, you close an investigation plan and return to the previous method in the execution stack trace.

Figure 3.11 Stepping out closes the current investigation plan by executing the method and then pausing the execution right after the method call. This operation helps you continue the execution and return one layer in the execution stack.

When dropping an execution frame, unlike stepping-out, the execution return in the previous plan before the method call. This way, you can replay the call. **If the step-out operation is like fast-forward, dropping an execution frame (figure 3.12) is like rewind.**



When you drop a frame, you return to the previous layer in the execution stack trace before the method was called.

Figure 3.12 When you drop a frame, you return to the previous layer in the execution stack trace before the method execution. This way, you can replay the method execution either by stepping again into it or stepping over it.

Figure 3.13 shows you, relative to our example, a comparison between stepping out from the `extractDigits()` method or dropping the frame that is created by the `extractDigits()` method. If you step out, you'll go back to line 12 in the `decode()` method where `extractDigits()` have been called from, and the next line the debugger will execute is line 13. If you drop the frame, the debugger goes back to the `decode()` method, but the next line that will execute is line 12. Basically, the debugger returned before the execution of the `extractDigits()` method.

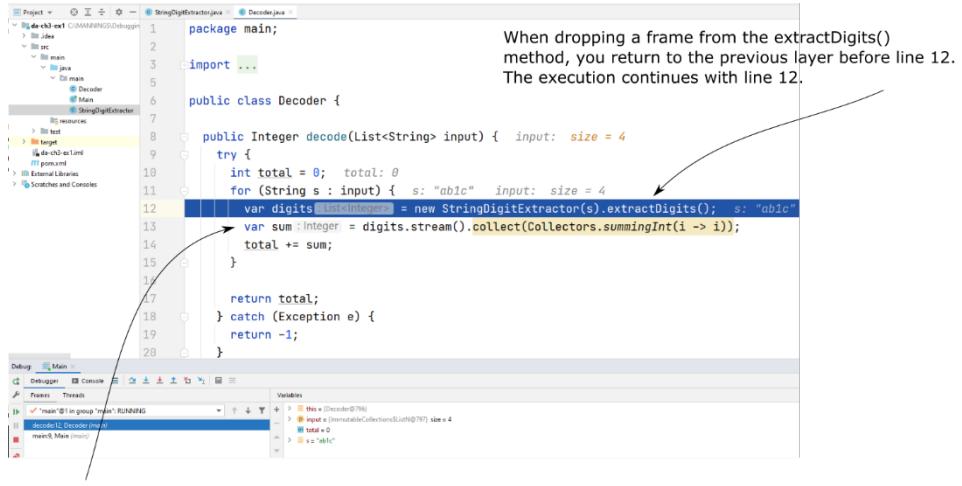


Figure 3.13 Dropping a frame vs. stepping out. When you drop a frame, you return before the method's execution. When you step out, you continue the execution but close the current investigation plan (represented by the current layer in the execution stack).

Figure 3.14 shows you how to use the drop frame functionality in IntelliJ IDEA. To drop the current execution frame, right-click on the method's layer in the execution stack trace and select "Drop Frame".

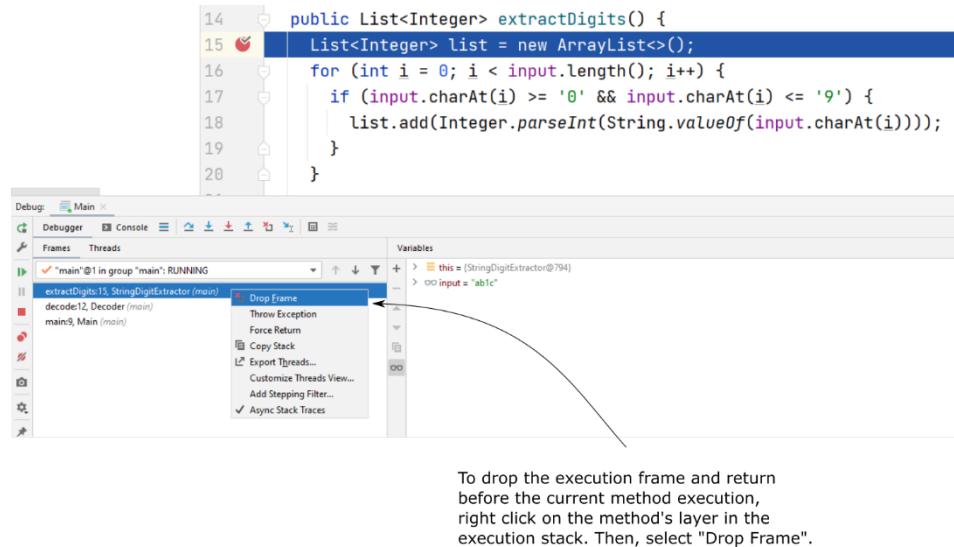


Figure 3.14 When using IntelliJ IDEA, you can drop a frame by right-clicking on the method's layer in the execution stack trace and then selecting “Drop Frame”.

So, why is the “drop frame” useful, and how it helps you save time? Whether you use an endpoint to find a specific case you want to investigate or fabricate one by changing the values of the variables as discussed in section 3.3, you’ll still sometimes find it useful to repeat the same execution several times to understand it. Understanding a certain piece of code is not always trivial, even if you use the debugger to pause the execution and take it step by step. But if you can go back now and then, reviewing the steps and how specific code instructions change the data may help you understand what’s going on.

But, you also need to pay attention when you decide to repeat particular instructions by dropping the frame. There are cases where using this approach could be more confusing than helpful. Remember that if you run any instruction that changes values outside of the app’s internal memory, you can’t undo that by dropping the frame. Examples of such cases where something is changed outside the app are (figure 3.15):

- Modifying data in a database (insert, update, or delete)
- Changing the filesystem (creating, removing, or changing files)
- Calling another app resulting in data being changed for that app.
- Adding a message into a queue that is read by a different app and results in data changes for that app.
- Sending an email message.

You can drop a frame that results in committing a transaction that changes data in a database, but going back to a previous instruction won’t also undo the changes made by the transaction. If the app calls an endpoint that posts something into a different service, the

changes resulted by the endpoint call cannot be undone by dropping the frame. If the app sends an email message, dropping the frame cannot take back the message, and so on.

Even if you can go back to a previous instruction using drop frame, some events cannot be undone.

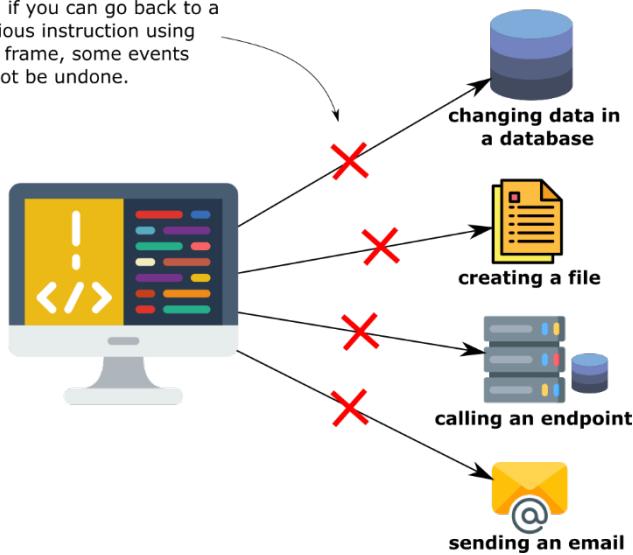


Figure 3.15 The drop frame operation cannot undo some events. Some examples of such events are changing data in the database, changing data in the file system, calling another app, or sending an email message. You need to be careful with such cases as they might affect your investigation.

You need to be careful with these scenarios where data is changed outside the app, as sometimes repeating the same code won't even have the same result. Take as an example a simple piece of code presented in listing 3.2 (which you can find in project da-ch3-ex2). What happens if you drop the frame after the execution of the line that creates a file?

```
Files.createFile(Paths.get("File " + i));
```

Not only that the created file remains in the file system, but the second time when you execute the code after dropping the frame results in an exception (because the file already exists). This is a simple example that proves going back in time while debugging is not always helpful. The worst part is that, in real-world cases, it's not even that obvious as it is in this example. So, my recommendation is to avoid repeating the execution of large pieces of code and, before deciding to use this approach, make sure that part of the logic doesn't make external changes.

If you observe differences that seem unnatural after running a dropped frame again, it might be because the code changes something externally. Often, in big apps, observing such behavior is not straightforward. For example, your app might use a cache, log data using a certain library in a difficult way to observe or execute code that is completely decoupled through interceptors (aspects).

Take a look at listing 3.2. Calling the `Files.createFile()` method creates a new file in the file system. If you drop the frame after running this line, you'll return before the `createFile()` method has been called. However, this doesn't undo the file creation.

Listing 3.2 A method that makes changes outside the app when executing

```
public class FileManager {

    public boolean createFile(int i) {
        try {
            Files.createFile(Paths.get("File " + i));      #A
            return true;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

#A Creating a new file in the file system.

3.5 Summary

- A conditional breakpoint is a breakpoint to which you associate a Boolean condition. The debugger pauses the execution only if the provided condition is true. You can use conditional breakpoints to pause the execution only when particular conditions apply. This way, you save time by skipping the need to navigate yourself through code until you get the desired investigation scene.
- You can use breakpoints to log messages in the console with the values of certain variables during the execution without pausing the execution on that specific line. Log messages are often helpful, and this approach is quite helpful because you can add log messages without changing the code you investigate.
- When the debugger pauses the execution on specific lines of code, you can alter the data on the fly to create custom scenarios according to what you want to investigate. This approach helps you avoid needing to wait until the execution gets into a conditional breakpoint. In some cases, where you don't have an appropriate environment, changing data while debugging saves you a lot of time you would have needed to prepare the data in the environment.
- Changing variables' values to create a custom investigation scenario could save a lot of time when trying to understand only a piece of the logic of a long-running process or when you don't have the desired data in the environment where you run the app. However, changing more than one or two variable values at a time may add complexity and make your investigation more challenging.

- You can step out of an investigation plan and return to the point before the method was called. This way to go back in time in execution is called “dropping a frame”. But remember that returning to a previous point in execution is not always possible without introducing a side effect. If the app changed anything externally (for example, committed a transaction and changed some database records, changed a file in the filesystem, or made a RESTful call to another app), returning to a previous execution step doesn’t undo these changes.

4

Finding issues' root causes in apps running in remote environments

This chapter covers

- Debugging an app installed in a remote environment
- Upskilling debugging techniques with a hands-on example

One of my friends recently had a problem where a particular part of the software he was implementing was very slow. Generally, when we have such performance issues, we suspect an I/O interface causes it (such as a connection to a database or reading or writing in a file). Remember I told you in chapter 1 that such interfaces often cause slowness, so it's common to suspect them. But in their case, none of these were causing the issue. They individually investigated all of the suspicious calls that could (in their perspective) be the cause of the problem. Neither of them was to blame. "Seems we're in big trouble, but we have a last resort.", I remember he said. The "last resort" was to connect the debugger directly to the running app on the environment where the issue occurred, and this action saved the day.

The performance issue was caused by the simple generation of a random value (a universally unique identifier [UUID] they stored in the database). It turned out that the operating system has something called "entropy". The entropy is randomness collected by the operating system the app uses when generating random values. The operating system uses hardware sources (such as mouse movements, the keyboard, and so on) to collect this entropy. But when we deploy the app in a virtualized environment such as a virtual machine or a container (which is pretty common for app deployments today), the operating system has less sources to create its entropy. Thus, the app can get into a situation where there's not enough entropy to create the random value it needs. This situation causes performance problems and, in some cases, can also have a negative impact on the app's security.

Such a problem as the one in this story could be really challenging to investigate without connecting directly to the environment where the problem occurs. For such scenarios, you must learn how remote debugging works. You can only examine certain cases in particular environments. Suppose your client observes an issue that doesn't happen when you run the app on your computer. You definitely cannot solve it with "it's working on my machine".

You need to connect to that environment where the problem occurs to investigate a specific problem that you cannot reproduce on your computer. Sometimes, you don't have other options, and you have to take the challenging path, but in some cases, the environment where the issue occurs is open for remote debugging. Remote debugging, or debugging an app installed in an external environment, is the subject we discuss in this chapter.

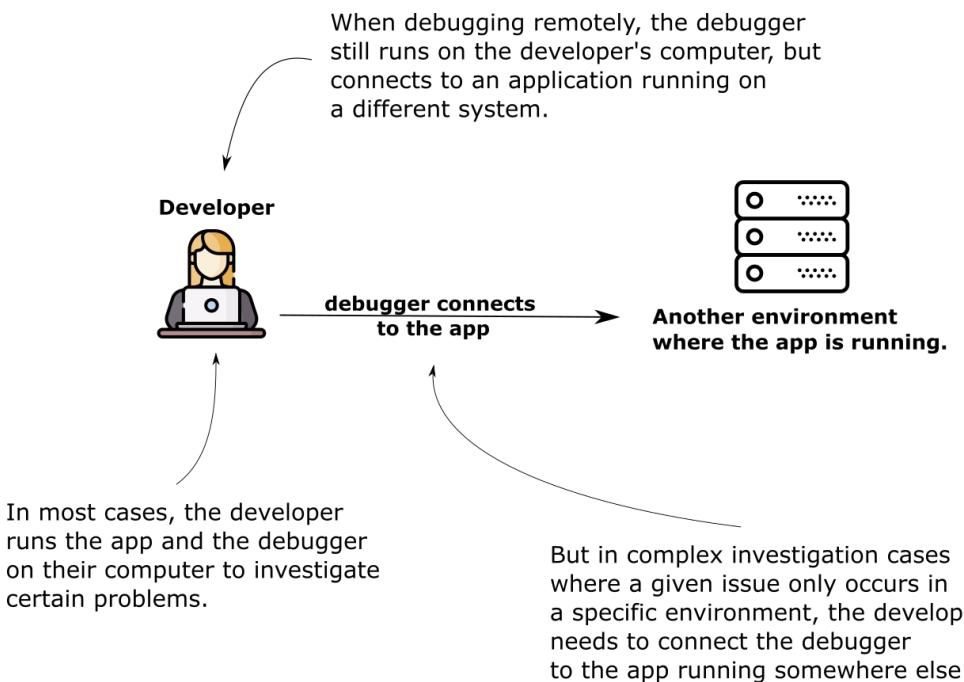


Figure 4.1 Remotely debugging an app. The developer can run the debugger tool locally on their computer but connect it to an app instance running in a different environment. This approach allows the developer to investigate problems that only occur in specific environments.

We'll start the chapter by discussing what remote debugging is and when you expect to use it, and also when you should not use this method. Then, to apply this technique, we'll take a scenario of an issue we need to investigate. You'll learn how an app needs to be configured such that remotely debugging it is possible and how to connect and use the debugger for a remote environment.

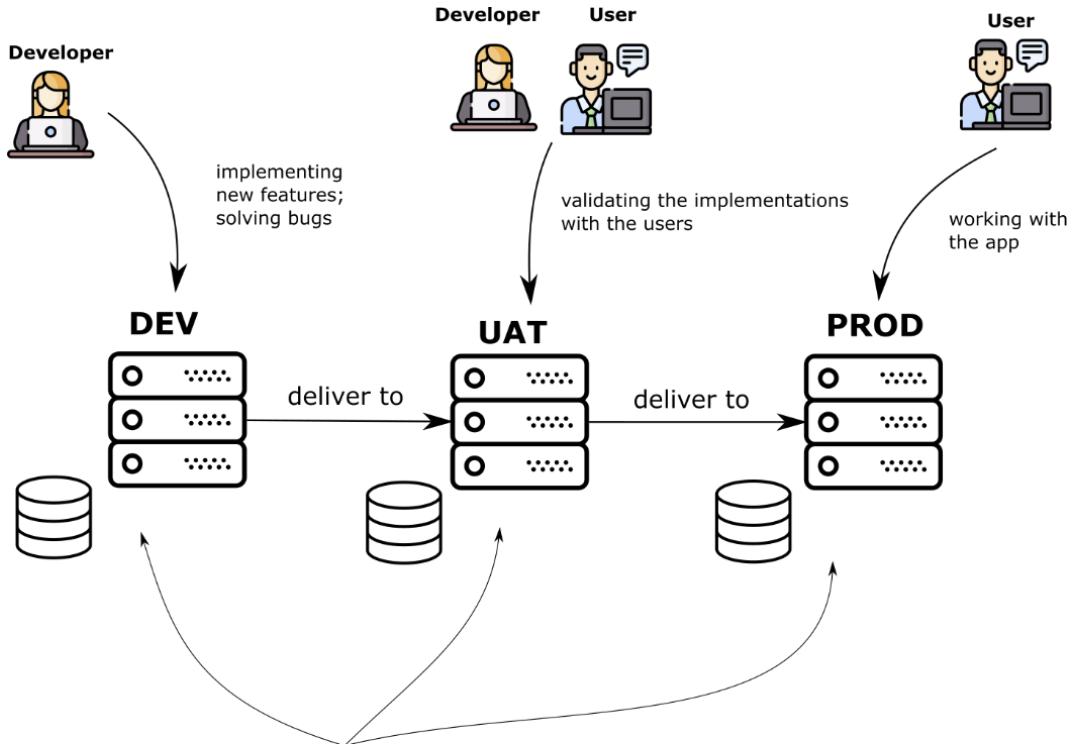
4.1 What is remote debugging?

In this section, we discuss what remote debugging is, when to use it, and also when to avoid it. Remote debugging is basically nothing more than applying the debugging techniques you learned in chapters 2 and 3 on an app that doesn't run locally on your system but instead runs in an outside environment. But why would you need to use such techniques in a remote environment? To answer this question, let's briefly review a usual software development process.

When developers implement an app, they don't write it for their local systems. The final purpose of an app we implement is to deploy it in a production environment where the app helps the users solve various business problems. Moreover, when implementing the software, we often don't deploy the app directly in the environment where the users use it. The environment where the user uses the app is called the production environment. Instead, we use similar environments to roughly test the capabilities and fixes we implement before installing them in an environment where they are officially used with real data.

As described in figure 4.2, a development team uses at least three environments when developing an app:

1. *The development environment (DEV)* is an environment similar to the one where the app is deployed. Developers mainly use this environment to test the new capabilities and fixes they implement after developing them on their local system.
2. *The user acceptance test environment (UAT)- once successfully tested in the development environment, the users need to confirm that the new implementations and fixes work according to their expectations. The app is installed in the user acceptance test environment (UAT). The users can test the new implementations and confirm that they work fine before the app is delivered to the environment to use it with real data.*
3. *The production environment (PROD) – After the users confirm a new implementation works as expected and feel comfortable using it, the app is installed in the production environment.*



The environments might run different app versions, use different configurations, and usually they connect to different databases.

Figure 4.2 When working on a real-world app, developers often use multiple environments to run the app they build. First, the developers build the app in development (dev) environments. Then, once a feature or a solution is ready, they present it to the users (or stakeholders of the app) using a user acceptance test (UAT) environment. Finally, the stakeholders confirm the implementation works fine in the UAT environment before installing it in a production (PROD) environment.

But what if an implementation works fine on your local computer but behaves differently in another environment? You might wonder, how is it possible for an app to work differently? Even when using the same compiled app in two different environments, we could observe differences in the app's behavior. Some reasons for these differences could be:

- The data available in the app's environments is different. Different environments use different database instances, different configuration files, and so on.
- The operating system in which the app is installed is not the same.
- The way the deployment is orchestrated could be different. For example, one environment could use virtual machines for the deployment, while another uses a containerized solution.
- Permission setup could be different in each environment.

- The environments could have different resources (allocated memory or CPU).

The previous list is just some of the many things that could make a given output or behavior different. The last time I had such a problem (not long ago) in an app, the app produced a different output due to the result of a request sent to a web service the app used in the implemented use case. Because of security issues, we didn't use the same endpoint in dev, and we couldn't connect to the one the app used in the environment with the problem. These conditions made the investigation challenging (honestly, we didn't even consider that endpoint could cause our issue until we started debugging).

Remote debugging can really help you understand the software behavior faster in such cases where the behavior you investigate is isolated in a specific deployment. However, keep one crucial piece of advice in mind: **Never use remote debugging in the production environment** (figure 4.3). I would add here also, make sure that you always understand at least the main differences between the environments you use.



TIP Paying attention to how the environments you use differ from one another gives you clues of what could go wrong. It could even spare you the time of investigating an issue in some cases where just knowing these details will empirically give you the answer to a problem.

As you'll learn further in this chapter, you need to attach a piece of software we name "agent" to the app execution to enable remote debugging. Some of the consequences of attaching the debugging agent (and why you shouldn't do this in a production environment) are

- The agent causes slowness in the app execution; this slowness can cause performance problems.
- The agent needs a way to communicate with a debugger tool through the network. To enable this, you need to make specific ports available in the network, and opening such ports could cause vulnerability issues.
- Debugging a specific piece of code could interfere with what the users are doing if they use the same part of the app simultaneously.
- Sometimes debugging could block the app indefinitely and force you to restart the process to make it work again.

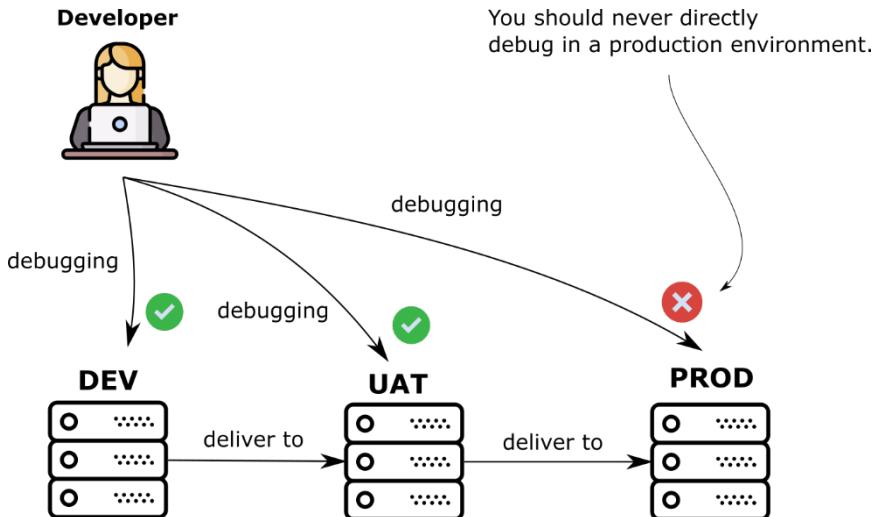


Figure 4.3 The developers implement the app using the development (DEV) and user acceptance test (UAT) environments. In these environments, it's OK to debug the apps. But remember never to debug apps in the production (PROD) environment. Allowing debugging in production could affect the app's execution, interfere with the users' actions and even expose sensitive data becoming a security vulnerability.

4.2 Investigating in remote environments

In this section, we'll work with a scenario where we'll consider debugging an app that runs in a remote environment. I'll start by describing the scenario in section 4.2.1. Then, in section 4.2.2, using an app provided with this book (project da-ch4-ex1), we'll discuss starting an app for remote debugging and how to discover the issue by attaching a debugger to the remotely running app and using the techniques you learned in chapters 2 and 3.

4.2.1 The scenario

In this section, I'll describe the scenario that we'll use in this chapter as our case study. Suppose you work in a team that implements and maintains a large application used by many clients to manage their product inventory. Recently, your team implemented a new capability that helps your client easily manage their costs. The team successfully tested the behavior in the development environment and installed the app in the UAT environment to allow the users to validate the feature before moving it to production. However, the responsible person for testing the new capability calls you and shows you that the web interface where the new data should be displayed shows nothing.

Concerned, you take a look and quickly observe that the problem is not the frontend but an endpoint on the backend that seems to behave weirdly. When calling the endpoint in the UAT environment, you observe the HTTP response status code is 200 OK, but the app doesn't return the data in the HTTP response (figure 4.4). You check the logs, but nothing shows there either. Since you don't observe the problem either locally or in the development

environment, you decide to remotely connect your debugger to the UAT to find the cause of this issue.

NOTE Even if we discuss debugging an app running in a remote environment, to make the example simpler and allow you to focus easier on the discussed subject, we'll use the local system to run the app to which we connect remotely. For this reason, you'll observe in the visuals that I use "localhost" to access the environment running the app. In a real-world scenario, the app would run on a different system which would be identified with an IP address or DNS name.



Figure 4.4 The scenario you have to investigate. The `/api/product/total/costs` endpoint should return the total costs from the database. Instead, when sending a request to the endpoint, the app behaves weirdly. The HTTP status is 200 OK, but the total costs, which you expected to be a list of values, comes back null in the HTTP response.

4.2.2 Finding issues in remote environments

In this section, we use remote debugging to investigate the case study described in section 4.2.1. We'll start by configuring and running the app to allow the connection with a remote debugger and then attach the debugger to start our investigation.

The app would be already running in a real-world case. In a real-world scenario the app wouldn't be most likely already configured to allow remote debugging. That's why, in this section, we'll begin with starting the app, so you are aware of the full picture of remote debugging and know which are the prerequisites of such an approach.

When starting the app, if you want to be able to remotely debug it, you need to make sure you attach a debugger agent to the execution. To attach a debugger agent to a Java app execution, you use the `-agentlib:jdwp` parameter to the `java` command line as

presented in figure 4.5. When attaching the debugger agent, you specify a port number to which you'll need later to attach the debugger tool. Basically, the debug agent acts as a server, listening for a debugger tool to connect on the configured port, allowing the debugger tool to run the debug operations (pausing the execution on a breakpoint, stepping over, stepping into, and so on).

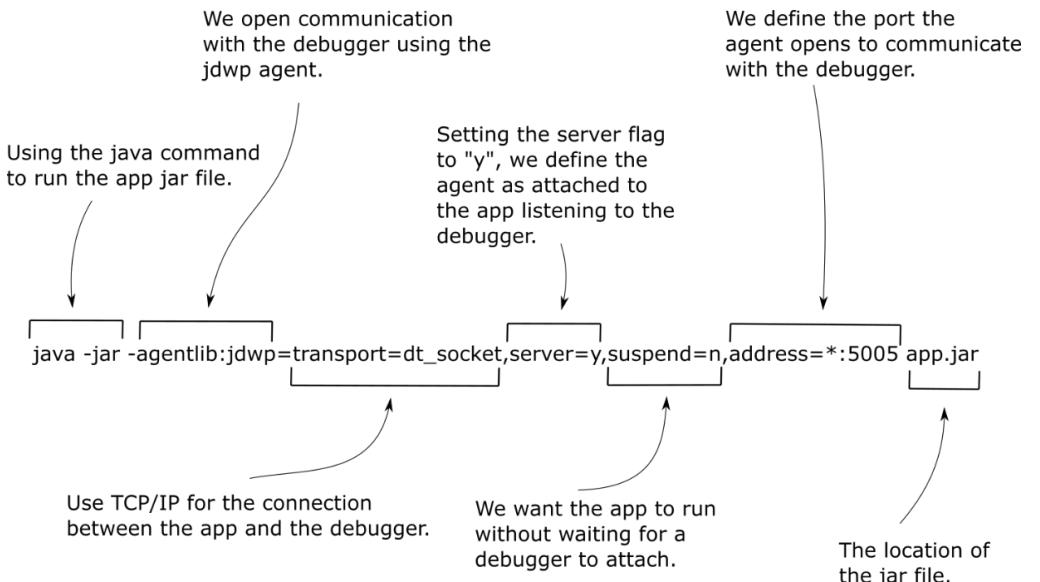


Figure 4.5 You have to attach a debugger agent to an app you want to debug. When debugging an app locally, the IDE attaches the debugger, and you don't have to worry about the debug agent. But when running an app in a remote environment, you need to attach a debugger agent at the app start yourself.

You can easily copy the command from the next snippet.

```
Java -jar -agentlib:jdwp=transport=dt_socket, server=y,suspend=n,address=*:5005 app.jar
```

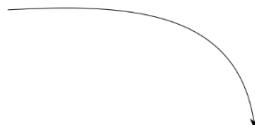
Observe the few configurations specified in the command:

- We use `transport=dt_socket` to configure the way the debugger tool communicates with the debugger agent. The `dt_socket` configuration means we use TCP/IP to establish the communication over a network. When remotely debugging, this is always the way you use for establishing the communication between the agent and the tool.
- We use `server=y`, which means the agent acts as a server after attaching to the app execution. The agent waits for a debugger tool to connect to it and control the app execution through it. You would use the `server=n` configuration to connect to a debugger agent rather than starting one.

- We use `suspend=n` to tell the app to start without waiting for a debugger tool to connect. If you want to prevent the app from starting until you connect a debugger to it, you need to use `suspend=y`. In our case, we have a web app, and the problem appears when calling one of its endpoints, so we need the app to start first to call the endpoint, that's why we use `suspend=n`. If we were investigating a problem with the server boot process, we would most likely need to use `suspend=y` to allow the app to start only after we have the debugger tool.
- The `address=*:5005` tells the agent to open port 5005 on the system. This is the port the debugger tool will connect to communicate with the agent. The port value must not be already in use on the system, and the network needs to permit the communication between the debugger tool and the agent (the port needs to be opened in the network).

Figure 4.6 shows the app starting with the debugger agent attached. Observe the message printed in the console right after the command tells us the agent listens to the configured port 5005.

You can use the command line to start the application. When you start the app you must make it available to connect a debugger to it on a given port.



```
$ java -jar -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005 da-ch4-ex1-0.0.1-SNAPSHOT.jar
Listening for transport dt_socket at address: 5005
.
.
.
:: Spring Boot ::
=====
2021-08-21 08:59:12.122  INFO 83884 --- [           main] com.example.Main                         : Starting Main v0.0.1-SNAPSHOT using Java 11.0.12 on EN1310832 with PID 83884 (C:\MANNINGS\Debugging Java Applications\CODE\spilca3\code\da-ch4-ex1\target\da-ch4-ex1-0.0.1-SNAPSHOT.jar started by lspilca in C:\MANNINGS\Debugging Java Applications\CODE\spilca3\code\da-ch4-ex1\target) .
```

Figure 4.6 When you run the command to start the app, you can see the app begins executing. At the same time, you can observe in the console the debugging agent printed a line showing that it listens for a debugger to attach on the configured port 5005.

Once your remote app has a debugger agent attached, you can connect the debugger to start investigating the issue. Remember, we assume here that the network is configured to allow communication between the two apps (the debugger tool and the debugger agent). We run both on the localhost for our example, so for our demonstration, such networking configurations are not an issue. But for a real-world example, you should always make sure you can establish the communication before starting to debug. In most cases, you'd likely need help from someone from the infrastructure team to help you open the needed port in

case the communication is not allowed. Remember that usually, ports are by default closed for communication for security considerations.

Further, we'll demonstrate how to attach the debugger to a remote app using IntelliJ IDEA Community. The steps to run the debugger on the app running in a remote environment are:

1. Add a new running configuration.
2. Configure the remote address (IP address and port) of the debugger agent.
3. Start debugging the app.

Figure 4.7 shows you how to open the **Edit Configurations** section to add a new running configuration.

To add a remote debugging configuration in IntelliJ, first, choose the **Edit Configurations** item from the menu.

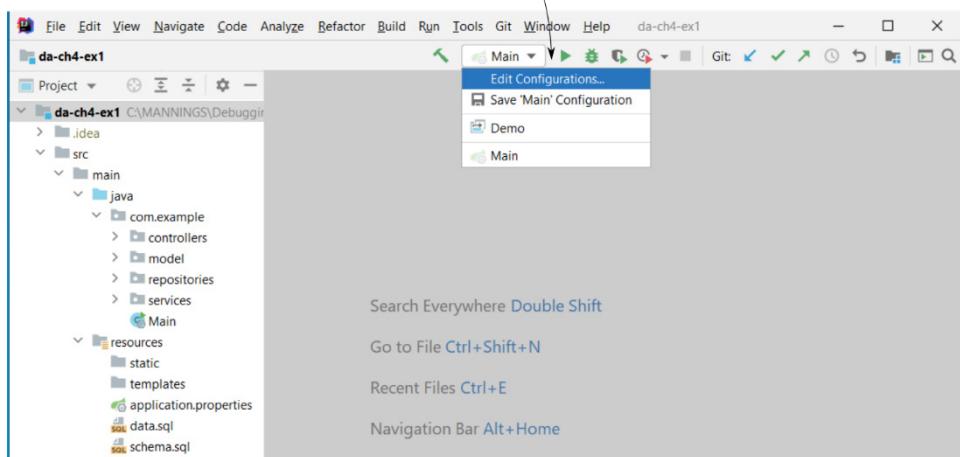


Figure 4.7 You can use an IDE to configure the debugger to attach to an already running app in a particular environment, as long as the app has a debugger agent attached to it. In IntelliJ IDEA Community, you need to create a new running configuration to tell the debugger to stick to an already running app. You can add a new run configuration by selecting the **Edit Configurations** as presented in the visual.

Figure 4.8 shows you how to add a new running configuration.

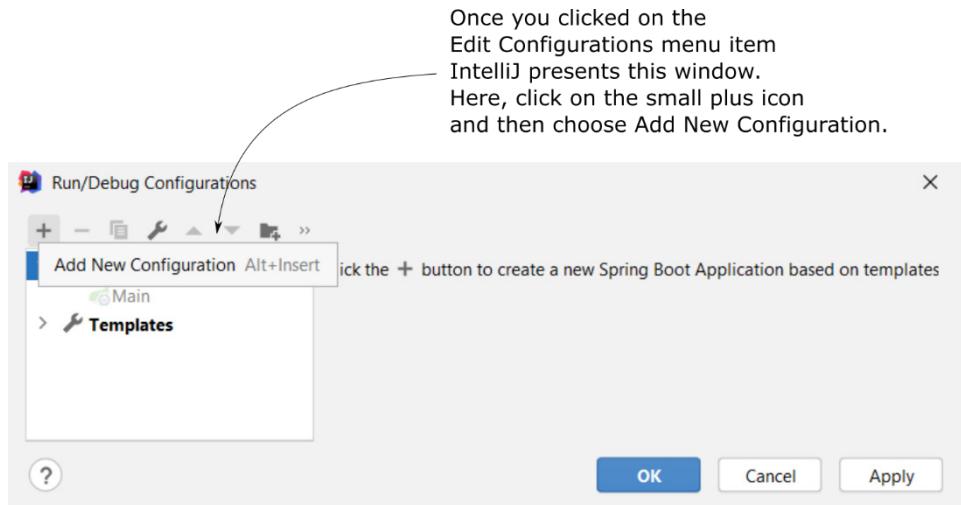


Figure 4.8 Once you selected Edit Configurations, you can add a new configuration. First, click on the plus icon and then Add New Configuration.

Since we want to connect to a remote debug agent, we need to add a new remote debugging configuration, as presented in figure 4.9.

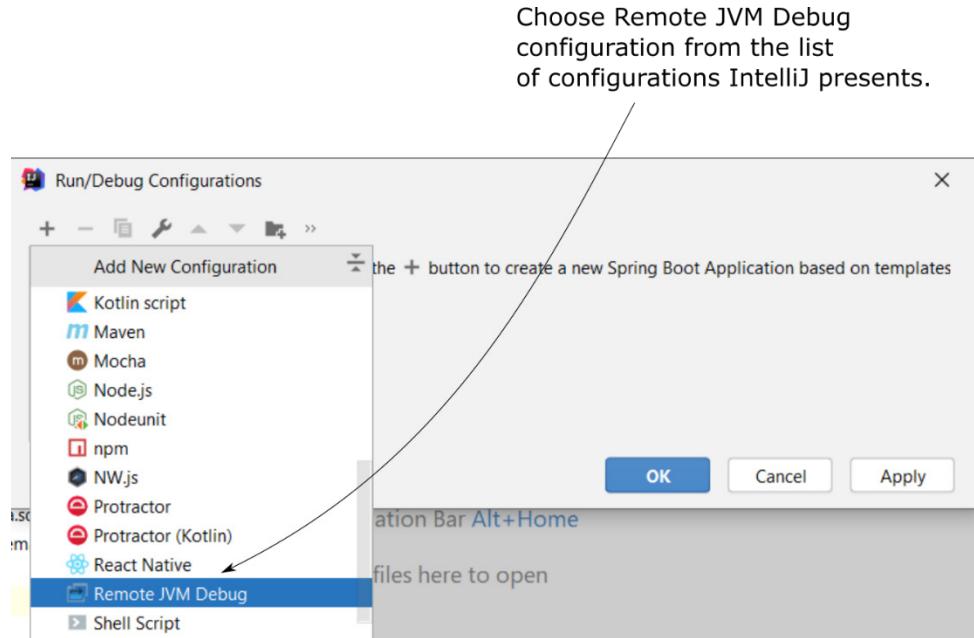


Figure 4.9 Since we want to attach to an app running in a remote environment, select the Remote JVM Debug configuration type.

Configure the address of the debugger agent as shown in figure 4.10. In our case, we run the app on the same system where we also have the debugger, so we use "localhost". Instead of "localhost", in a real-world example, where the app would run on a different system, you'd have to use the right IP address of that system. Port 5005 is the one we configured the agent to listen for connecting with a debugger tool.

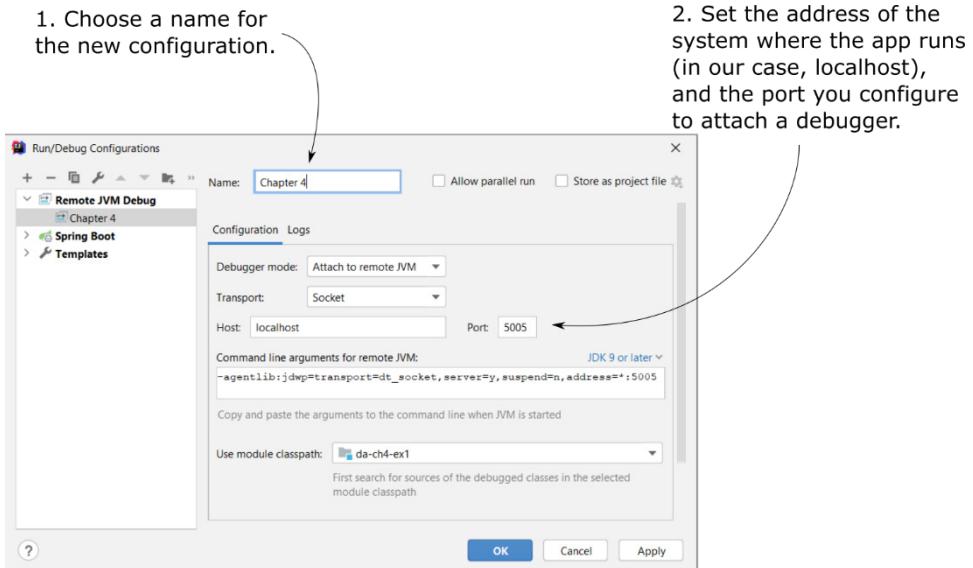


Figure 4.10 Give a name to the new configuration you add and specify the address of the environment and the port you configured the debugger agent to listen on. In our case, we configured port 5005 for the debugger agent when starting the app.

Remember that we connect the debugger tool to the debugger agent, which opens port 5005 (figure 4.11). Don't confuse the port opened by the debugger agent (5005) with our web app's port (8080).

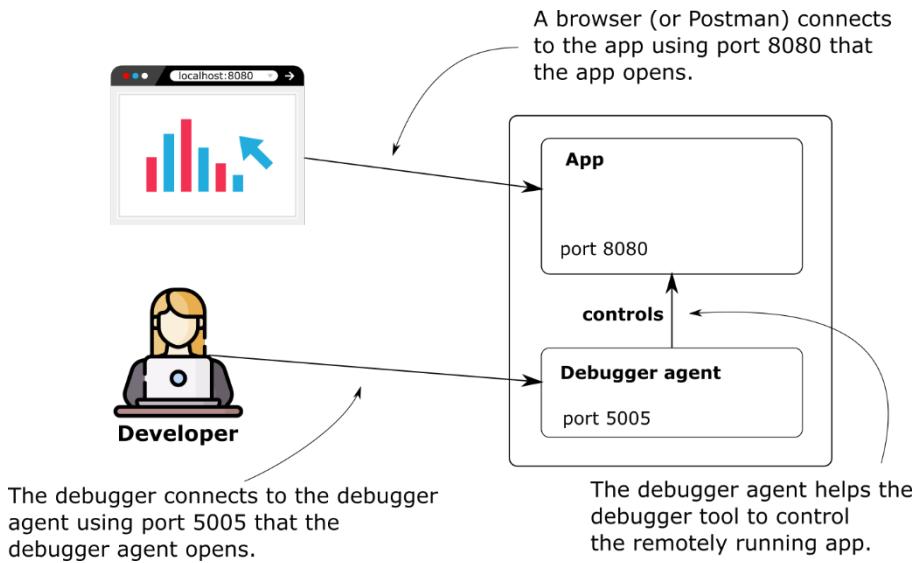


Figure 4.11 The debugger tool that runs on the developer's computer connects to the debugger agent on port 5005. The debugger agent helps the debugger tool control the app. The app also opens a port, but this port is for its clients (the browser in the case of a web app).

Once you have a configuration in place, start the debugger (figure 4.12). The debugger will start “talking” with the debugger agent attached to the app to allow you to control the execution.

Choose the newly added configuration,
and click on the debug button.

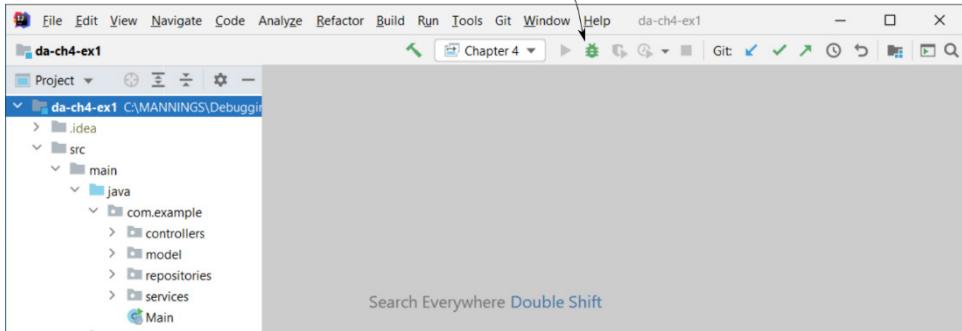


Figure 4.12 You can run the debugger now using the newly added configuration. Click on the small “bug” icon to start the debugger.

Now, you can use the debugger the same way you learned to use it in chapters 2 and 3. The most important thing you need to **be careful with is the version of the code you use** (figure 4.13). When locally debugging an application, you know that the IDE compiles the app and then attaches the debugger to the freshly compiled code. However, when you connect to a remote app, you can't be that sure anymore that the source code you have corresponds to the compiled code of the remote app to which you attach the debugger. The team started on new tasks, and potentially, code that you need to investigate was changed, added, or removed in the same classes involved in the issue you want to investigate. Using a different source code version could lead to strange and confusing debugger behavior. For example, the debugger might show you are navigating empty lines, even lines outside the methods or the classes. The execution stack trace could become inconsistent with the expected execution.

To correctly view the app execution, the developer needs to have the same source code version that generated the executable installed in the remote environment.

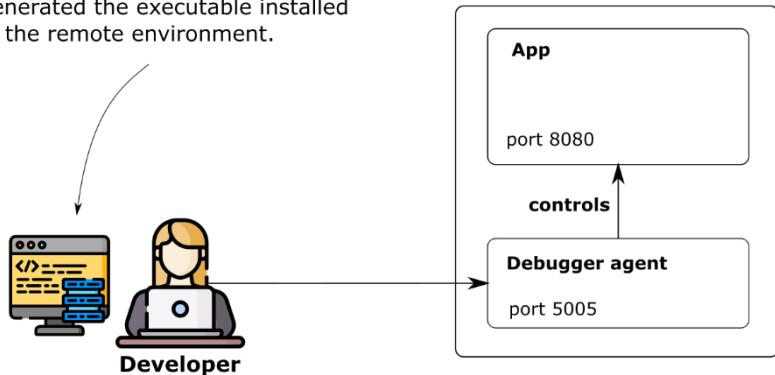
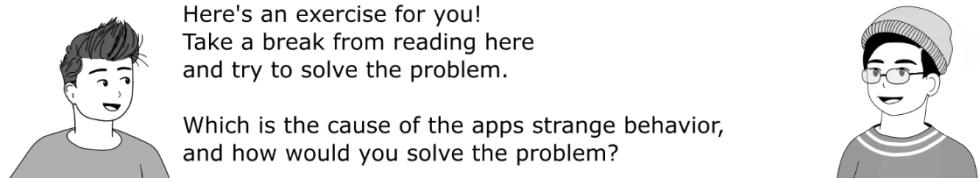


Figure 4.13 The developer needs to make sure they have the same version of source code as the one used to generate the app's executable running in the remote environment. Otherwise, the debugger's actions could become inconsistent with the code the developer investigates and would create more confusion than help the developer understand the app's behavior.

Fortunately, today, we use source code versioning software such as Git or SVN, and we can always find out which version of the source code created the app we deployed. Before debugging, you need to make sure you have the same source code as the one compiled into the app you want to investigate remotely. Use your source code versioning tool to check out the exact source code version.



We continue in the next pages with solving the problem.

Let's add a breakpoint on the first line that raises concerns. For us, this would be line 23 in the `ProductService` class, as presented in figure 4.14. Reading the code, I observe this is where the data that the app should return in the HTTP response is selected from the database. I would like first to understand if the data is correctly retrieved from the database, so I intend to pause the execution on this line and use step-over to see the result returned by the method.

After starting the debugger,
you can use it just as when
you were investigating a local app.
Add breakpoints and navigate through
the code.



The screenshot shows an IDE interface with the following details:

- Project:** da-ch4-ex1
- File:** ProductService.java
- Breakpoint:** A red circular icon with a white apple symbol is placed on line 23, which contains the code `var products : List<Product> = productRepository.findAll();`.
- Code Snippet (line 23):**

```
var products : List<Product> = productRepository.findAll();
```
- Code Snippet (line 24):**

```
var costs : Map<String, BigDecimal> = products.stream()
```
- Code Snippet (line 25):**

```
.collect(Collectors.toMap(
```
- Code Snippet (line 26):**

```
Product::getName,
```
- Code Snippet (line 27):**

```
p -> p.getPrice().multiply(new BigDecimal(p.getQuantity()))
```
- Code Snippet (line 28):**

```
response.setTotalCosts(costs);
```

Figure 4.14 Just as when debugging an app locally, you can add breakpoints and use navigation operations. Add a new breakpoint on line 23 in the ProductService class, as presented in this figure.

After adding the breakpoint, use Postman (or a similar tool) to send the HTTP request with unexpected behavior (figure 4.15). Postman (which you can download from the following link <https://www.postman.com/downloads/>) is a simple tool you can use to call a given endpoint and lately seems to have become one of the favorite tools developers use for this purpose. Postman has a user-friendly GUI, but if you prefer the command line, you can alternatively choose to use a tool such as cURL. To make the example simple and allow you to focus on the discussed subject, I'll use Postman with the book examples.

Observe Postman doesn't immediately show the HTTP response anymore. Instead, you observe the request remains pending. The cause of Postman's behavior is the fact that the debugger now paused the app on the line you marked with the breakpoint.

Using a tool, such as Postman, send a request to the endpoint. You observe that the request doesn't finish, and it remains in a pending state. The reason it stays pending is because the debugger paused the app on the line you marked with a breakpoint.

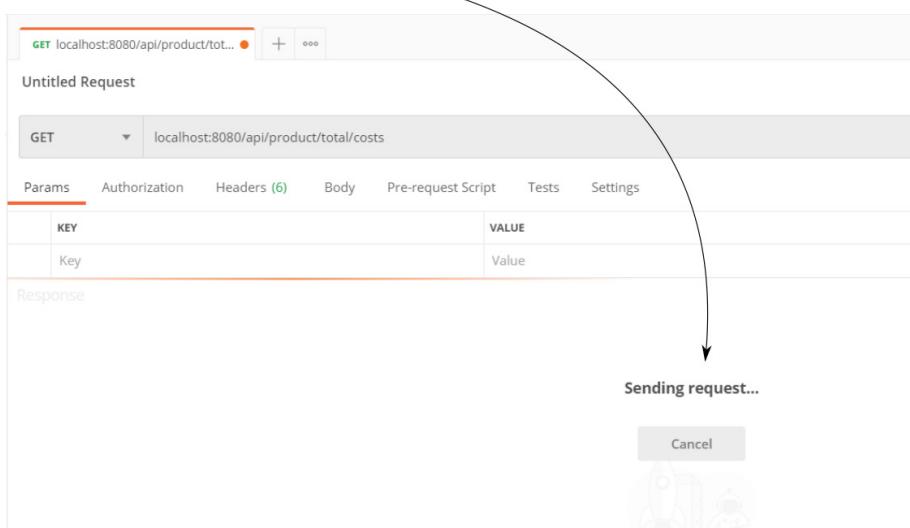


Figure 4.15 When using Postman to send the request, you observe the response doesn't come back immediately anymore. Instead, Postman seems to wait indefinitely for the response. The reason is: the app paused the execution on the line you marked with a breakpoint.

Back in the application (figure 4.16), you observe that the debugger indeed paused the execution on the line you marked with a breakpoint. You can start now using the navigation operations to investigate the problem.

Back in the IDE, you observe the execution paused on the line you marked with a breakpoint. You can now start navigating the code to investigate the issue.

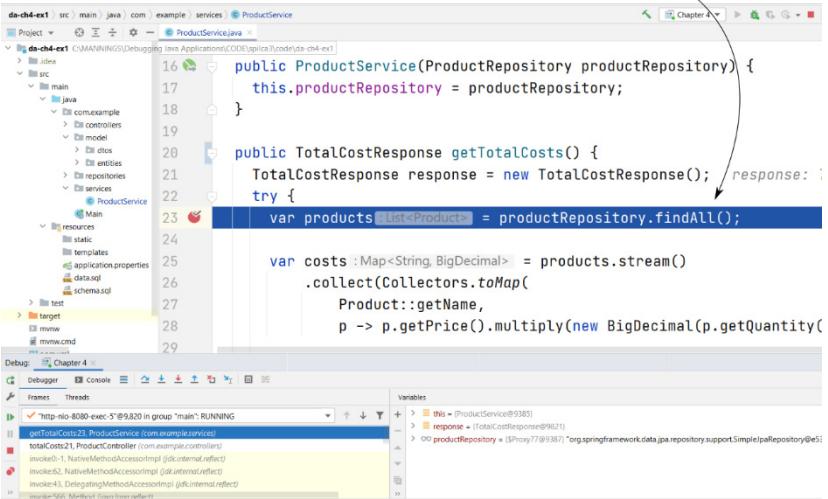


Figure 4.16 When you check the IDE, you observe the debugger indeed paused the execution on the line you marked with a breakpoint. Thus, you can use navigation operations to continue your investigation.

Using the step-over operation, you observe that instead of returning the data from the database, the app throws an exception (figure 4.17). Now you start understanding what the problem is.

1. First, the developer who implemented this functionality used a primitive type to represent a column that could take null values in a database. Since a primitive in Java is not an object type and cannot hold the value null, the app throws an exception.
2. Secondly, the developer used the `printStackTrace()` method to print the exception message. Using the `printStackTrace()` method is not helpful since you can't easily configure the output for various environments. You believe this is the reason why you couldn't see anything in the logs in the first place. We'll discuss more how to properly use logs with investigation techniques in chapter 5.
3. Also, the problem did not happen locally or in the DEV environment because there were no null values for that field in the database.

Clearly, the code needs to be refactored, and maybe an enhancement of the code review process should be discussed with the team in the next retrospective meeting. But you are happy that you found the cause of the issue and know how to solve it.

Using the step-over operation, you observe that the code throws an exception. Now you have a clue why the endpoint doesn't return the expected result.

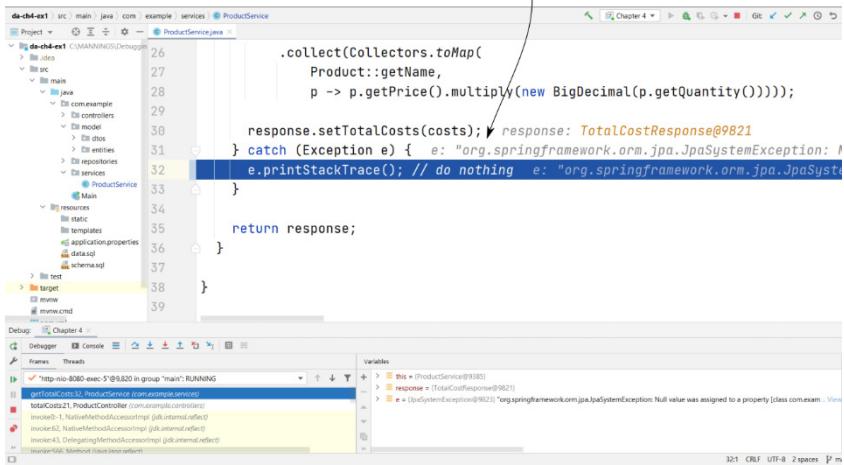


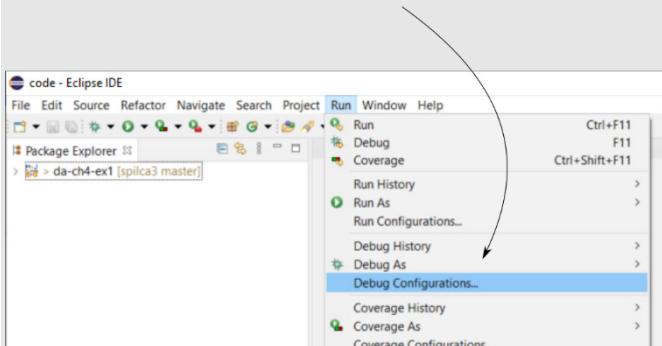
Figure 4.17 When using the step-over navigation operation, you observe the app throws an exception. Now you have an idea of what the problem is and decide further on how to solve it.

Creating a remote configuration in Eclipse IDE

I use IntelliJ IDEA as the primary IDE for the examples of the book. But as I stated in earlier chapters, this book isn't about using a certain IDE. You can apply the techniques we discuss in this book with a variety of tools of your choice. For example, you can do remote debugging with other IDEs, such as Eclipse. In this sidebar, we discuss adding a remote debugging configuration in Eclipse IDE.

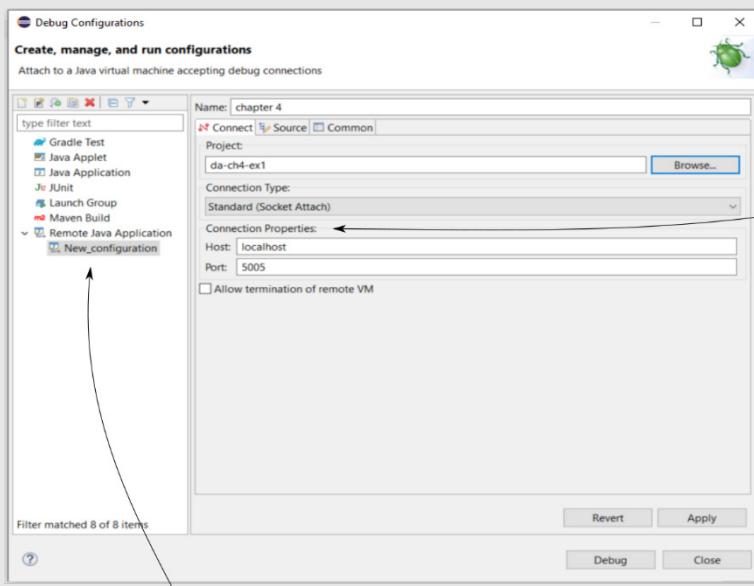
The next figure shows you how to add a new debugging configuration in Eclipse IDE.

To add a new debugging configuration in Eclipse IDE, choose Run > Debug Configurations



To add a new debug configuration in Eclipse IDE, choose Run > Debug Configurations. You can configure then the debug configuration to attach to the debugging agent controlling the remote app.

Same as in IntelliJ IDEA, you need to configure the debugging agent's address (IP address and port) to which the debugger tool connects.



From the left side of the window,
add a new Remote Java Application
configuration.

Add a new Remote Java Application debug configuration and set the address of the debugger agent. You can then save the configuration and use the debugging feature to connect remotely to the app for debugging.

Once you added the configuration, start the debugger and add the breakpoints to pause the execution where you want to start investigating your code.

4.3 Summary

- Sometimes, the specific unexpected behavior of a running app happens only in certain environments where the app executes. When this situation happens, debugging becomes more challenging.
- You can use a debugger with a Java app that executes in a remote environment with some conditions:
 - The app should be started with a debugger agent attached.
 - The network configuration should allow the communication between the debugger tool and the debugger agent attached to the app in the remote environment.
- Remote debugging allows you to use the same debugging techniques as local debugging and attaching to a process that runs in a remote environment.

- Before debugging an app running in a remote environment, make sure the debugger uses a copy of the same source code that created the app you investigate. Suppose you don't have the exact same source code, and changes were made meanwhile in the parts of the app involved in your investigation. In that case, the debugger might behave weirdly, and your remote investigation will become rather more challenging than helpful.

5

Making the most of logs: Auditing app's behavior

This chapter covers

- Effectively using logs to understand an app's behavior
- Correctly implementing log capabilities in your app
- Avoiding issues caused by logs

In this chapter, we'll discuss using log messages that an app records. The concept of logging didn't first appear with software. For centuries people used logs to help them understand past events and processes (figure 5.1). Logging helped many people since we invented writing, and it still helps us today.

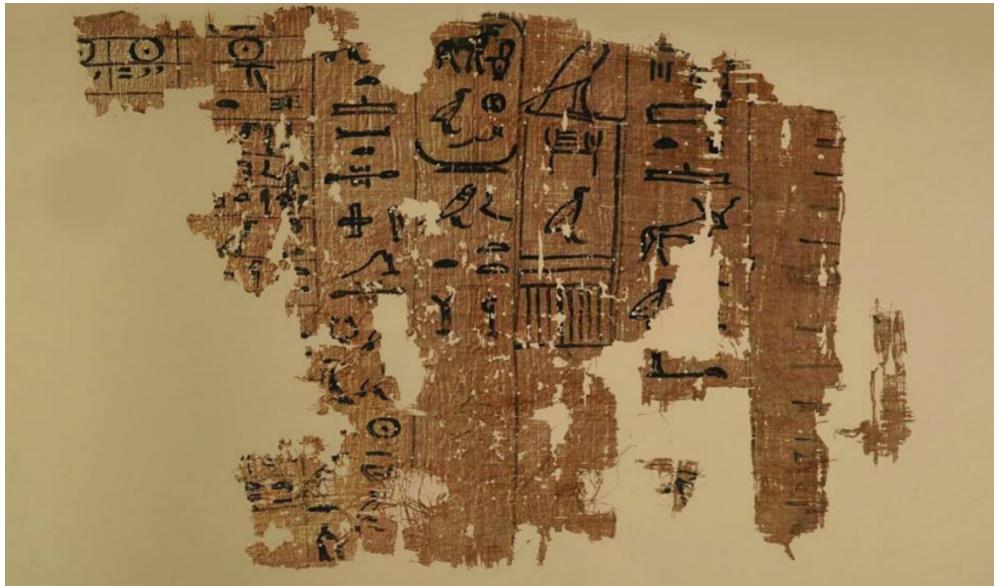


Figure 5.1 The oldest known logbook. It describes the process of building the Great Pyramid of Giza (Cairo, Egypt).

Any ship has a logbook. The sailors note down the decisions (direction, speed increase or decrease, and so on) and given or received orders, along with any encountered event in the logbook (figure 5.2). If something happens with the board equipment, they can use the logbook notes to understand where they are and navigate to the nearest shore. If an accident happens, the logbook notes can be used in the investigation to find out how the unfortunate event could have been avoided.

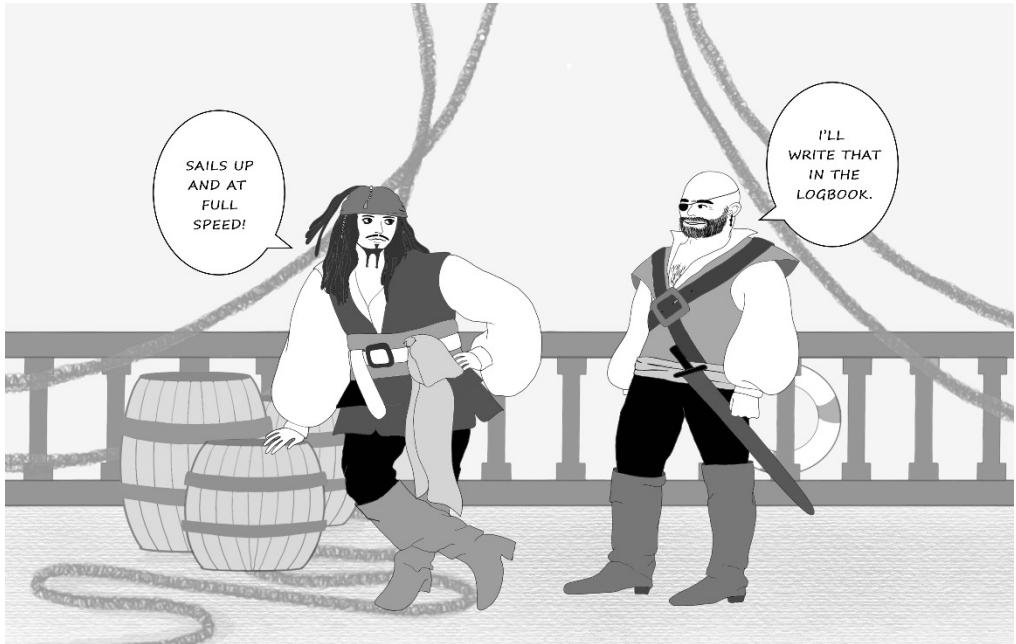


Figure 5.2 Sailors store events in logs which they can use later to figure out their route or analyze the crew response to a given event. In the same way, apps store log messages so developers can later analyze a potential issue or discover breaches in the app.

Also, if you ever watched a chess game, you'll observe that each player writes down each piece's movement. Why do chess players need to note that down every time? These logs help them recreate the entire game afterward. They study their moves, and their opponent's moves to spot potential mistakes they or their opponents made or their opponent's vulnerabilities.

For similar reasons, applications log messages, too. We use those messages to understand what happened during execution. Reading the log messages, you recreate the app's execution the same as a chess player recreates a whole game. We can use logs when we investigate a strange or unwanted behavior or spot more challenging to observe issues such as security vulnerabilities.

I'm sure you already know how logs look like. You've seen log messages, at least when running your app with an IDE (figure 5.3). Any IDE shows you the log console, which is one of the first things any software developer learns. But an app doesn't only display log messages in the IDE's console. Real-world apps store logs to allow the developer to investigate certain app behavior that happened at a given time.

When running an app on your local system using the IDE, you find the log messages in the console.

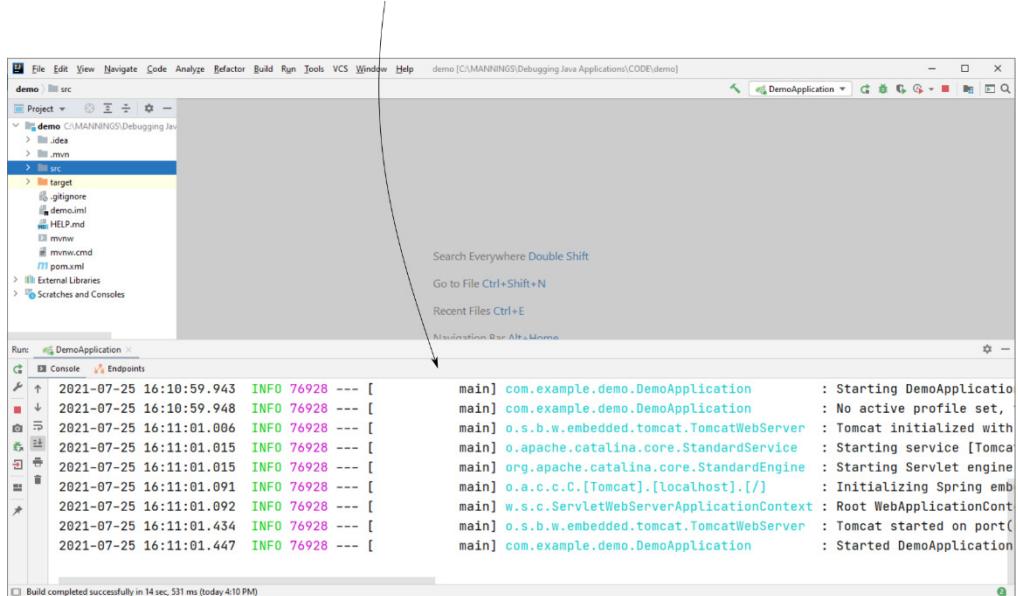


Figure 5.3 IDE log console. Any IDE shows a log console. While logging messages in the console is useful when running the app locally, real-world apps also store logs to use them if you need to understand how the app behaved at a given time.

Figure 5.4 shows the anatomy of a standard-formatted log message. A log message is just a string, so theoretically, it can be any sentence. However, clean and easy-to-use logs need to follow some best practices that you'll learn throughout this chapter. For example, besides a description, a log message contains the timestamp when the app wrote the message, the severity, and the part of the app that wrote the message (figure 5.4).

TIMESTAMP. **When** did the app write the message?

The timestamp shows when a message was logged.

The timestamp is a vital detail which allows us to chronologically order the messages. For this reason, the timestamp should always be at the beginning of the message.

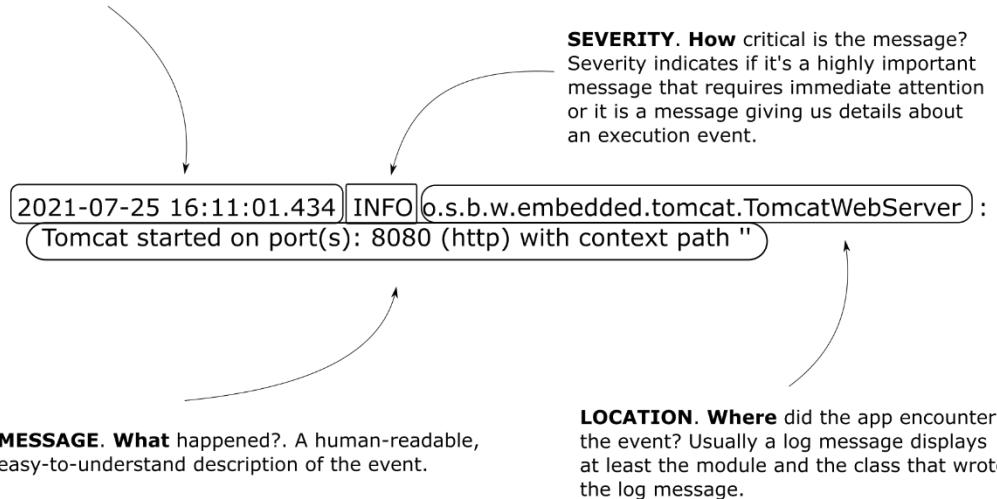


Figure 5.4 The anatomy of a well-formatted log message. Besides describing a situation or an event the app encountered, a log message should also contain several other relevant details. Any log message should contain at least the timestamp when the app logged the message, the event's severity, and where the message was written. These details help you more easily investigate a problem when you need to use the logs.

In many cases, logs are an efficient way to investigate an app's behavior. Some examples are

- Investigating an event or a timeline of events that already happened in an environment.
- Investigating issues where interfering with the app changes the app's behavior (Heisenbugs)
- Understanding the application's behavior in the long term
- Raising alarms for critical events that require immediate attention.

Besides such cases, as we've discussed, we generally don't use just one technique when investigating how a particular app capability behaves. Depending on the scenario it investigates, a developer will combine several techniques to understand a particular behavior. You'll find cases where you'll combine using the debugger with logs and eventually even other techniques (which you'll learn in the following chapters) to understand why something works the way it does.

I always recommend to developers that they **check the logs before doing anything else** when they investigate an issue (figure 5.5). In many cases, checking the app's logs messages helps you immediately observe some strange behavior that gives you a start for your investigation. The logs won't necessarily answer all the questions, but having a starting

point is extremely important. If the log messages show you directly where to start your investigation, you already saved plenty of time!

Before deciding which investigation technique to use, you should first read the log messages.

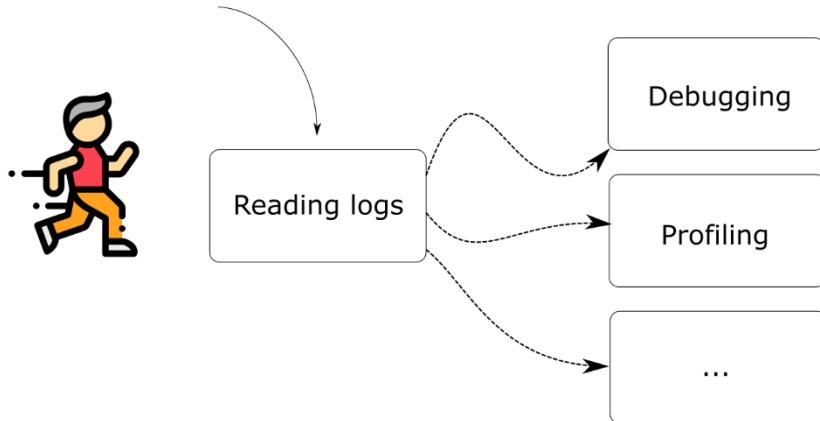


Figure 5.5 Whenever you investigate an issue, the first thing you do should always be reading the app's logs. In many cases, the log messages give you a starting point. Even if they can't always directly answer all the questions, they often give you valuable hints on what you should do next to solve the problem.

In my opinion, logs are not only extremely valuable - they are an indispensable tool for any application. In section 5.1, we discuss how to use logs and typical investigation scenarios where using logs is essential. In section 5.2, you'll learn how to implement logging capabilities in your app properly. We'll discuss implementing logging levels to help you easier filter events and issues caused by logs. In section 5.3, we discuss the differences between using logs and remote debugging.

Besides everything you find in this chapter, I also recommend reading part 4 of *Logging in Action* by Phil Wilkins (Manning, 2022). This chapter focuses more on investigation techniques with logs, while *Logging in Action* will dive more deeply into the logs' technicalities. You'll also find logging demonstrated using a different language than Java (Python). Many of the techniques we discuss in the book you're currently reading can be applied to other languages and technologies.

5.1 Investigating issues with logs

Like any other investigation technique, you'll find scenarios for which using logs is more relevant and scenarios where logs are less relevant to use in your investigations. Understanding the main cases where using logs will help you makes your decision easier when choosing which one to use. Knowing this will minimize the time you spend investigating a particular issue or behavior. In this section, we discuss various scenarios where using logs helps you understand software's behavior easier. We'll begin by discussing

several key points of log messages and then analyze why these characteristics help investigate various scenarios.

TIP Whenever you have an issue, first check the logs. The logs don't necessarily give you a complete answer, but they often provide clues for solving a puzzle or at least give you some starting point for your investigation.

One of the most important advantages of log messages is that they allow you to visualize the execution of a certain piece of code at a given time. When you use a debugger, as we discussed in chapters 2 through 4, your attention is mainly on the present. You look at how the data looks while the debugger pauses the execution on a given line of code. A debugger doesn't give you many details on the execution history. You can use the execution stack trace to identify the execution path, but everything else is focused on the present.

However, logs are a tool that focuses on the app's execution over a past period (figure 5.6). Log messages have a strong relationship with time.



NOTE Always include the timestamp in a log message. You'll use the timestamp to easily identify the order in which messages were logged and give you an idea of when the app wrote a certain message. I recommend the timestamp be in the first part (at the beginning) of the message.

Remember to consider the timezone of the system where your app is running. One thing that could be confusing (especially for a beginner) is the fact that the logged time would appear as being shifted with a few hours because of the difference of the time zone between where the app runs and where the developer is.

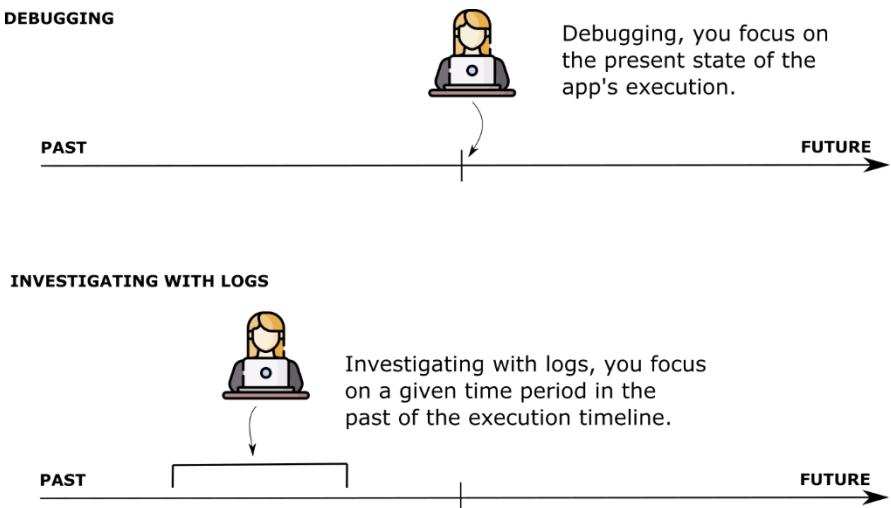


Figure 5.6 When investigating an issue with the debugger, you focus on the present. When you do the same thing with the log messages, you focus on a given period in the past. This difference is one of the details you'll consider when deciding whether to use logs or the debugger.

5.1.1 Using logs to identify exceptions

One of the most used characteristics of logs is that they help you identify a problem after it occurred and help you investigate its root cause. Often, we use logs to know where to start an investigation. We then continue exploring the problem using other tools and techniques, such as the debugger (as discussed in chapters 2 through 4) or a profiler (as discussed in chapters 6 through 9). Finding exception stack traces in the logs is a very often encountered scenario. The next snippet shows an example of a Java exception stack trace.

```
java.lang.NullPointerException
at java.base/java.util.concurrent.ThreadPoolExecutor
↳ runWorker(ThreadPoolExecutor.java:1128) ~[na:na]
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker
↳ run(ThreadPoolExecutor.java:628) ~[na:na]
at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable
↳ run(TaskThread.java:61) ~[tomcat-embed-core-9.0.26.jar:9.0.26]
at java.base/java.lang.Thread.run(Thread.java:830) ~[na:na]
```

Seeing something like this in the application's log tells you something potentially went wrong with a given feature. Each exception has its own meaning, and it helps you identify where the app encountered a problem. For example, a `NullPointerException` tells you that somehow an instruction referred to an attribute or a method through a variable that didn't contain a reference to an object instance (figure 5.7).

If the app throws a `NullPointerException` on this line, it means that the `invoice` variable doesn't hold an object reference. In other words, the `invoice` variable is null.

```
var invoice = getLastIssuedInvoice();

if (client.isOverdue()) {
    invoice.pay(); ←
}
```

Figure 5.7 A `NullPointerException` indicates the app execution encountered a situation where a behavior had to be called without the behaving instance. But it doesn't mean that the line that produced the exception is also the cause of the problem. The exception could only be a consequence of the root cause. You should always look for the root cause instead of locally treating a problem.

NOTE You should remember that the place where an exception occurred is not necessarily the root cause of the problem. An exception tells you where something went wrong. But the exception itself can be a consequence of a problem. An exception is not necessarily the problem itself. Don't decide quickly on solving the exception locally by adding a try-catch-finally block or an if-else statement. First, make sure you understand the root cause of the problem and find a solution to solve the root cause.

I often find beginners being confused by this aspect. Let's take a simple `NullPointerException`. A `NullPointerException` is probably the first exception any Java developer encounters and one of the most simple to understand. However, when you find a `NullPointerException` in the logs, you need first to ask yourself: why is that reference missing? It could be missing because a particular instruction that the app executed earlier didn't work as expected (figure 5.8).

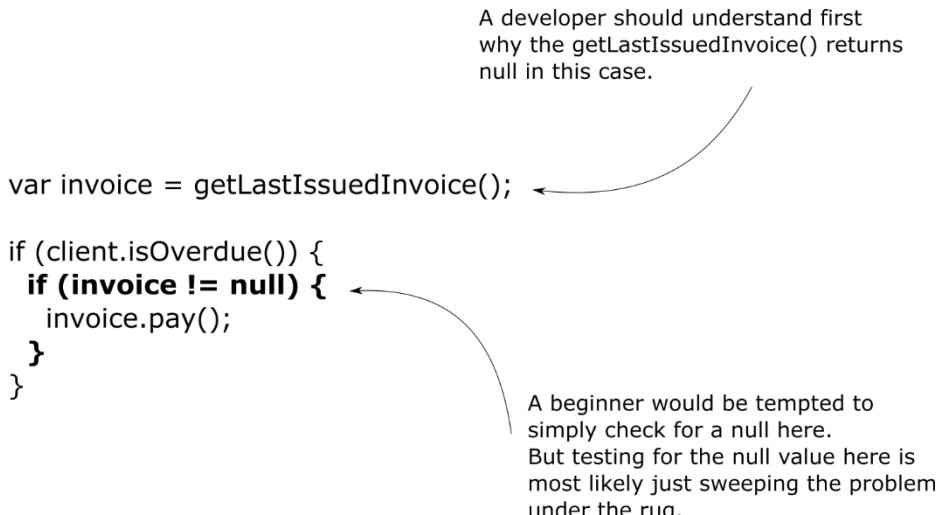


Figure 5.8 Locally solving the problem is in many cases equivalent to sweeping it under the rug. Without solving the root cause, more issues could appear later. Always look for the root cause and remember that an exception you find in the logs doesn't necessarily indicate the root cause.

5.1.2 Using exception stack traces to identify who calls a method

One of the techniques developers consider unusual, but I find advantageous in practice, is logging an exception stack trace to identify who calls a specific method. From when I started my software developer career, I've worked a lot with messy codebases of (usually) large applications. One of the difficulties I encountered was finding out who calls a given method when an app was running in a remote environment. If you tried finding out how that method gets called just by reading the app's code, you would have discovered hundreds of ways that method could've been called.

Of course, you can be lucky enough and have access to remote debug the app, as discussed in chapter 4. You would then have access to the execution stack trace the debugger provides. But what if you can't use the debugger remotely? We can use a logging technique instead!

Exceptions in Java have a capability that is often disregarded: they keep track of the execution stack trace. When discussing exceptions, we often call the execution stack trace an "exception" stack trace. But it's, in the end, the same thing. The exception stack trace shows you the chain of method calls that caused a specific exception, and you have access to this information even without throwing that exception. In code, it's enough to use the exception as presented in the next code snippet:

```
new Exception().printStackTrace();
```

Consider a method as the one presented in listing 5.1. If you don't have a debugger, you can simply print the exception stack trace as I did in this example as the first line in the method to find out the execution stack trace. Mind that this instruction only prints the stack trace,

and doesn't throw the exception, so it doesn't interfere anyhow with the executed logic. You find this example in project da-ch5-ex1.

Listing 5.1 Printing the execution stack trace in logs using an exception

```
public List<Integer> extractDigits() {
    new Exception().printStackTrace();      #A
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < input.length(); i++) {
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }
    return list;
}
```

#A Printing the exception stack trace.

The next snippet shows how the app prints in the console the exception stack trace. In a real-world investigation scenario, the stack trace helps you immediately identify the execution flow that leads to a call you investigate, as we discussed in chapters 2 and 3. In this example, you can easily observe from the logs that the `extractDigits()` method was called on line 11 of the `Decoder` class from within the `decode()` method.

```
java.lang.Exception at
    main.StringDigitExtractor.extractDigits(StringDigitExtractor.java:15)
    at main.Decoder.decode(Decoder.java:11)
    at main.Main.main(Main.java:9)
```

5.1.3 Measuring time spent to execute a given instruction

Log messages are an easy way for measuring the time a given set of instructions took to execute. You can always log the difference between the timestamp before and after a given line of code. Suppose you investigate a performance issue where some given capability takes too long to execute. You suspect the cause is a query the app executes to retrieve data from the database. For some parameter values, the query is slow and drops the overall app's performance.

To find out which parameter causes the problem, you can write in logs the query and the time the app needed to execute the query. Once you find out which parameter values cause the problem, you can start looking for a solution. Maybe we need to add one more index on a table in the database, or perhaps you can re-write the query to make it faster.

Listing 5.2 shows you how to log the time spent by the execution of a specific piece of code. For example, let's find out how much time it takes the app to run the operation of finding all the products from the database. Yes, I know, we have no parameters here – I simplified the example to allow you focus on the discussed syntax. But in a real world app, you'd most likely investigate a more complex operation.

Listing 5.2 Logging the execution time for a certain line of code

```
public TotalCostResponse getTotalCosts() {
    TotalCostResponse response = new TotalCostResponse();

    long timeBefore = System.currentTimeMillis();      #A
    var products = productRepository.findAll();      #B
    long spentTimeInMillis =           #C
        System.currentTimeMillis() - timeBefore;

    log.info("Execution time: " + spentTimeInMillis);   #D

    var costs = products.stream().collect(
        Collectors.toMap(
            Product::getName,
            p -> p.getPrice()
                .multiply(new BigDecimal(p.getQuantity()))));
}

response.setTotalCosts(costs);

return response;
}
```

#A Logging the timestamp before the method's execution.

#B Executing the method for which we want to calculate the execution time.

#C Calculating the time spend making the difference between the timestamp after execution and the timestamp before the execution.

#D Printing the execution time.

This technique is simple but effective when you precisely measure how much time the app spent executing that given instruction. However, I would only use this technique for temporarily investigating an issue. I don't recommend you let such logs in the code for a long time since they'd most likely not be needed later, and they make the code more difficult to read. ***Once you've solved the problem and don't need to know the execution time for that line of code, you can remove the logs.***

In chapters 6 through 9, we'll discuss sampling and profiling. You'll also learn that you can measure the execution time of given instructions using a profiler tool. But, as you'll find out in chapter 6, the profiling tool interferes with the app's execution which usually leads to imprecise results. When we get to chapter 6, we'll discuss a comparison between the approach of logging the execution time and using the profiler to identify code parts that take a long time to execute.

5.1.4 Investigating issues in multithreaded architectures

Multithreaded architectures are a particularly sensitive type of capability in an application. A capability that uses multiple threads to define its functionality, we say it has a multithreaded architecture (figure 5.9).

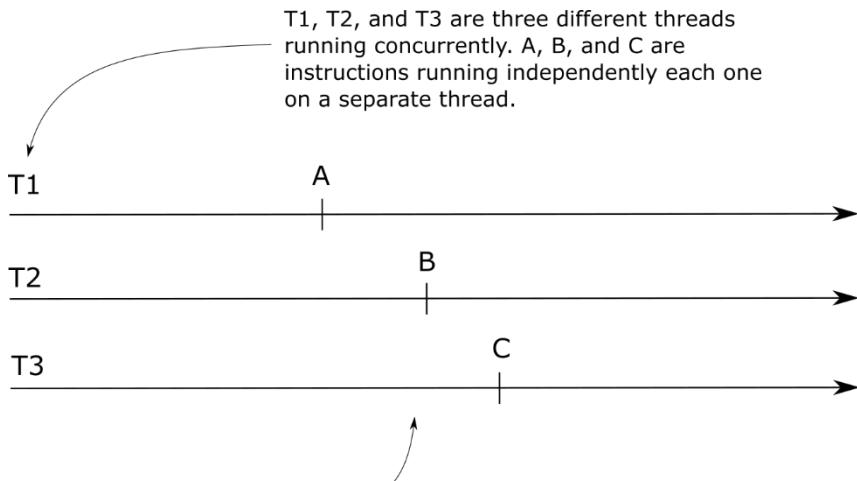


Figure 5.9 A multithreaded architecture. An app capability using multiple threads running concurrently to process data is a multithreaded app. Unless explicitly synchronized, instructions running on independent threads (A, B, and C) can run in any order. Suppose you want to investigate a certain behavior that only happens when these three instructions execute in the exact order A, B, C.

Such capabilities are usually sensitive to external interference. If you, for example, use a debugger or a profiler, which are tools that interfere with the app's execution, the app's behavior might change (figure 5.10).

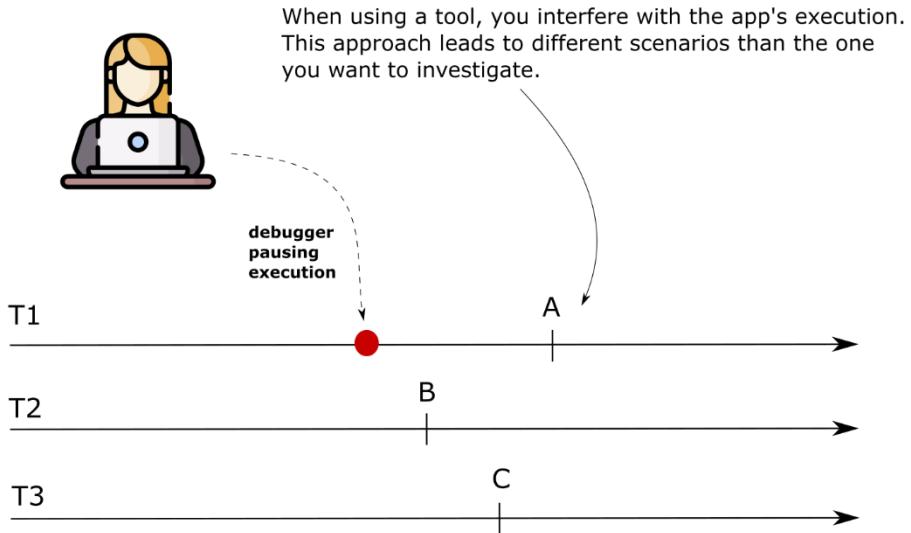


Figure 5.10 When using a tool such as a debugger or a profiler, the tool interferes with the execution, making some (or all) threads slower. Because of this, the execution often changes, and some instructions may execute in a different order than the scenario you wanted to investigate. In such a case, using a tool is no longer useful since you can't research the behavior you wanted at the beginning.

Using logs, however, there's a smaller chance the app will be affected while running. Indeed, even logs can sometimes interfere in such apps, but they don't have a big enough impact on the execution to change the app's flow in most cases. So, logs can be a solution for retrieving the needed data for your investigation in such cases.

Since logs messages contain the timestamp (at least recommended as discussed earlier in the chapter), you can order the log messages to find out in which order the operations are executed. In a Java app, sometimes you'll find it helpful to log the thread's name executing a certain instruction. You can get the name of the current thread in execution using the following instruction:

```
String threadName = Thread.currentThread().getName();
```

In a Java app, all threads have a name. The developer can name them, or the JVM will identify the threads using a name with the pattern Thread-x, where x is an incremented number. For example, the first thread created will be named Thread-0, the next one Thread-1, and so on. As we'll discuss in chapter 10, when we'll deal with thread dumps, a good practice is naming your app's threads so you can identify them easier when investigating a case.

5.2 Implementing logging

In this section, we discuss best practices for implementing logging capabilities in apps. To make your app's log messages efficient for investigations and avoid causing trouble for the app's execution, you need to take care of some implementation details.

We'll start discussing how apps persist logs in section 5.2.1 with the advantages and disadvantages of these practices. In section 5.2.2, you'll learn how to classify the log messages on severities to make the app more performant in execution and use the log messages more efficiently. In section 5.2.3, I'll tell you what problems log messages could cause and how to avoid them. Even if logging seems harmless, it could affect your app if not properly implemented.

5.2.1 Persisting logs

Persistence is one of the essential characteristics of log messages. As discussed earlier in this chapter, logging is different from other investigation techniques because it focuses more on the past than the present. We read logs to understand something that happened, so the app needs to store them to read them later. Depending on how the app stores the log messages, its performance and the usability of the logs can be affected. Working with many apps, I had the chance to see various ways in which developers implemented the log messages persistence:

- Storing logs in a non-relational database
- Storing logs in files
- Storing logs in a relational database

All of these can be good choices depending on what your app does. Let's enumerate some of the main things you need to consider to make the right decision.

STORING LOGS IN NON-RELATIONAL (NoSQL) DATABASES

Non-relational databases help you make the compromise for performance over consistency. You can use a non-relational database to store logs in a more performant way, allowing the database the chance to miss log messages or not store them in the exact chronological order in which the app wrote them. But, as we discussed earlier in this chapter, a log message should always contain the timestamp when the message was stored, preferably at the beginning of the message.

Storing log messages in non-relational databases is quite often today. In most cases, apps use a complete engine that provides storing the logs and capabilities to retrieve, search, and analyze the log messages. Today's two most used engines today are the ELK stack (<https://www.elastic.co/what-is/elk-stack>) and Splunk (<https://www.splunk.com/>).

STORING LOGS IN FILES

Older apps used to store logs in files. While out there, you might still find such applications that write their log messages directly in files, this approach is not so common anymore today. Writing the messages in files is generally slower and searching for the logged data is also more difficult. However, I want to make you aware of these aspects because you'll most likely find many tutorials and examples where apps could store their logs in files. Usually,

authors use this approach because it's simpler and allows the student to focus on the discussed subject. But don't be puzzled by such implementations. In today's real-world apps, you should avoid writing logs in files.

STORING LOGS IN A RELATIONAL DATABASE

We rarely use relational databases to store log messages. Mainly, a relational database guarantees data consistency. Consistency ensures you that you wouldn't lose any log messages. Once stored, you're sure you can retrieve them. But consistency comes with a compromise in performance.

In most apps, losing a log message is not a big deal, and you'll prefer performance over consistency. But, as always, in real-world apps, you find exceptions. For example, governments worldwide impose regulations about log messages for financial apps, especially in capabilities related to payments. Such capabilities should generally have specific log messages that the app isn't allowed to lose. Failure to comply with such regulations is drastically sanctioned.

5.2.2 Defining logging levels and using logging frameworks

In this section, we discuss logging levels and properly implementing logging in an app using logging frameworks. We'll start by understanding why logging levels are essential, and, after, by implementing an example, you'll learn how to use a logging framework and observe why they are useful.

Logging levels, also named "severities" are a way to classify the log messages over the probability of needing them in your investigation. An app usually produces a large number of log messages while running. However, often when investigating a problem, you don't need all the details in all the log messages. Some of the log messages are more important than others. By "more important" I mean here that they either represent critical events that always require attention or it's more likely you'll need them.

The most common log levels (severities) are:

1. **Error** – Describing a critical issue. The app should always log such events. Usually, unhandled exceptions in Java apps are logged as a severity-level error.
2. **Warn** – Describing an event that potentially is an error, but the application managed to handle it. For example, if a connection to a third-party system fails initially but the app manages to send the call on a second retry, the problem should be logged as a warning.
3. **Info** – The "common" log messages. These represent the main app execution events that help you understand the app's behavior in most situations.
4. **Debug** – Fine-grained details that we should enable only when info messages are not enough.



Note that different libraries may use more than these four severity levels or can use different names for them. For example, in some cases, above the ones I mentioned, you might find apps or frameworks also using the severity levels Fatal (more critical than Error) and Trace (less critical than Debug). In this chapter, I'll focus only on the most encountered severities and terminologies in real-world apps.

Classifying the log messages based on severities allows you to minimize the number of log messages your app stores. You only allow your app to log the most relevant details and enable more logging only when you investigate a case for which you need more details.

Take a look at figure 5.11, which presents the log severities pyramid. The main points this visual tells about logs are:

- An app logs a small number of critical issues, but these have high importance, so they always need to be logged.
- The more you go to the bottom of the pyramid, the more log messages the app writes, but they become less critical and less frequently needed in investigations.

For most investigation cases, you won't need the messages classified as debug. Also, being so many, you'll realize that, on the contrary, in most cases, they make your research less comfortable. For this reason, these messages are generally disabled, and you enable them only if you really face a scenario where you need more details.

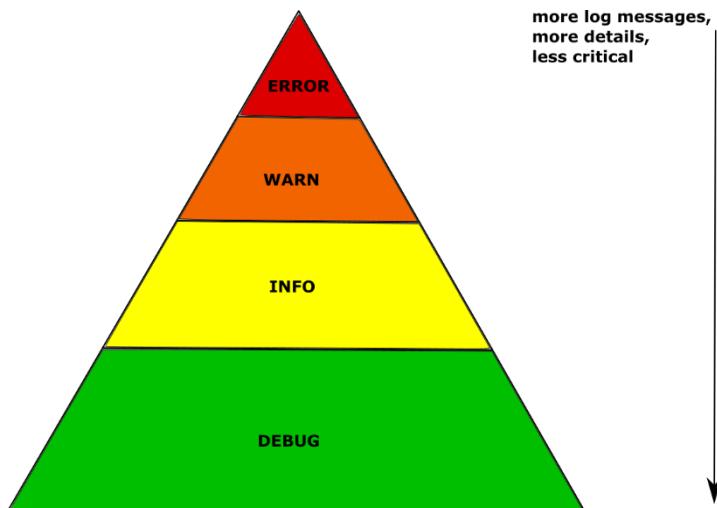


Figure 5.11 The log severity pyramid. On the top are the critical log messages that usually require immediate attention. The bottom represents the log messages that only represent details you'll rarely need. From the top to the bottom, the log messages become less essential but bigger in number. Usually, the debug level messages are disabled by default, and the developer chooses to enable them only when facing a hard investigation case where they need fine-grained details about the app's execution.

When you started learning Java, you were taught how to print something in the console using `System.out` or `System.err`. Eventually, you learned to use `printStackTrace()` to log an exception message, as I also used in section 5.1.2. But these ways to work with logs in Java apps don't give enough flexibility for configuration, so instead of using them in real-world apps, I recommend you go with a logging framework.

Implementing the logging levels is simple. Today, the Java ecosystem offers various logging framework possibilities such as Logback, Log4J, or the Java Logging API. These frameworks are similar to one another, and using them is straightforward.

Let's take an example and implement logging with Log4J. You find this example in project `da-ch5-ex2`. To implement the logging capabilities with Log4J, we first need to add the Log4J dependency to our project. In our Maven project, you need to change the `pom.xml` adding the Log4J dependency as shown in listing 5.3.

Listing 5.3 Dependencies you need to add in the pom.xml file to use Log4J

```
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.14.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.14.1</version>
    </dependency>
</dependencies>
```

Once you have the dependency in the project, you can declare a `Logger` instance in any class where you want to write log messages. With Log4J, the simplest way to create a logger instance is using the `LogManager.getLogger()` method as presented in listing 5.4. With the logger instance, you can write log messages using the logger's methods that are named the same as the severity of the event they represent. For example, if you want to log a message with the `INFO` severity, you'll use the `info()` method. If you want to log a message with the `DEBUG` severity, you'll use the `debug()` method, and so on.

Listing 5.4 Writing the log messages with different severities

```

public class StringDigitExtractor {

    private static Logger log = LogManager.getLogger();      #A

    private final String input;

    public StringDigitExtractor(String input) {
        this.input = input;
    }

    public List<Integer> extractDigits() {
        log.info("Extracting digits for input {}", input);    #B
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < input.length(); i++) {
            log.debug("Parsing character {} of input {}",      #C
                      input.charAt(i), input);
            if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
                list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
            }
        }
        log.info("Extract digits result for input {} is {}", input, list);
        return list;
    }
}

```

#A Declaring a logger instance for the current class to write log messages.

#B Writing a message with the info severity

#C Writing a message with the debug severity

Once you decided which messages to log and used the Logger instance to write these messages, you need to configure Log4J to tell how and where to write these messages. We'll use an XML file that we name log4j2.xml to configure Log4J. This XML file has to be in the app's classpath, so in our case, we'll add it to the resources folder of our Maven project. We need to define three things (figure 5.12):

1. A *logger* – tells Log4J which messages are to be written to which appender.
2. An *appender* – tells Log4J where to write the log messages.
3. A *formatter* – tells Log4J how to print the messages.

The logger defines which messages the app logs. In this example, we use "Root" to write the messages from any part of the app. Its attribute "level" that has the value "info" means only the messages with a severity of INFO and higher are logged. The logger can decide to only log messages from specific app parts, not all of them. For example, when using a framework, you are rarely interested in the log messages the framework prints, but you are often interested in your app's log messages. So you'll define a logger that excludes the framework's log messages and only prints those coming from your app. Remember that your purpose is to write only the essential log messages, otherwise an investigation would be more challenging since you would need to filter yourself the log messages that are not essential.

In a real-world app, you can define multiple appenders, which will most likely be configured to store the messages in different sources like a database, or files in the file system. Earlier, in section 5.2.1 we discussed multiple ways in which apps can persist the log messages. Appenders are just implementation which take care for storing the log messages in a given way.

The appender also uses a formatter that defines the format of the message. For this example, the formatter specifies the messages should include the timestamp and the criticality, so the app only needs to send the description.

The logger decides which log messages are printed. For example, a logger can decide to only log the messages with severity info and above coming from a specific package in the app.

An appender decides where to log the messages. For example, an appender can write the messages in the system console, and another can write them in a database.

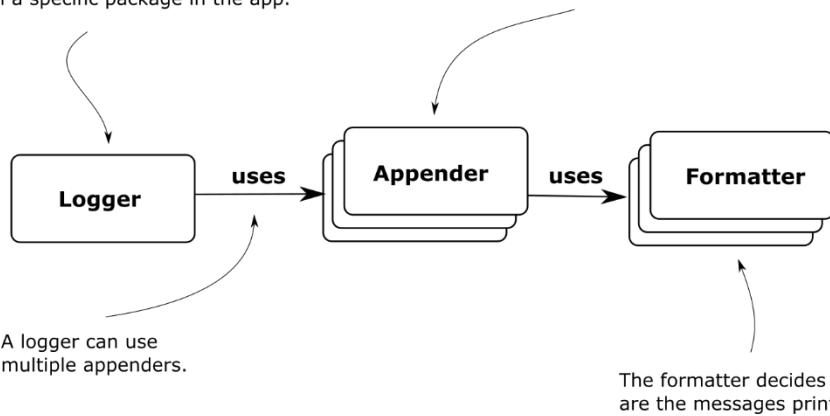


Figure 5.12 The relationship between the appender, logger, and formatter. A logger uses one or more appenders. The logger decides what to write (for example, only log messages printed by objects in the package com.example). The logger gives the messages to be written to one or more appenders. Each appender implements a certain way to store the messages. The appender uses formatters to shape the messages before storing them.

Listing 5.5 shows the configuration with both the definition of an appender and a logger. In this example, we only define one appender, which tells Log4J to log the messages in the standard output stream of the system (the console).

Listing 5.5 Configuring the appender and the logger in the log4j2.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>      #A
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{yy-MM-dd HH:mm:ss.SSS} [%t]
             %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>      #B
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

#A Defining an appender.

#B Defining a logger configuration.

Figure 5.13 visually shows the link between the XML configuration in listing 5.5 and the three components it defines: the logger, appender, and formatter.

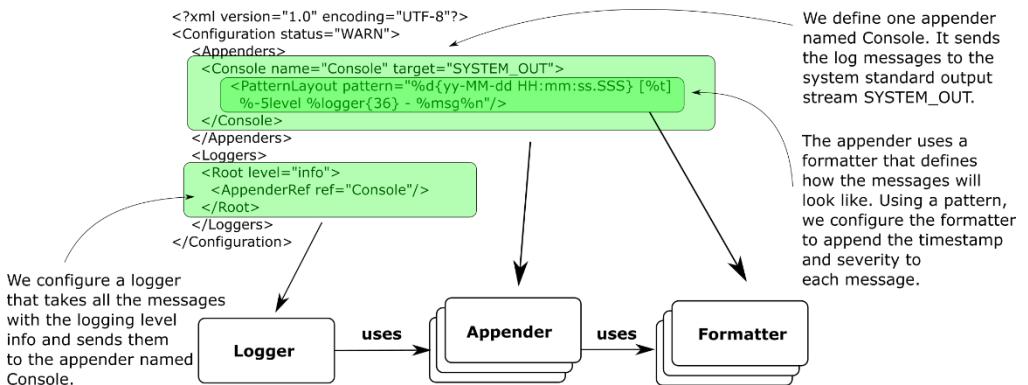


Figure 5.13 The components in configuration. The logger Root takes all the log messages with severity info the app writes. The logger sends the messages to the appender named Console. The appender Console is configured to send the messages to the system terminal. It uses a formatter to attach the timestamp and the severity to the message before writing them.

The next snippet shows a part of the logs printed when running the example. Observe that the messages with the DEBUG severity aren't logged since they are lower in severity than info (line 10 in listing 5.5).

```

21-07-28 13:17:39.915 [main] INFO main.StringDigitExtractor - Extracting digits for input
    ab1c
21-07-28 13:17:39.932 [main] INFO main.StringDigitExtractor - Extract digits result for
    input ab1c is [1]
21-07-28 13:17:39.943 [main] INFO main.StringDigitExtractor - Extracting digits for input
    a112c
21-07-28 13:17:39.944 [main] INFO main.StringDigitExtractor - Extract digits result for
    input a112c is [1, 1, 2]
...

```

If we wanted the app to also log the messages with the DEBUG severity, you would have to change the logger definition as presented in listing 5.6.

Listing 5.6 Using a different severity configuration

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">      #A
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{yy-MM-dd HH:mm:ss.SSS} [%t]
                %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>

    <Loggers>
        <Root level="debug">      #B
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>

```

#A Setting the logging level for internal Log4J events

#B Changing the logging level to debug.

In listing 5.6, you'll observe a "status" and a "logging level". This aspect usually creates confusion. Most of the time, you care about the level attribute. The "level" attribute tells which messages will be logged by your app according to their severity. The "status" attribute in the `<Configuration>` tag is the severity of the Log4J events, so the issues the library encounters. We can say that the "status" attribute is the logging configuration of the logging library.

After changing the logger as presented in listing 5.6, you'll observe the app also writes the messages with the priority as presented in the next snippet.

```

21-07-28 13:18:36.164 [main ] INFO main.StringDigitExtractor - Extracting digits for input
ab1c
21-07-28 13:18:36.175 [main] DEBUG main.StringDigitExtractor - Parsing character a of input
ab1c
21-07-28 13:18:36.176 [main] DEBUG main.StringDigitExtractor - Parsing character b of input
ab1c
21-07-28 13:18:36.176 [main] DEBUG main.StringDigitExtractor - Parsing character 1 of input
ab1c
21-07-28 13:18:36.176 [main] DEBUG main.StringDigitExtractor - Parsing character c of input
ab1c
21-07-28 13:18:36.177 [main] INFO main.StringDigitExtractor - Extract digits result for
input ab1c is [1]
21-07-28 13:18:36.181 [main] INFO main.StringDigitExtractor - Extracting digits for input
a112c
...

```

A logger library gives you the needed flexibility to make sure you only log what you need and avoid writing unnecessary messages. Writing the minimum of log messages you need to investigate a certain issue is a good practice helping you understand the logs easier but also keeping the app performant and maintainable. It also gives you the possibility of configuring the logging without needing to re-compile the app.

5.2.3 Problems caused by logging and how to avoid them

We store log messages to give us later the possibility to use them to understand how the app behaved at a certain point in time or how the app behaved over time. Logs are necessary and extremely helpful in many cases, but they can also become malicious if misused. In this section, we discuss three main problems logs can cause and how to avoid them (figure 5.14):

- *Security and privacy issues* – caused by log messages exposing private data.
- *Performance issues* – caused by the app storing too many or too large log messages.
- *Maintainability issues* – log instructions make the source code more difficult to read.



Figure 5.14 Small details can sometimes cause big problems. Developers sometimes consider an app's logging capability as harmless by default and disregard the problems it may introduce. Logging, however, like all the other software capabilities, deals with the data and, wrongly implemented, can affect the app's functionality and maintainability.

SECURITY AND PRIVACY ISSUES

Security is one of my favorite topics and one of the most important subjects any developer needs to consider when they implement an app. One of the books I wrote concerns security,

and if you implement apps using Spring framework and want to learn more about securing them, I recommend you read my book, *Spring Security in Action* (Manning, 2020).

Surprisingly, logs can sometimes cause vulnerabilities in applications, and, in most cases, these issues happen because developers are not attentive to what details they expose. Remember that with logs, you make specific details visible to anyone that can access the logs. You always need to think about whether the data you log should or not be visible to those that can access the logs (figure 5.15).

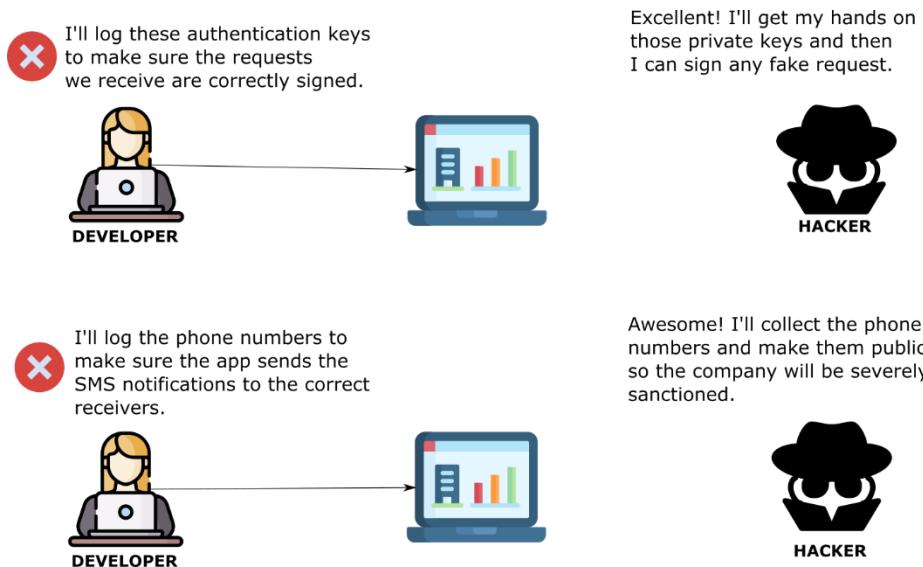


Figure 5.15 Log messages should not contain secret or private details. No one working on the app or the infrastructure where the app is deployed shouldn't access such data. Exposing sensitive details in logs can help a malicious person (hacker) find easier ways to break the system or attract severe sanctions for the company owning the app.

The following snippet shows some examples of log messages that expose sensitive details and cause vulnerabilities.

```
Successful login. User bob logged in with password RwjBaWIs66
Failed authentication. The token is unsigned. The token should have a signature with
IVL4KikMfz.
A new notification was sent to the following phone number +1233...
```

What's wrong with the logs presented in the previous snippet? The first two log messages expose private details. You should never log passwords or private keys used to sign tokens or any other exchanged information. A password is something only its owner should know. For this reason, no app should store any password in clear text (be it in a log or the

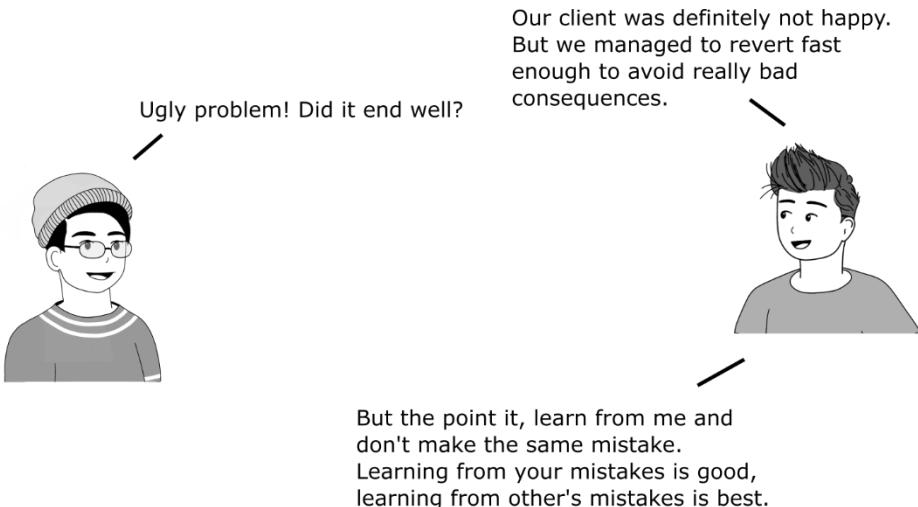
database). Private keys and similar secret details are stored in secrets vaults to protect them from being stolen. If someone would get the value of such a key, they could impersonate an application or a user acting on their behalf.

The third log message example in the snippet exposes a phone number. A phone number is considered a personal detail, and around the world, specific regulations restrict the usage of such details belonging to individuals. For example, the European Union applies a set of regulations named General Data Protection Regulation (GDPR) starting with May 2018. An application with users in any European Union state must comply with these regulations to avoid severe sanctions. Throughout various restrictions on using the users' personal data, the regulations imply that any user can request all their personal data an app uses and request immediate deletion of the data. Storing information such as phone numbers in logs exposes these private details and makes retrieving and deletion more difficult.

PERFORMANCE ISSUES

Writing logs imply sending details (usually as strings) through an I/O stream somewhere outside the app. We could simply send this information to the app's console (terminal), or we could store it in files or even a database, as we'll discuss in section 5.3. Either way, you need to remember that logging a message is also an instruction that takes time. So adding too many log messages could dramatically affect the apps' performance.

I remember an issue my team and I had to investigate some years ago. One of the customers in Asia reported a problem with the application we were implementing. We were implementing an app used in factory plants for inventory purposes. The reported problem wasn't causing big trouble, but we found it challenging to get to what's causing it, so we decided to add some more log messages that we hoped to help us discover the root cause. After delivering a patch with the small change, the system became very slow, almost unresponsive sometimes, and, long-story-short, we caused a production standstill and had to revert our change fast. We somehow managed to change the mosquito into an elephant with what was most likely one of the biggest mistakes for which I was also part of the cause.



But how could some simple log messages cause such big trouble? The way the logs were configured to be stored in that specific situation was to send the messages to a separate server in the network where they persisted. The fact that the network was extremely slow in that factory plant combined with the log message added to a loop that was iterating over a significant number of items made the app extremely slow.

But, in the end, we learned some things that helped us be more careful and avoid repeating the same mistake:

- Make sure you understand how the app logs the messages. Remember that even for the same app, different deployments might have different configurations. In section 5.2.3, we discussed different ways to persist logs and their advantages and disadvantages.
- Avoid logging too many messages. Don't log messages in loops iterating over a large number of elements. Logging too many messages would anyway make reading the logs complicated. If you need to log messages in a large loop, use a condition to narrow the number of iterations for which the message is logged.
- Make sure that the app stores a given log message only when that's really needed. You can achieve this using logging levels, as we discussed in section 5.2.2.
- Implement the logging mechanism in such a way to be able to enable and disable it without needing to restart the service. This approach will help you change the logging level to a finer grained one, get your needed details, and then make it less sensitive again.

AFFECTING MAINTAINABILITY

Another thing that log messages could affect negatively is the app's maintainability. If you add log messages too frequently, they can make the app's logic more difficult to understand. Let me give you an example: try reading listings 5.7 and 5.8. Which of these listings show an easier-to-understand code?

Listing 5.7 A method implementing a simple piece of logic

```
public List<Integer> extractDigits() {
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < input.length(); i++) {
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }

    return list;
}
```

Listing 5.8 shows the same code but using an excessive number of log messages.

Listing 5.8 A method implementing a simple piece of logic crowded with log messages

```
public List<Integer> extractDigits() {
    log.info("Creating a new list to store the result.");
    List<Integer> list = new ArrayList<>();
    log.info("Iterating through the input string " + input);
    for (int i = 0; i < input.length(); i++) {
        log.info("Processing character " + i + " of the string");
        if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {
            log.info("Character " + i +
                    " is digit. Character: " +
                    input.charAt(i));
            log.info("Adding character" + input.charAt(i) + " to the list");
            list.add(Integer.parseInt(String.valueOf(input.charAt(i))));
        }
    }

    Log.info("Returning the result " + list);
    return list;
}
```

Both listings 5.7 and 5.8 show the same piece of logic implemented. But in listing 5.8, I added many log messages that make the method's logic more challenging to read as you observe.

How do we avoid affecting the app's maintainability? The things you need to remember are:

- You don't necessarily need to add a log message for each instruction in the code. Identify those instructions that could provide the most relevant details, and remember that you can add extra logging later if you really encounter a case where the existing log messages are not enough.
- Keep the methods small enough so that you only need to log the parameters' values and the value the method returned after the execution.
- Some frameworks give you possibilities to decouple a part of the code from the method. For example, in Spring, you can use custom aspects to log the result of a method's execution (including the parameters' values and the value the method returned after the execution).

5.3 Logs vs. remote debugging

In chapter 4, we discussed remote debugging, and you learned that you could connect the debugger to an app executing in an external environment. I start this discussion because my students often ask me: why we would even need to use logs since we can connect and directly debug a given issue. But as I mentioned earlier in this chapter and also in the previous ones, these debugging techniques you learn don't exclude one another. Sometimes one is better than the other, in other cases, you need to use them together.

So let's analyze what we could and couldn't do with logs versus remote debugging to figure out how you can efficiently use these two techniques. The following table shows a side-by-side comparison of logs and remote debugging.

Table 5.1 Logs vs. Remote debugging

Capability	Logs	Remote debugging
Can be used to understand a remotely executing app's behavior.		
Needs special network permissions or configurations to use it.		
Persistently stores execution clues.		
Allows you to pause the execution on a given line of code to understand what the app does.		
Can be used to understand an app's behavior without interfering with the executed logic.		
Is recommended for production environments.		

You can use both logs and remote debugging to understand the behavior of a remotely executing app. But both approaches have their own difficulties. Logging implies that the app already writes the events and the data needed for the investigation. If that's not the case, then you need to add those instructions and redeploy the app. This technique is what developers usually name "adding extra logs". Remote debugging allows your debugger to connect to the remotely executing app, but to do that, specific network configurations and permissions need to be granted.

A big difference is the philosophy each technique implies. Debugging is focused on the present. You pause the execution and observe the app's state. Logging is more about the past. You get a bunch of log messages and analyze the execution focusing on a timeline. It's

no surprise if you'd find yourself using both simultaneously to understand more complex issues, and I can tell you from my experience that sometimes using logs versus debugging depends on the developer's taste. I sometimes see developers using one or another technique just because they feel more comfortable with their choice. This phenomenon happens because we are different, and our brains sometimes work differently.

5.4 Summary

- Always check the app's logs when you start investigating any issue. The logs might indicate to you what's wrong or at least give you a starting point for your investigation.
- Any log message should include the timestamp. Remember that a system doesn't guarantee the order in which the logs are stored in most cases. The timestamp will help you order the log messages chronologically and understand what time the message was logged.
- Avoid saving too many log messages. Not absolutely every detail is relevant and helpful in investigating a potential issue, and storing too many log messages can affect the app's performance, and you may also find it more difficult to read when investigating a case.
- You should implement logging to allow you to enable more logging only if needed. This way, a running app logs only essential messages, but you can enable more logging for a short time if you require more details for the scenario you investigate.
- An exception in the logs is not necessarily the problem itself. The exception could be a consequence of a problem. Research first what caused the exception before treating it locally.
- You can use exception stack traces simply to find out who called a given method. In large, messy, and challenging to understand codebases, this approach could be very helpful, and it could save you plenty of time.
- Never write sensitive details (such as passwords, private keys, or personal details) in a log message. Logging passwords or private keys introduce security vulnerabilities since anyone having access to the logs would be able to see and use them. Writing personal details such as names, addresses, or phone numbers could not comply with various governments' regulations. It could attract penalties to the organization providing the app.

6

Identifying resource consumption problems using profiling techniques

This chapter covers

- Evaluating resources consumption
- Identifying issues with resources consumption

“And for you, Frodo Baggins, I give you the light of Earendil, our most beloved star. May it be a light to you in dark places when all other lights go out.”

— Galadriel (*Lord of the Rings* by J.R.R. Tolkien)

In this chapter, we start discussing using a profiler. We'll continue the discussion in chapter 7. A profiler might not be as powerful as the light of Earendil, but this tool is definitely a light in dark cases when all the other lights go out. The **profiler** is a powerful tool that helped me understand the root cause of an app's strange behavior in many difficult situations. I consider learning to use a profiler is a must for any developer, as it can be the compass to guide you in situations where you lost hope in finding out what causes a given problem. As you'll learn in this chapter, the profiler intercepts the executing JVM processes and offers extremely useful details such as

- How the app consumes resources such as the CPU and memory
- The threads in execution and their current statuses
- The code in execution and the resources spent by a given piece of code (such as the duration of each method execution)

In section 6.1, we analyze some scenarios where you'll understand how these details can be useful and why they are so important. In section 6.2, we discuss using a profiler to solve scenarios such as the ones we discuss in section 6.1. We'll start by installing and configuring

a profiler in section 6.2.1. We'll then continue with analyzing how the app consumes the system resources in section 6.2.2, and in section 6.2.3, we discuss identifying when an app has issues with managing the used memory.

We'll continue our discussion about using a profiler in chapter 7, where you'll learn how to identify the code in execution and eventually performance problems related to it.

For the examples of this chapter, we'll use the VisualVM profiler. VisualVM is a free-to-use profiler and an excellent tool I've successfully used for many years. You can download VisualVM here: <https://visualvm.github.io/download.html>. VisualVM is not the only profiling tool that exists for Java apps. Some other known examples of tools that offer profiling capabilities are Java Mission Control (<https://www.oracle.com/java/technologies/jdk-mission-control.html>) and JProfiler (<https://www.ej-technologies.com/products/jprofiler/overview.html>

). However, the things we'll do with VisualVM you can similarly do with other tools.

6.1 Where would a profiler be useful?

In this section, we analyze three scenarios where using a profiling tool would help you. This way, you'll know the profiler might be the tool you need when you encounter a similar situation. We'll discuss:

- Identifying abnormal usage of resources
- Finding out which part of code executes
- Identifying slowness in app's execution

In section 6.2, we'll then deal in detail with observing the resources usage and ways to identify problems related to resource consumption. Chapter 7 discusses identifying code in execution and how you can granularly investigate how it spends resources.

6.1.1 Identifying abnormal usage of resources

One of the most common usages of a profiler is observing how an app consumes CPU and memory. Observing how the app consumes the CPU and memory resources usually helps you understand if the app has specific problems and is the first step in investigating such issues. In section 6.2, we'll discuss using a profiler to observe the resources consumption and spot potential CPU and memory usage issues. In chapters 10 through 11, you'll then learn how to continue such an investigation to identify the root cause of such problems using thread dumps and heap dumps.

Observing how the app consumes resources will usually lead you to two categories of issues:

1. **Thread related issues** - usually concurrency issues caused by lack or improper synchronization
2. **Memory leaks** – situations where the app fails to remove unneeded data from memory, causing slowness in execution and potentially a complete failure of the app.

I encountered both categories of issues in real-world apps more than I would have desired. My "favorite" thread-related issue I had to solve and investigate using a profiler was causing

battery problems on a mobile device. The effects of such issues are very diverse. In some cases, they only cause slowness in the app, in other cases, they might cause the app to fail entirely. For this scenario, I remember the slowness wasn't the biggest problem. Being a mobile app running on an Android device, the users observed that the device's battery consumed unnaturally fast when using this app. This behavior was definitely caused by an issue that required investigation. After spending some time investigating the app's behavior, it turned out that one of the libraries the app used was sometimes creating threads that remained in executing, doing nothing but consuming the system's resources. In a mobile app, the usage of the CPU resources most often reflects better in the battery's consumption. Once you discover the potential problem, you need to investigate it further with a thread dump, as you'll learn in chapter 10. Generally, the root cause of such problems is a faulty synchronization of the threads.

I find memory leaks now and then in apps. In most cases, the final result of a memory leak is an `OutOfMemoryError` that leads to an app crash. So when I hear about an app crashing now and then randomly, I first think it should be a memory problem.



TIP Whenever you hear about an app crashing randomly, you can suspect a memory leak.

The root cause of such problems is often an error in coding that allows object references to exist even after the objects are no longer needed. Remember that although the JVM has an automatic mechanism for releasing the unneeded data from memory (we name this mechanism the Garbage Collector [GC]), it's still the developer's responsibility to ensure all references to unnecessary data are removed. If we implement the code to keep references to objects, the GC doesn't know they aren't used anymore and won't remove them. We call such a situation a "memory leak". In section 6.2.3, you'll learn to use the profiler to identify when such a problem exists, then, in chapter 11, you'll learn to research the root cause of such an issue using a heap dump.

6.1.2 Finding out what code executes

As a developer and consultant, I sometimes worked with large, complex, and dirty code bases. I was several times in a situation where I had to investigate a particular app capability, could reproduce a problem but had no idea which part of the code is involved with that capability's execution. You first need to know what code executes, read the code, and potentially use a debugger to identify the root cause to solve a problem.

Some years ago, I had to investigate a problem with a given legacy app running some processes. The company's management made the uninspired decision to let only one developer be responsible for the given code. No one else had any idea what's there and how

to work with it. When that developer left, with no documentation and without a friendly codebase, I was asked to help identify the cause of an issue. The first look through the code scared me a bit: no class design at all, and a combination of Java and Scala mixed with some Java reflection-driven code.

So how do you find out which code you need to investigate in such a case?

Fortunately, a profiler has the useful capability to sample the executing code. The tool intercepts the methods that execute and visually displays what execute, which gives you just enough to start an investigation. Once you find the code in execution, you can read it and, eventually, decide to use a debugger, as discussed in chapters 2 to 4.

With a profiler, you can find what code executes behind the scenes without first looking into the code. This capability of the profiler is called "**sampling**" and it is particularly useful when the code is so muddy that you can't understand what and who is calling.

6.1.3 Identifying slowness in an app's execution

In some cases, you have to deal with performance problems. The general question you want to answer for such cases is: what takes so long to execute? Empirically, the developers always suspect first the parts of the code related to I/O communication. Calling a web service, connecting to a database, or storing data in a file are examples of I/O actions that are often causes of latencies in apps. Still, it's not always the case that an I/O call causes the slowness problem. And even then, unless you know the codebase by heart (which rarely happens), it would still be difficult to spot where the problem comes from without some help.

Fortunately, the profiler is quite "magical", having the ability to intercept code in execution and calculate the resources each piece of code consumes. We'll discuss these abilities that help you identify the code in execution and how it consumes the resources in chapter 7.

6.2 Using a profiler

In this section, we'll demonstrate using a profiler to solve issues like those presented in the scenarios we discussed in section 6.1. We'll begin (in section 6.2.1) with installing and configuring VisualVM, which is the tool we'll use in this book for profiling. We'll then examine the different things the profiler offers that you'll use when investigating issues. We'll use an app for the demonstration for each topic, small enough to allow you to focus on the presented subject but complex enough to be relevant to our discussion.

In section 6.2.2, we discuss system resource consumption and how to identify if your app has any issues in this regard. In section 6.2.3, you'll learn what kind of memory issues an app can encounter and how to spot such issues.

6.2.1 Installing and configuring VisualVM

In this section, you'll learn to install and configure VisualVM. Before using a profiler, you need to make sure you correctly install and configure such a tool. Then, you can use the examples provided with the book to try out each of the capabilities we discuss in this chapter. If you work on a real-world project, I recommend using the techniques we discuss with the app you implement.

Installing VisualVM is straightforward. Once you downloaded the version according to your operating system from the official site (<https://visualvm.github.io/download.html>), the only thing you need to make sure is you correctly configure is the location of the JDK you want VisualVM to use. In the configuration file (visualvm.conf in the /etc subfolder), which you find at the etc/visualvm.config location in the VisualVM's folder, define the location of the JDK in your system. You need to assign the JDK path to the `visualvm_jdkhome` variable and uncomment the line (remove the # in front of it), as presented in the next snippet. VisualVM works with Java 8 or above.

```
visualvm_jdkhome="C:\Program Files\Java\openjdk-17\jdk-17"
```

Once you configured the JDK location, you can run VisualVM using the executable you find in the bin folder where you installed the app. If you correctly configured the JDK location, the app will start, and you'll see an interface similar to the one presented in figure 6.1.

All the Java processes running locally.

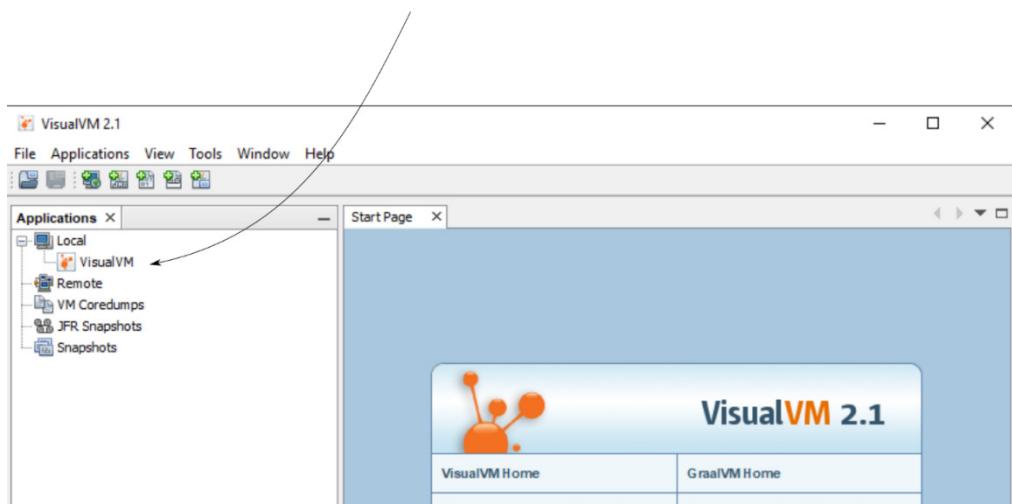


Figure 6.1 VisualVM welcome screen. Once you configured and started VisualVM, you'll find that the tool has a simple and easy-to-learn GUI. The welcome screen displays on the left the processes running locally that you can investigate with the tool.

Let's start a Java app and observe that the process appears in the left part of the VisualVM frame. You can use the project da-ch6-ex1 provided with the book. Either use the IDE to start the app or start the app from the console directly. Profiling a Java process is not affected by the way the app was started.

Once you started the app, VisualVM displays the process on the left side. Usually, if you didn't explicitly give a particular name to the process, VisualVM displays it using the main class name as presented in figure 6.2.

Once you start your app, you will also see its process on the left side of the VisualVM frame. Since we gave no particular name to our process, VisualVM displays it using the main class name.

Double-click on the process name and VisualVM displays the details tab for the process.

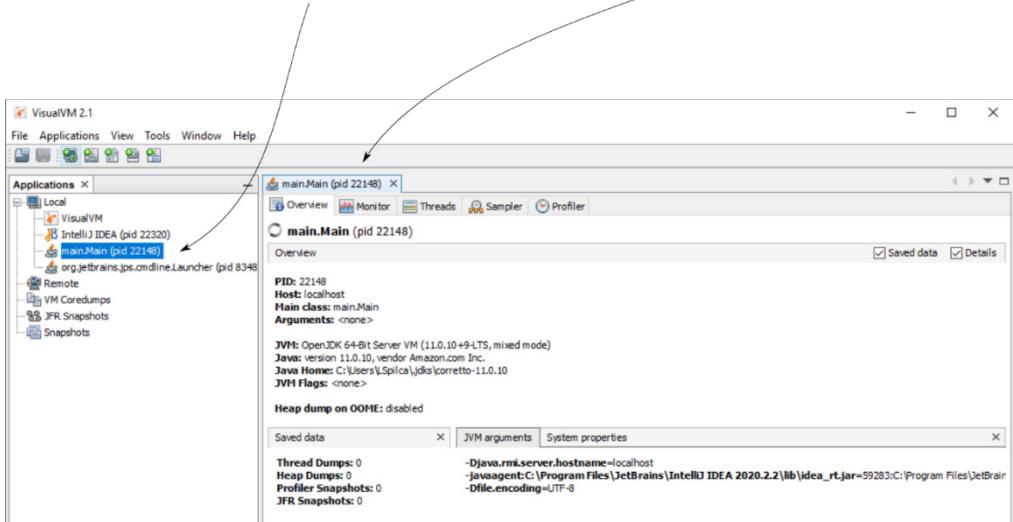


Figure 6.2 Double-click on a process name to start using VisualVM for investigating that process. After double-clicking on the process name, a new tab will appear. You'll find in this tab all the needed capabilities VisualVM provides for exploring that particular process.

Generally, starting the app should be enough. However, in some cases, I observed that VisualVM doesn't know how to connect to a local process because of various issues, as presented in figure 6.3. In such a case, the first thing to try is explicitly specifying the domain name using a VM argument when starting the application you want to profile, as presented in the next snippet.

```
-Djava.rmi.server.hostname=localhost
```

A similar problem could also be caused by using a JVM version that VisualVM doesn't support. If adding the `-Djava.rmi.server.hostname=localhost` argument doesn't solve your issue, also check that the JVM distribution you configured is among the ones VisualVM supports (according to the download section on their website <https://visualvm.github.io/download.html>).

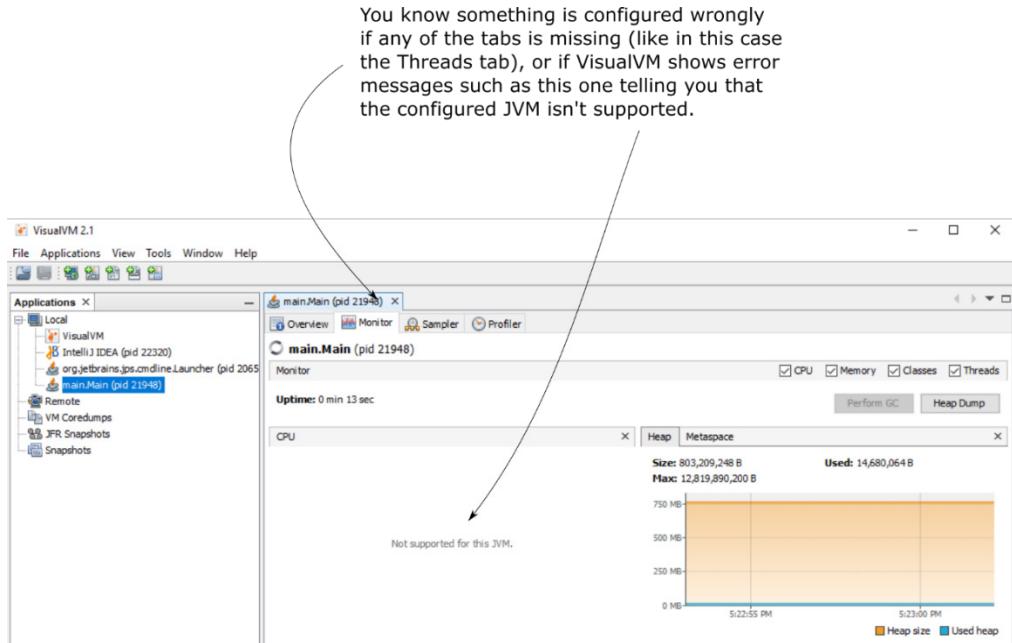


Figure 6.3 If the tool seems not to work properly, you need to check the way it is configured. Such problems are either caused because the configured JVM distribution is not among the ones VisualVM supports. Sometimes, the tool can't connect to the local process you want to investigate for some reason. In such cases, to solve the problem, use a different JVM distribution that complies with the tool's requirements or check how the process you want to investigate was started.

6.2.2 Observing the CPU and memory usage

One of the simplest things you can do with a profiler is to observe how your app consumes the system's resources. This way, you can spot problems such as memory leaks or zombie threads in your app.



A **memory leak** is when your app doesn't deallocate unneeded data. In time, there's no more free memory. This is a problem.

As you'll learn in this section, you can visually identify if your app doesn't correctly behave using the profiler. Zombie threads are threads that remain in a continuous execution consuming the app's resources. As you'll learn in this section and section 6.2.3, you can easily observe such problems using VisualVM.

I prepared some projects to show you how to use the profiler to identify app issues that cause abnormal resource consumption. We'll run the apps provided with the book one by one, and we'll use VisualVM to observe the behavior and identify abnormalities.

Let's start the app da-ch6-ex1. The idea of the app is simple. Two threads continuously add values to a list, while two other threads continuously remove (consume) the values from this list. You find the implementation of the producer thread in listing 6.1. We often call this implementation a producer-consumer approach which is a multithreaded design pattern commonly encountered in apps.

Listing 6.1 The producer thread adds values to a list

```
public class Producer extends Thread {
    private Logger log = Logger.getLogger(Producer.class.getName());
    @Override
    public void run() {
        Random r = new Random();
        while (true) {
            if (Main.list.size() < 100) {      #A
                int x = r.nextInt();
                Main.list.add(x);          #B
                log.info("Producer " + Thread.currentThread().getName() +
                        " added value " + x);
            }
        }
    }
}
```

#A Setting a maximum number of values for the list.
#B Adding a random value in the list.

Listing 6.2 shows the implementation of the consumer thread.

Listing 6.2 The consumer thread removes values from the list

```
public class Consumer extends Thread {
    private Logger log = Logger.getLogger(Consumer.class.getName());

    @Override
    public void run() {
        while (true) {
            if (Main.list.size() > 0) { #A
                int x = Main.list.get(0);
                Main.list.remove(0); #B
                log.info("Consumer " + Thread.currentThread().getName() +
                    " removed value " + x);
            }
        }
    }
}
```

#A Check if the list contains any value.

#B If the list contains values, remove the first value from the list.

The `Main` class (listing 6.3) creates and starts two instances of the producer thread and two instances of the consumer thread.

Listing 6.3 The Main class creates and starts the producer and consumer threads

```
public class Main {
    public static List<Integer> list = new ArrayList<>(); #A

    public static void main(String[] args) {
        new Producer().start(); #B
        new Producer().start(); #B
        new Consumer().start(); #B
        new Consumer().start(); #B
    }
}
```

#A Creating a list to store the random values the producer generates.

#B Starting the consumer and produces threads.

This application wrongly implements a multithreaded architecture. More precisely, multiple threads concurrently access and change a list of type `ArrayList`. Because `ArrayList` is not a concurrent collection implementation in Java, it doesn't manage itself the threads access. Multiple threads accessing this collection potentially enter in a **race condition**.

A race condition happens when multiple threads compete to access the same resource. We say they are in a race to access that resource.

In project da-ch6-ex1 you find an implementation that lacks the threads synchronization. Running the app, you'll observe that after a short time, some of the threads stop because of exceptions caused by the race condition, while others will remain forever alive doing nothing (zombie threads). We'll use VisualVM to identify all these problems. Then, we'll run project da-ch6-ex2 which applies a correction to the app, synchronizing the threads that access the

list. We'll compare the results displayed by VisualVM for the first example with the second example to understand the difference between a normal vs. a problematic app.

The app will run quickly and then stop (potentially showing an exception stack trace in the console). The next code snippet shows what the log messages the app prints in the console look like.

```
Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-0 added value -361561777
Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-1 added value -500676534
Aug 26, 2021 5:22:42 PM main.Producer run
INFO: Producer Thread-0 added value 112520480
```

You might think that this app only has three classes. You don't need any profiler to spot the problem. Reading the code is enough here. The apps we use are simplified examples to allow you to focus on using the profiler. Indeed, with only three classes in the app, you might be able to spot the problem without using a separate tool. But in the real world, the apps would be more complex, and problems would be a lot more challenging to spot without an appropriate tool (such as a profiler).

Even if the app looks like it's taking a pause from running, you'll observe something interesting when you use VisualVM to investigate what happens behind the scenes. To investigate this unexpected behavior, we'll follow a few steps:

1. Check the process CPU usage
2. Check the process memory usage
3. Visually investigate the executing threads

The process consumes a lot of CPU resources, so, somehow, it seems to be still alive. To observe that, use the **Monitor** tab in VisualVM after double-clicking the process name in the left panel. One of the widgets you'll find on this tab shows you the CPU usage (figure 6.4).

What actually happens is that the consumer and producer threads seem to have entered a continuous running state where they consume the system's resources even if they don't correctly fulfill their tasks. In this case, the state is a consequence of the race conditions the threads faced by trying to access and change a non-concurrent collection. But we already know it's something wrong with this app. We want to observe the symptoms such problems cause, so that in other similar situations you'll realize your app encountered a similar problem.

Also, in this widget, you find how much CPU resource the Garbage Collector (GC) uses. The GC is the JVM mechanism that deals with removing the data the app doesn't need anymore from memory. The Garbage Collector CPU usage is valuable information because it can indicate that the app has a problem with memory allocation. If the GC spends a lot of CPU resources, it might signify that the app has a memory leak issue. We'll discuss more in detail identifying memory leaks in section 6.2.3.

But in this case, the GC doesn't spend any CPU resources. This is not a good sign. A translation for this could be: "The app spends a lot of processing power but doesn't process anything". These signs usually indicate zombie threads which are generally a consequence of concurrency problems.

You can use the CPU usage widget to check the CPU consumption. Observe that in this case the process spends about 50% of the system's processing power.

Another interesting fact to observe is how much is the Garbage Collector responsible for the CPU consumption. In this case, the Garbage Collector spends no resources.

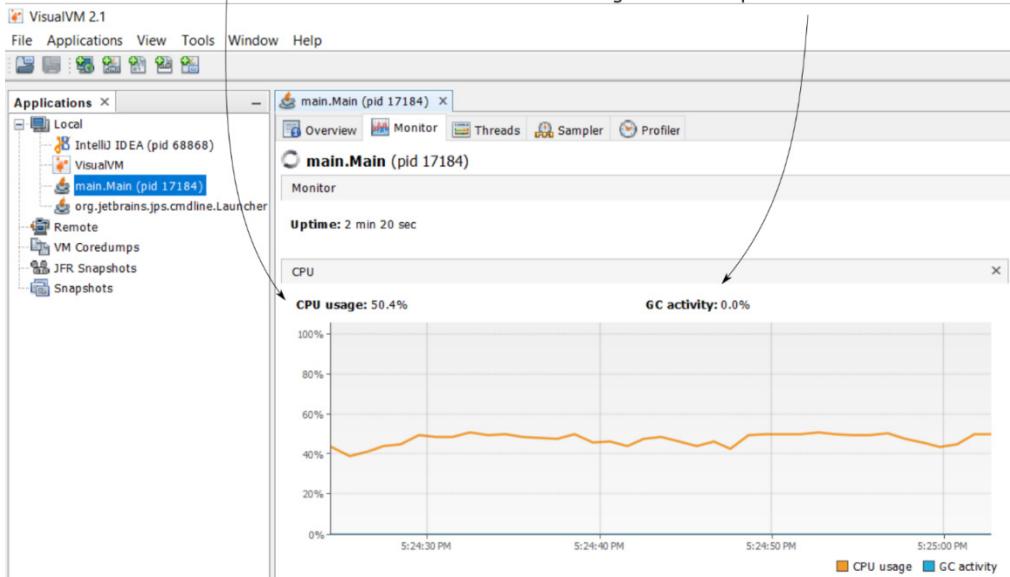


Figure 6.4 Using VisualVM to observe the CPU resources usage. The widget you find in the Monitor tab shows you how much CPU the process uses and how much of the usage is caused by the Garbage Collector. This information helps you understand if the app has execution problems and is excellent guidance for the next steps in your investigation. In this particular example, the process spends about 50% CPU. The GC doesn't influence this value. These signs are usually indicators of zombie threads usually generated by concurrency problems.

Generally, the next step would be to throw an eye also at the widget showing you the memory consumption. This widget is strategically placed near the one showing you the CPU consumption, as presented in figure 6.5. We'll discuss this widget in more detail in section 6.2.3, but for the moment, just observe that the app doesn't spend almost any memory at all. This behavior is again not a good sign, being equivalent to "the app does nothing". So only using these two widgets, we concluded that we are most likely facing a concurrency issue.

On the right side of the CPU usage widget, you find another widget displaying the memory consumption.

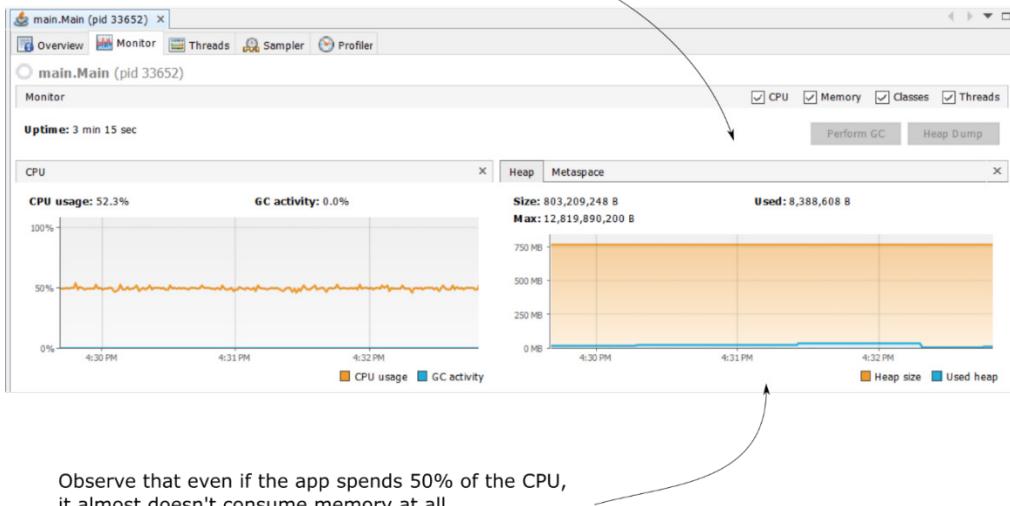


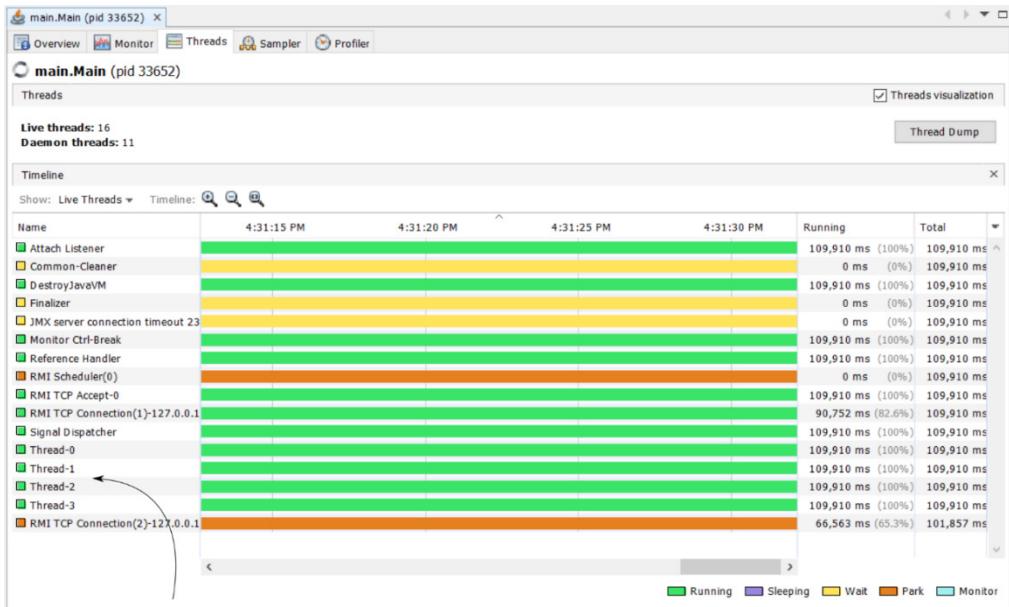
Figure 6.5 On the right side of the CPU usage widget, you find the memory usage widget. You can observe in this example that the app doesn't use almost any memory at all. This is also the reason why we observe the GC activity is zero. App not consuming any memory is not a good sign since it's somehow equivalent to the app not doing anything.



Before going into the details and investigating the threads in execution, I prefer to use VisualVM to observe visually how the threads execute. In most cases, doing so gives me some clues about which threads I need to pay attention to. Once I get this info, I use a thread dump to find out the concurrency problem and understand how to fix it.

We'll discuss using thread dumps in chapter 10. For now, we will focus only on the high-level widgets the profile offers, and we'll compare the results these widgets provide for a healthy and unhealthy app.

Figure 6.6 shows the **Threads** tab, which you find near the Monitor tab. The **Threads** tab offers a visual representation of the threads in execution and their states. In this example, you see that all the four threads the app started are still executing and in a running state.



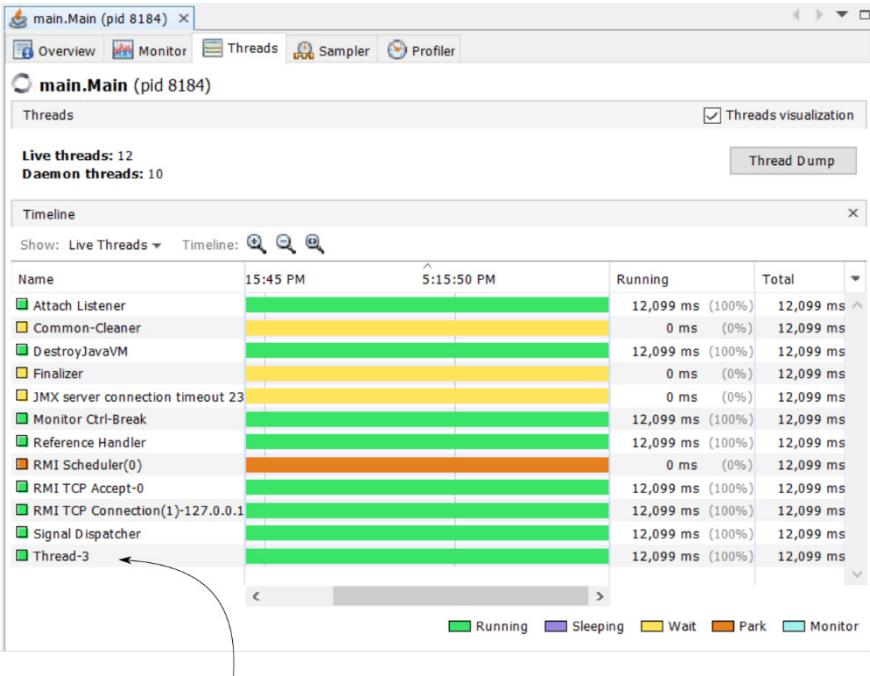
Even if the app doesn't seem to do anything,
the four threads it created are continuously running.
These running threads that do nothing but staying alive
are called zombie threads. The only thing they do now
is consuming CPU resources.

Figure 6.6 The Threads tab offers a visual representation of the threads that are alive and their statuses. Observe the widget shows all the process threads, including those started by the JVM. This widget helps you easily identify which threads you should pay attention to and eventually investigate deeper using a thread dump.

Mind that concurrency problems might have different results. Not necessarily all the threads will remain alive, for example. Sometimes, the concurrent access might cause exceptions that interrupt some or all the threads entirely. The next snippet shows an example of such an exception that can occur during app's execution.

```
Exception in thread "Thread-1" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
 bounds for length 109
 at java.base/java.util.ArrayList.add(ArrayList.java:487)
 at java.base/java.util.ArrayList.add(ArrayList.java:499)
 at main.Producer.run(Producer.java:16)
```

If such an exception happens, then some threads might be stopped, and the **Threads** tab won't show them anymore. Figure 6.7 shows such a case where the app threw an exception, and only one of the threads stayed alive.



In this example, only one of the threads remained alive and became a zombie thread. The other threads encountered exceptions caused by the race conditions and stopped.

Figure 6.7 If exceptions occur during app's execution, some of the threads might be stopped. This figure shows a case where the concurrent access caused exceptions in three of the threads and stopped them. Only one thread remained alive. You must be aware that concurrency problems in multithreaded apps can have different unexpected results.

With this example, we focus only on discovering a resource consumption problem. From here, the next step would be using a thread dump to find out the exact cause of the concurrency problem that causes the issue. We'll discuss everything about thread dumps in chapter 7, but for now, let's remain focused on identifying resource consumption issues. Now, we'll run the same verifications on a healthy app and compare it with what you saw until now. This way, you'll know how to recognize correct and incorrect app behavior immediately.

The example you find in project da-ch6-ex2 is the corrected version of the same app. I added some synchronized blocks to avoid concurrent access for the threads and, this way, the race condition problems. I used the list instance as the thread monitor for the synchronized code blocks for both consumers and producers. Listing 6.4 shows the changes in the Consumer class.

Listing 6.4 Synchronizing the access for the consumer

```
public class Consumer extends Thread {
    private Logger log = Logger.getLogger(Consumer.class.getName());
    public Consumer(String name) {
        super(name);
    }
    @Override
    public void run() {
        while (true) {
            synchronized (Main.list) { #A
                if (Main.list.size() > 0) {
                    int x = Main.list.get(0);
                    Main.list.remove(0);
                    log.info("Consumer " + Thread.currentThread().getName() + " removed value " + x);
                }
            }
        }
    }
}
```

#A Synchronizing the access on the list, using the list instance as a thread monitor.

Listing 6.5 shows the synchronization applied to the `Producer` class.

Listing 6.5 Synchronizing the access for the producer

```
public class Producer extends Thread {
    private Logger log = Logger.getLogger(Producer.class.getName());
    public Producer(String name) {
        super(name);
    }
    @Override
    public void run() {
        Random r = new Random();
        while (true) {
            synchronized (Main.list) { #A
                if (Main.list.size() < 100) {
                    int x = r.nextInt();
                    Main.list.add(x);
                    log.info("Producer " + Thread.currentThread().getName() + " added value " + x);
                }
            }
        }
    }
}
```

#A Synchronizing the access on the list, using the list instance as a thread monitor.

Aside from synchronizing the threads, I also decided to give custom names to each thread. I always recommend this approach. Did you spot the default names the JVM gave our threads

in the previous example? Generally, Thread-0, Thread-1, Thread-2 are not names you can easily use to identify a given thread. I prefer giving threads custom names whenever I can to identify them when I need them quickly. Moreover, I give them names starting with an underline to be easier to sort them when I need them. Starting their name with an underline, it's easier to have them listed first. I defined the constructor in the `Consumer` and `Producer` classes as presented in listings 6.4 and 6.5 and used the `super()` constructor to give the names of the threads. I then gave them names as presented in listing 6.6.

Listing 6.6 Setting custom names for the threads

```
public class Main {
    public static List<Integer> list = new ArrayList<>();
    public static void main(String[] args) {
        new Producer("_Producer 1").start();
        new Producer("_Producer 2").start();
        new Consumer("_Consumer 1").start();
        new Consumer("_Consumer 2").start();
    }
}
```

The first observation after starting this app is that the console continuously shows logs. The app doesn't stop anymore as it happened with example `da-ch6-ex1`. Let's use VisualVM to observe resource consumption. In the CPU utilization widget, you observe that the app spends less CPU, while in the memory usage widget, you now find that the app really uses some of the allocated memory while running. Also, we can observe also activity of the Garbage Collector. As you will learn later in this chapter, the valleys in the memory graph on the right side are a result of the activity of the GC.

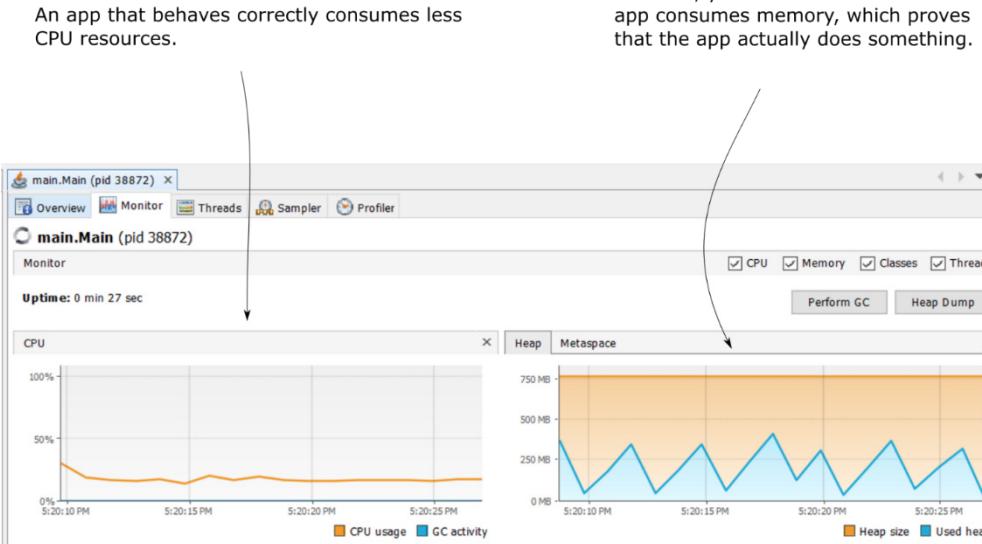


Figure 6.8 After correctly synchronizing the code, the resources consumption widgets look differently. You can observe that the CPU consumption is lower, and the app really uses memory.

Moreover, the **Threads** tab shows that the monitor sometimes blocks the threads, which only allows one thread at a time through a synchronized block. The fact that the threads don't run continuously makes the app consume less CPU, as shown in figure 6.8. Figure 6.9 shows the threads visualization in the **Threads** tab.

NOTE Even if we added the synchronized blocks, there is still code that remained outside of these blocks. For this reason, the threads may still appear to run concurrently sometimes (as shown in figure 6.9).

Observe that the threads are not continuously running anymore. The profiler shows you when the threads are blocked by a monitor, waiting, or sleeping.

Instructions that remained out of the synchronized blocks can still cause moments when we'll see threads running concurrently. Observe this case where the two producer threads appear both green at the same time on the diagram.



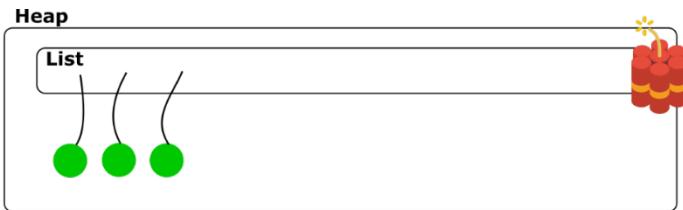
Figure 6.9 The Threads tab helps you visualize the execution of the threads in your app. Since we named all the threads starting with underline, you can simply sort them by name now to see them grouped. Observe their execution is interrupted from time to time by the monitor, which only allows one thread at a time through the synchronized blocks of code.

6.2.3 Identifying memory leaks

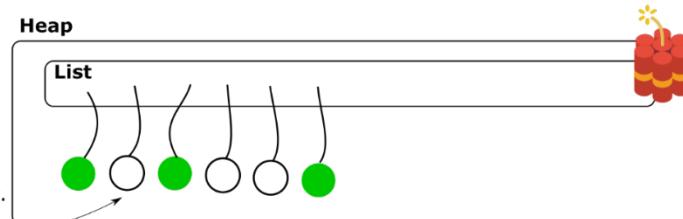
In this section, we discuss memory leaks and how to determine when your app is affected by such an issue. A memory leak is a wrong behavior of an app that stores and keeps references to unused objects (figure 6.10). Because of these references, the Garbage Collector (the mechanism responsible for removing unneeded data from the app's memory) cannot remove these objects. While the app continues to add more data, the memory fills up. When the app doesn't have enough space to add the new data, it throws an `OutOfMemoryError` and stops. We'll use a simple app that causes an `OutOfMemoryError` to demonstrate how to identify such an issue using VisualVM.

1

Suppose you have an app creating object instances and keeping references to these instances in a list.

**2**

The app continues to create new instances. Some of the previously created instances are not needed anymore, but the app doesn't remove their references from the list.

**3**

Because the app continues to keep the references, the GC fails to remove the unneeded objects from the memory. The memory gets full, and, at some point the app can't allocate anymore objects.

The process stops and the app fails with an `OutOfMemoryError`.

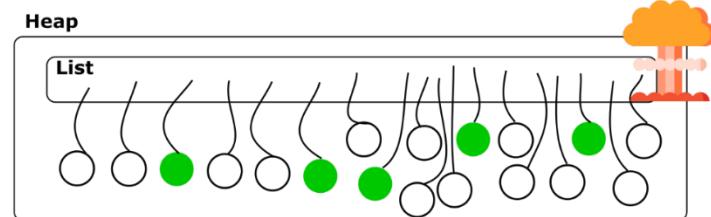


Figure 6.10 An `OutOfMemoryError` is like a ticking bomb. An app fails to remove references to objects it doesn't use anymore. The Garbage Collector (GC) can't remove these instances from the memory because the app keeps their references. While more objects are created, the memory gets full. At some point, there's no more space in the heap to allocate other objects, and the app fails with an `OutOfMemoryError`.

In the example provided with project da-ch6-ex3, you find a simple app that stores random instances into a list but never removes their references from there. Listing 6.7 shows you the simple implementation of an example producing an `OutOfMemoryError`.

Listing 6.7 Producing an `OutOfMemoryError`

```
public class Main {
    public static List<Cat> list = new ArrayList<>();
    public static void main(String[] args) {
        while(true) {
            list.add(new Cat(new Random().nextInt(10)));      #A
        }
    }
}
```

#A Continuously adding new instances to a list until the JVM runs out of memory.

Class `Cat` used in listing 6.7 is just a simple plain old java object as presented by the next code snippet.

```
public class Cat {  
    private int age;  
  
    public Cat(int age) {  
        this.age = age;  
    }  
  
    // Omitted getters and setters  
}
```

Let's run this app and observe resource usage with VisualVM. We're especially interested in the widget that shows memory usage. When a memory leak affects your app, this widget shows you that the used memory grows continuously. The GC tries to deallocate unused data from memory, but it can only remove too few. In the end, the memory gets filled, and the app cannot store the new data anymore and throws an `OutOfMemoryError` (figure 6.11).

Observe how the used memory grows continuously. The Garbage Collector makes efforts to free the memory but it can't remove most of the instances because the app still keeps their references in memory.

When all the allocated memory is occupied, and the app can't store the new data anymore, the app throws an `OutOfMemoryError`.

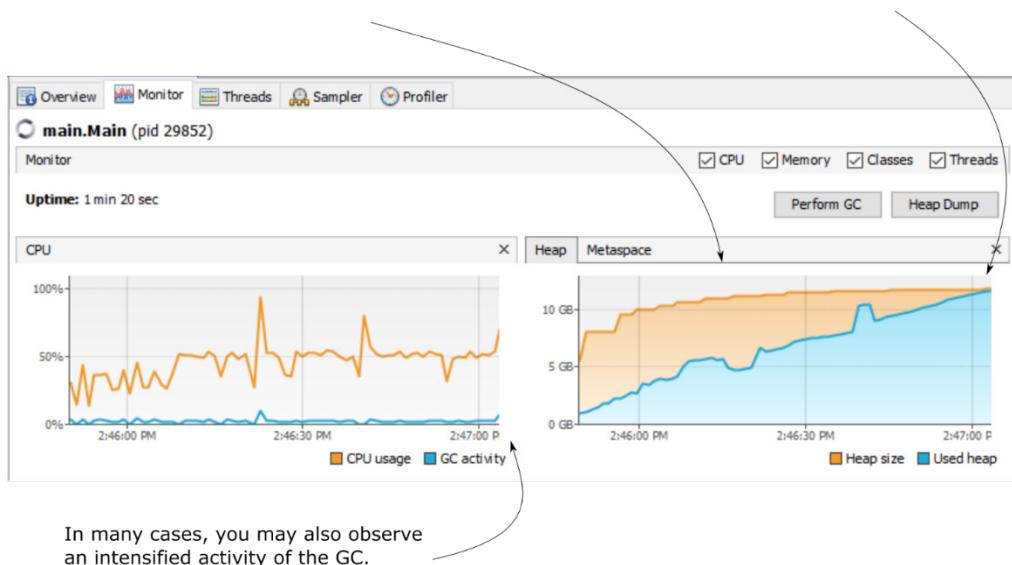


Figure 6.11 When a memory leak affects your app, the used memory grows continuously. You can observe that the Garbage Collector makes efforts to free the memory but cannot remove enough data. The used memory grows until the app can't allocate any more new data. At this point, the app throws an `OutOfMemoryError` and stops. In many cases, a memory leak also causes an intensified activity of the GC,

which can be seen in the CPU resource usage widget.

If you let the app run long enough, you'll eventually see the error stack trace in the app's console. The next snippet shows what the stack trace may look like.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.base/java.util.Arrays.copyOf((Arrays.java:3689)
    at java.base/java.util.ArrayList.grow(ArrayList.java:238)
    at java.base/java.util.ArrayList.grow(ArrayList.java:243)
    at java.base/java.util.ArrayList.add(ArrayList.java:486)
    at java.base/java.util.ArrayList.add(ArrayList.java:499)
    at main.Main.main(Main.java:13)
```

An important thing to remember is that an `OutOfMemoryError` stack trace doesn't necessarily indicate the place that causes the problem. Since an app only has one heap memory, it can be that a thread causes the problem, and another one is unlucky to be the last trying to use the memory location and get the error. The only sure way to identify the root cause is using heap dump, as you'll learn in chapter 11.

Figure 6.12 compares a normal behavior and the behavior of an app affected by a memory leak, as seen in VisualVM. For an app with a normal execution (not affected by a memory leak), you'll observe the graph has peaks and valleys. The app allocates memory filling it up (the peaks), but from time to time, the GC removes the data that's no longer needed (the valleys). Such a picture is usually a good sign that the capability you investigate is not affected by memory leaks.

However, if you spot that the memory progressively fills and the GC seems not to clean it, it might be a sign your app faces a memory leak. Usually, once you suspect a memory leak, you need to further investigate in detail what happens using a heap dump.

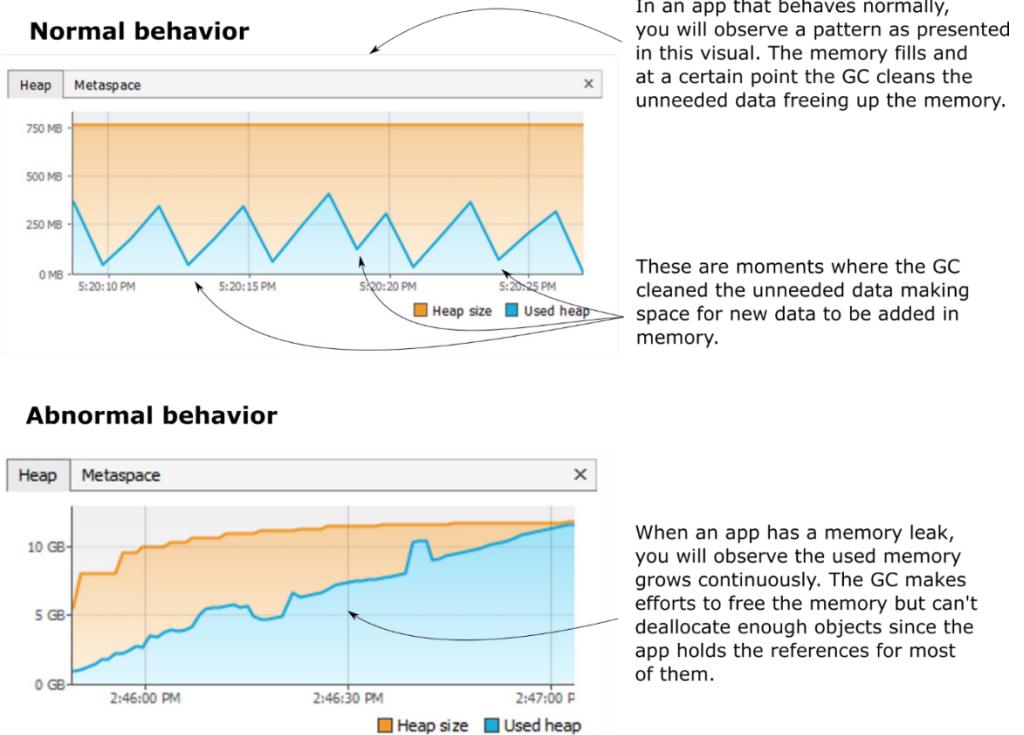


Figure 6.12 A comparison between the memory usage for a healthy app vs. an app suffering from a memory leak. The GC manages to free unneeded data from memory for a healthy app, never filling up all the allocated space. An app suffering from a memory leak doesn't allow the GC to remove enough data. At some point, the memory fills up completely, generating an `OutOfMemoryError`.

Mind that you can control the allocated heap size in a Java app. This way, you can enlarge the maximum limit the JVM allocates to your app. However, giving the app more memory is not a solution for a memory leak. This approach can be a temporary solution to give you more time to solve the real root cause of the problem. To set a maximum heap size for an app, you can use the JVM property `-Xmx` followed by the amount you want to allocate (E.g., `-Xmx1G` will allocate a maximum heap size of 1 Gb). You can similarly set a minimum and initial heap size using the `-Xms` property (E.g., `-Xms500m` would allocate a minimum heap size of 500Mb).

Aside from the normal heap space, any app also uses a metaspace. The metaspace is the memory location where the JVM stores the class metadata needed for the app's execution. In VisualVM you can observe the allocation of the metaspace too in the memory allocation widget. To evaluate the metadata allocation, use the **Metaspace** tab of the widget as presented in figure 6.13.

The Metaspace tab of the memory usage widget shows the size of the metaspace and how much of it is used.

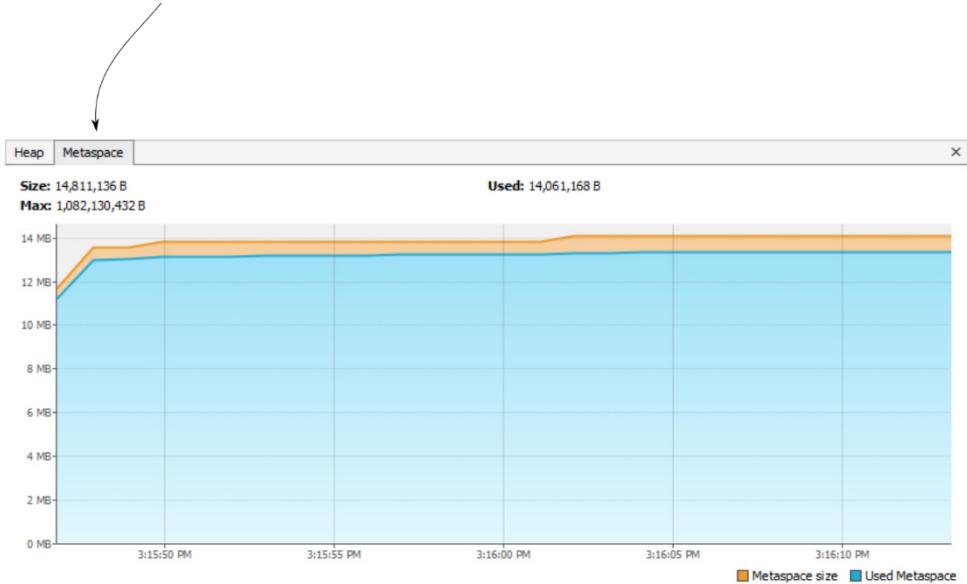


Figure 6.13 The metaspace is a part of the memory used to store class metadata. In particular cases, the metadata can be overflowed. VisualVM memory allocation widget also shows the usage of the metaspace.

An `OutOfMemoryError` on the metadata space happens less often, but it's not impossible. I recently dealt with such a case in an app that was misusing a framework for data persistence. Generally, frameworks and libraries using Java reflection are the most probable candidates to generate such problems if misused since they often rely on dynamic proxies and indirect calls.

In the situation I faced, an app was misusing a framework named Hibernate. I would not be surprised if you already heard about Hibernate since it's one of the most common solutions to manage persistent data in Java apps today, but no worries if you haven't heard about it yet. Hibernate is not the main point of our current discussion, and you'll get my point anyway. Hibernate is an excellent tool that helps implement the most used persistence capabilities of an app while eliminating the need to write unneeded code. Hibernate manages a context of instances and maps the changes to this context to the database. But one thing a developer using such a framework should know is that it's not recommended to have a very large context. In other words, don't work with too many records from the database at once!

The app I had trouble with defined a scheduled process loading many records from a database and processing them in a defined way. It seems that at some point, the number of records this process was fetching was so large that the load operation itself was causing the metaspace to fill—mind that the problem was misusing the framework, and not a bug in the

framework. The developers should never have used Hibernate for such a case but alternate, more low-level solutions like JDBC.

The problem was critical, and I had to find a short-term solution since a complete refactor would have taken a long time. Same as for the heap, you can customize the metaspace size. Using the `-XX:MaxMetaspaceSize` property you can enlarge the metaspace (E.g., `-XX:MaxMetaspaceSize=100M`). But remember that this is not a real solution to the problem. The long-term solution for such a case would be to refactor the functionality to avoid loading so many records at once in the memory and eventually use an alternate persistence technology if needed.

6.3 Summary

- A profiler is a tool that helps you observe the app's execution to identify the causes of certain problems which are more difficult to spot otherwise. A profiler shows you:
 - How an app spends system resources such as the CPU and memory
 - What code executes, and the duration of each method execution
 - The execution stack of methods on different threads
 - The executing threads and their statuses
- The profiler provides excellent visual widgets that help you understand certain aspects faster.
- You can observe the Garbage Collector's execution using the profiler. Observing how the Garbage Collector behaves helps you identify issues where the app doesn't correctly deallocate unused data from memory (memory leaks)

7

Finding hidden issues using profiling techniques

This chapter covers

- Sampling an app's execution to find the currently executing methods
- Observing execution times
- Identifying SQL queries the app executes

In chapter 6, I said a profiler is a powerful tool that can show you a path when all the lights have gone out (making an analogy here with the light of Earendil, which Galadriel gives Frodo in the epic *Lord of the Rings* novel by J.R.R Tolkien). Still, I've only shown you a small part yet. The profiler offers powerful techniques for investigating an app's execution, and learning to use these techniques properly helps you in many scenarios.

In many cases, I have had to evaluate or investigate app executions for codebases I could barely read – old apps without even a code design kept hidden in a wardrobe by companies. In such cases, the profiler was the only efficient way to find what was executing when a specific capability was triggered. You understand now why I compared the profiler with the light of Earendil. Because as Galadriel says, it really was a light in many dark places where all the other lights were out for me.

In this chapter, we're going to analyze three investigation techniques through profiling which I consider to be extremely valuable:

1. **Sampling** for finding out what part of an app's code executes
2. **Profiling** the execution (also named sometimes "instrumentation") to identify wrong behavior and optimization
3. Profiling the app to identify SQL queries an app uses to communicate with a Database Management System (DBMS)

We'll then continue our discussion in chapter 8 with advanced visualization techniques of an app's execution. When used appropriately, these techniques can save you tremendous time spent finding the causes of various issues. Unfortunately, even if these techniques are powerful, with experience, I observed many developers are unfamiliar with them. Some developers know these techniques exist but tend to believe they are difficult to use (I'll show you the contrary in this chapter). Consequently, many developers try using other methods to solve problems that could be solved much more efficiently with a profiler (as presented in this chapter).

To make sure you properly understand how to use these techniques and what issues can be investigated, I created four small projects. We'll use these projects to apply the profiling techniques we discuss. Section 7.1 discusses sampling – a technique you use to identify what code executes at a given time. In section 7.2, you'll learn how to profile for more details about the execution than sampling can offer. Section 7.3 discusses how to use a profiler to get details about SQL queries an app sends to a DBMS.

7.1 Sampling to observe executing code

What is sampling, and how can it benefit you? **Sampling** is an approach in which you use a profiler to identify what code the app executes. Sampling doesn't provide many details about the execution, but it draws the big picture of what happens, giving you valuable information on what you'll need to analyze further. For this reason, sampling should always be the first step when profiling an app, and, as you'll observe with our discussion, sampling may even be enough in many cases. For this section, I prepared project da-ch7-ex1. We'll use a profiler to sample this app and understand how we would use VisualVM to identify issues related to the execution time of a given capability.

The project we'll use to demonstrate sampling is a tiny app that exposes an endpoint /demo. When someone calls this endpoint, using cURL, Postman, or a similar tool, the app further calls an endpoint exposed by httpbin.org.

I like a lot using the site httpbin.org for many examples and demonstrations. Httpbin.org is an open-source web app and tool written in Python that exposes mock endpoints you can use to test different things you're implementing. In the current case, we call an endpoint where httpbin.org responds with a given delay. We'll use a 5-second delay for this example. We'll use this simple example to simulate a latency scenario in our app, and httpbin.org simulates the root cause of the problem.



By **latency**, we understand a situation where an app reacts slower than expected.

The scenario is also visually represented in figure 7.1.

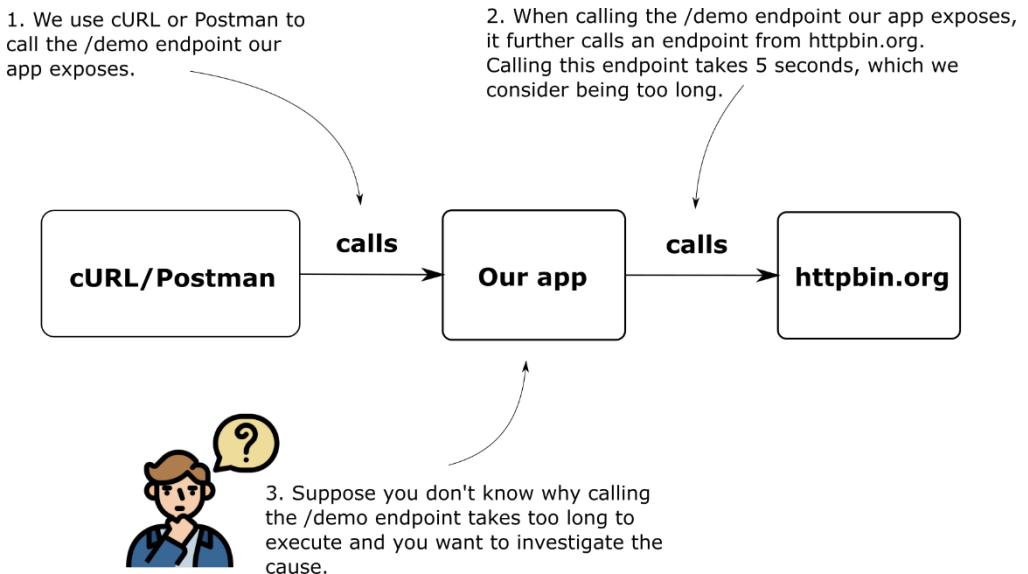


Figure 7.1 The app we investigate exposes an endpoint /demo. When someone calls this endpoint, they need to wait for 5 seconds for the app to respond back. We need to investigate this scenario to find out what takes so long for the endpoint to respond. We know our app calls a mock endpoint from httpbin.org that causes the delay, but we need to learn how to investigate this scenario with a profiler. This way, you'll know how to use similar techniques for similar real-world situations.

The profiling approach of an app has two steps:

1. **Sampling** to find out what code executes and where you should go more in detail (the approach we discuss in this section).
2. **Profiling (also named “instrumentation” sometimes)** to get more details about the execution of specific pieces of code.

Sometimes, step 1 (sampling) is enough to understand everything you need about a problem. So you might have cases where you don't necessarily need to get more details by profiling the app. As you'll learn in this chapter and chapters 8 through 10, step 2 (profiling)

can give you more details about the execution if you need. But you need to know what part of the code to profile before, and for that, you use sampling.

How does the problem occur in our example? When calling the /demo endpoint, the execution takes 5 seconds (figure 7.2), which we consider too long. Suppose that ideally, we would like the execution to take less than 1 second. We want to understand why calling the /demo endpoint takes so long – who causes the latency? It's our app, or something else that causes it to wait for this amount of time? Suppose, of course, that you don't already know the cause.

In similar cases, when you want to investigate a slowness problem in an unknown code base, using a profiler should be your first choice. The problem doesn't necessarily need to involve an endpoint. For this example, an endpoint was the easiest solution to implementing a simple app that allows you to focus on the investigation technique we discuss. But any situation where something is slow: calling an endpoint, executing a process, or a simple method call upon a particular event classify in this range of issues for which you should choose a profiler as your first option.

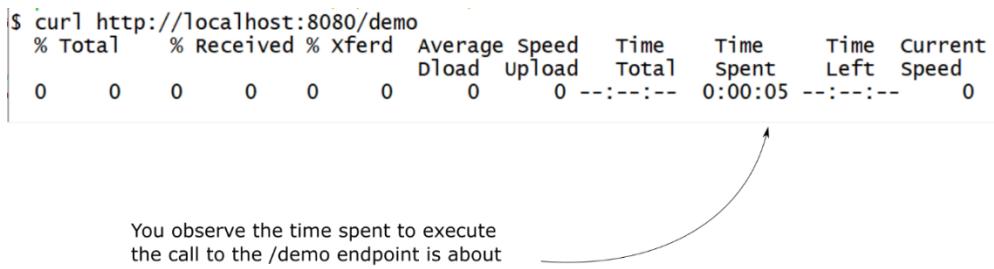


Figure 7.2 Someone calling the endpoint (in this figure using cURL) waits for about 5 seconds for the app to respond. In our scenario, this is the latency problem we investigate using a profiler.

We start the app and then VisualVM (the profiler we use for our investigations). Remember to add the VM option `-Djava.rmi.server.hostname=localhost`, as we discussed in chapter 6. That allows VisualVM to connect to the process. You select the process from the list on the left and then select the **Sampler** tab as presented in figure 7.3 to start sampling the execution.

Once you selected the process you are investigating from the left side of the window, you open the Sampler tab to sample the app's execution.

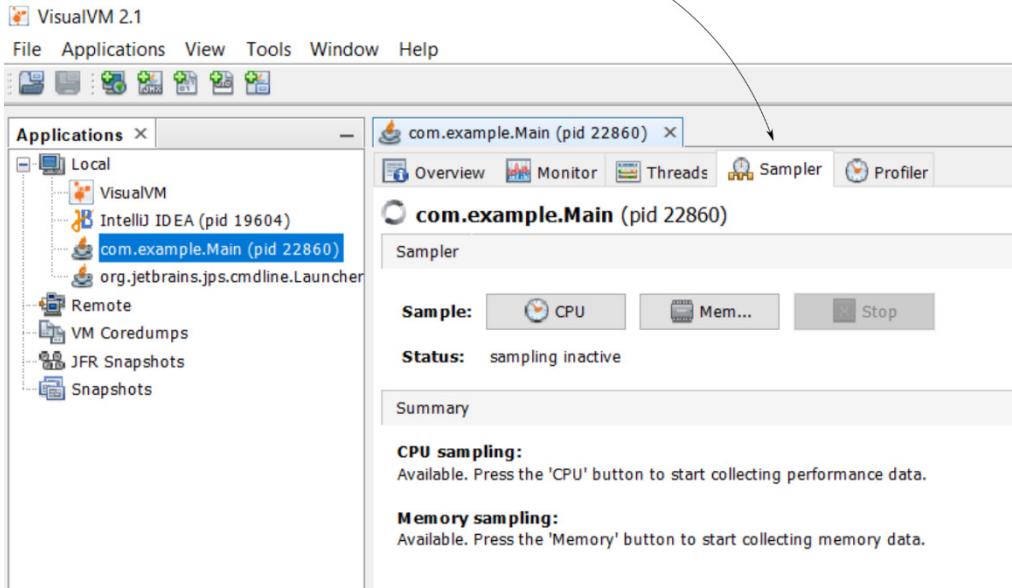


Figure 7.3 To start sampling the execution, select the process from the list on the left side and then the Sampler tab.

You sample the execution for three purposes:

- 1. Find out what code executes – sampling shows you what executes behind the scenes to it's an excellent way to find out what part of the app your need to investigate when you don't know exactly what happens during execution.**
- 2. Identify CPU consumption** – this is what we'll use to investigate latency issues and understand how methods share the execution time
- 3. Identify memory consumption** – to analyze memory-related issues. We'll discuss more sampling and profiling memory in chapter 11.

Select **CPU** (as shown in figure 7.4) to start sampling for performance data. Doing so, VisualVM displays a list of all the active threads and their stack traces. The profiler now intercepts the process execution and displays all the methods called and the approximate execution time. When you call the /demo endpoint, the profiler shows what happens behind the scenes while the app executes that capability.

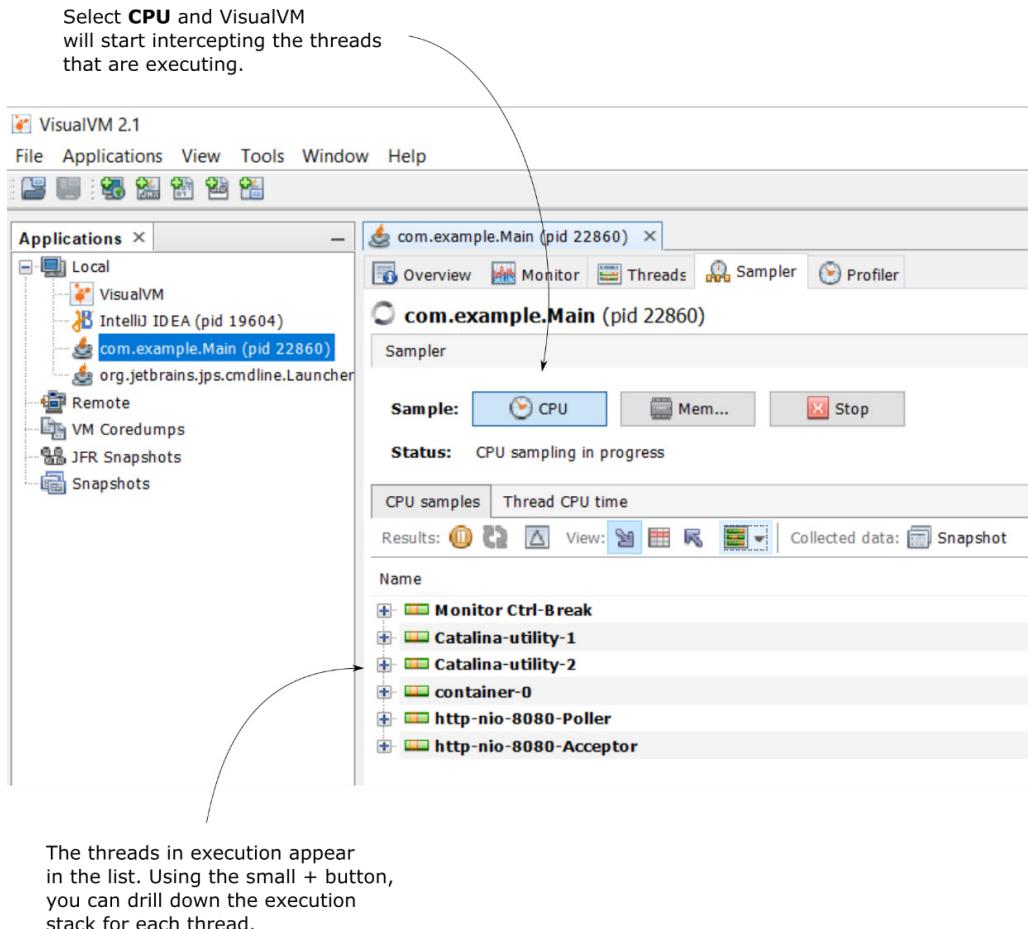
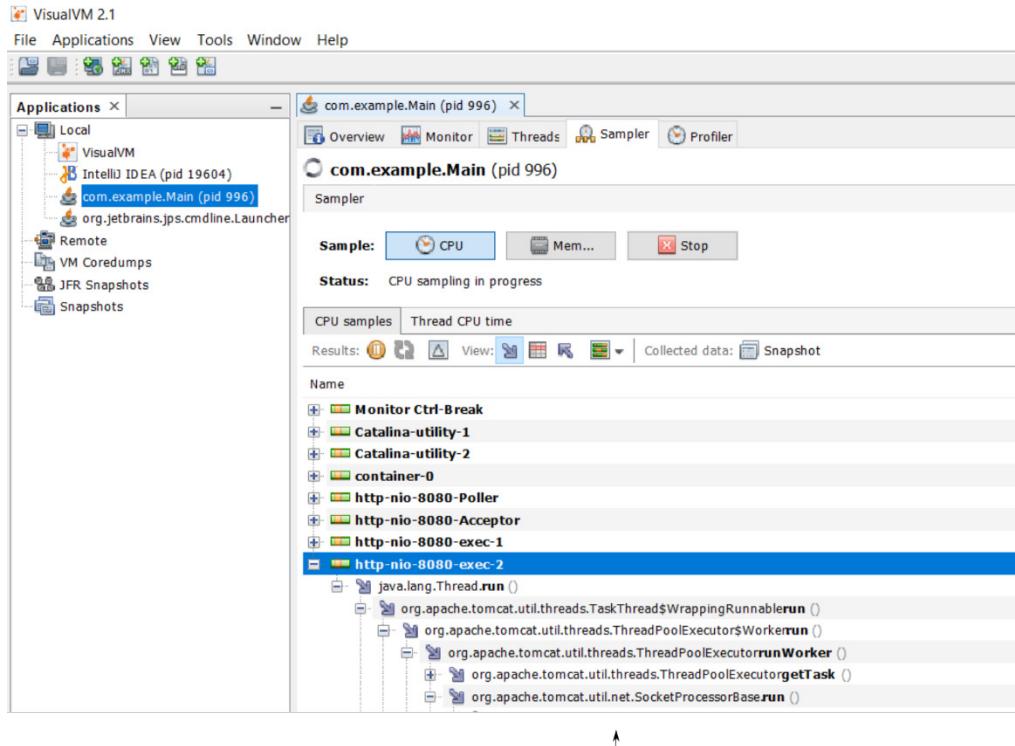


Figure 7.4 The profiler shows all the active threads in a list. You can expand each item to see the execution stack and an approximate execution time. While the app executes, the newly created threads will appear in the list, and you can analyze their execution.

We can now call the /demo endpoint and observe what happens. As also shown in figure 7.5, some new threads appear in the list. The app started these threads when we called the /demo endpoint. Opening them, you should observe precisely what the app does during its execution.

Before I go even deeper and discuss details such as the execution time, I want to highlight how vital this part already is. In many situations where I had to analyze code, I only used sampling to figure out where to look. I might not even have to investigate a performance or latency issue, but just find out where to start debugging. Remember our discussions in chapters 2 through 4. To debug something, you need to know where to add that breakpoint to pause the app's execution. Suppose you have no clue where to add a

breakpoint – then you can't debug. Sampling can be a way to shed some light in a situation where you can't figure out where to start debugging (especially in cases like the ones in my story at the beginning of the chapter about apps lacking completely a code design).



VisualVM reveals the full stack trace the app executed when you called the /demo endpoint. You can use the stack trace to identify which code the app executed, and, which instructions spent more time executing.

Figure 7.5 The stack traces show what the app executes. You observe every method and what other method it called further. This view helps you quickly find the code you focus on when investigating an issue on a certain capability.

Let's have a look now at the execution stack and understand what the profiler shows us. When you want to figure out what code executes, you simply expand the stack trace up to the point where it shows the methods of the app you are interested in. When you investigate a latency problem (as in this example), you expand the stack trace following the maximum execution time. In figure 7.6, you can observe how the profiler shows the execution time.

I have expanded the execution stack by selecting the small (+) button to the last method. The tool shows the time spent was about 5 seconds to understand the execution and find which method causes the latency.

In this particular case, we observe that only one method causes the slowness. This method is `getResponseCode()` of the `HttpURLConnection` class.

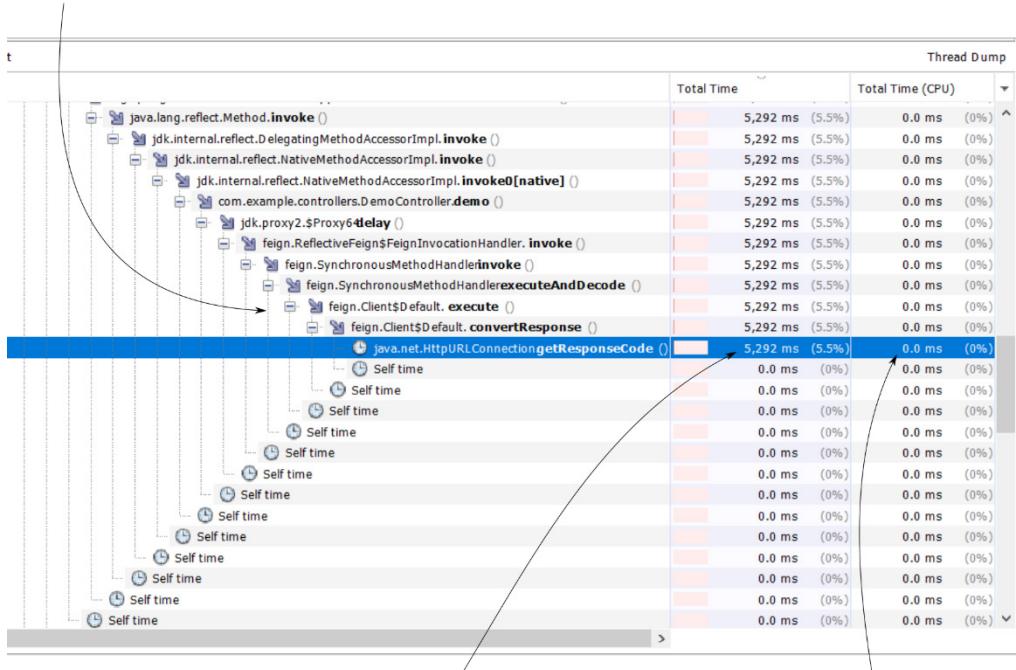


Remember that it's not always only one method that spends all the execution time in real-world scenarios. You'll often find that the time spent is shared among multiple methods that execute. But the rule is to focus first on the method that takes the longest time to execute.

An important aspect of this example is that the CPU time (how long the method works) is 0, even if the method spends 5 seconds in execution. The reason is that the method waits for something, and it doesn't spend CPU resources. This method waits for the HTTP call to end and to get a response. We can conclude that the problem is not in the app, since it's slow only because it waits for a response for its HTTP request.

It's extremely valuable to differentiate between the total CPU time and the total execution time. If a method spends CPU time, it means the method "works", so to improve the performance in such a case, you'd usually have to adjust (if possible) the algorithm to minimize its complexity. If the execution spends a small CPU time, but has a large execution time, the method waits for something instead. So it can be that an action takes a long time, but the app doesn't do anything. In such a case, you need to figure out what your app is waiting for.

The profiler doesn't intercept only your app's code base, but also code from frameworks and libraries the app uses.



The tool shows the total time spent by each method call. You can use this information to identify the root causes for app slowness. In this case, a method `getResponseCode()` from class `HttpURLConnection` spent all the execution time.

Another essential detail to observe here is that the CPU spent time is 0. This means that the app wasted the 5 seconds total execution time to wait for something rather than work on something.

Figure 7.6 Expanding the execution stack, you find which methods execute and how much time they spend executing. You can also deduce how much time they wait and how much they do work. The profiler shows both the app's codebase methods and the methods called from specific dependencies (libraries or frameworks) the app uses.

Another essential aspect to observe is that the profiler doesn't only intercept your app's codebase. You can see here also the dependencies' methods that are called during the app's execution. In this example, the app uses a dependency named OpenFeign to call the httpbin.org endpoint. You can observe in the stack trace packages that don't belong to your app's codebase. These packages are part of the dependencies your app uses to implement its capabilities. OpenFeign can be one of them, like in this example.

OpenFeign is a project part of the Spring ecosystem of technologies that a Spring app can use to call REST endpoints. Since this example is a Spring app, you will find packages of Spring-related technologies in the stack trace. No worries, you don't have to understand what each part of the stack trace does. Neither you'll know in a real-world scenario in all the

cases. In fact, this book is about understanding the code that you don't know yet, and, as discussed starting with chapter 1, one of the ways of using the techniques you learn in this book is helping you learn faster a new technology. On the other hand, if you want to learn Spring, I recommend starting with Spring Start Here (Manning 2021), another book I wrote. You'll also find in Spring Start Here details about using OpenFeign.

But, why is this point of observing dependencies' methods so important? Because sometimes, it's almost impossible to figure out what executes from a given dependency using other means. Take a look at the code written in our app to call the httpbin.org endpoint (listing 7.1). You can't see anywhere the actual implementation for sending the HTTP request. That's because, as it happens in many Java frameworks today, the dependency uses dynamic proxies to decouple the implementation.

Listing 7.1 The HTTP client implementation using OpenFeign

```
@FeignClient(name = "httpBin", url = "${httpBinUrl}")
public interface DemoProxy {
    @PostMapping("/delay/{n}")
    void delay(@PathVariable int n);
}
```

Dynamic proxies give an app a way to choose a method implementation at runtime. When an app capability uses dynamic proxies, it might actually call a method declared by an interface without knowing what implementation it will be given to execute at runtime (figure 7.7).

It is easier to use the framework's capabilities, but the disadvantage is that you don't know where to investigate an issue.

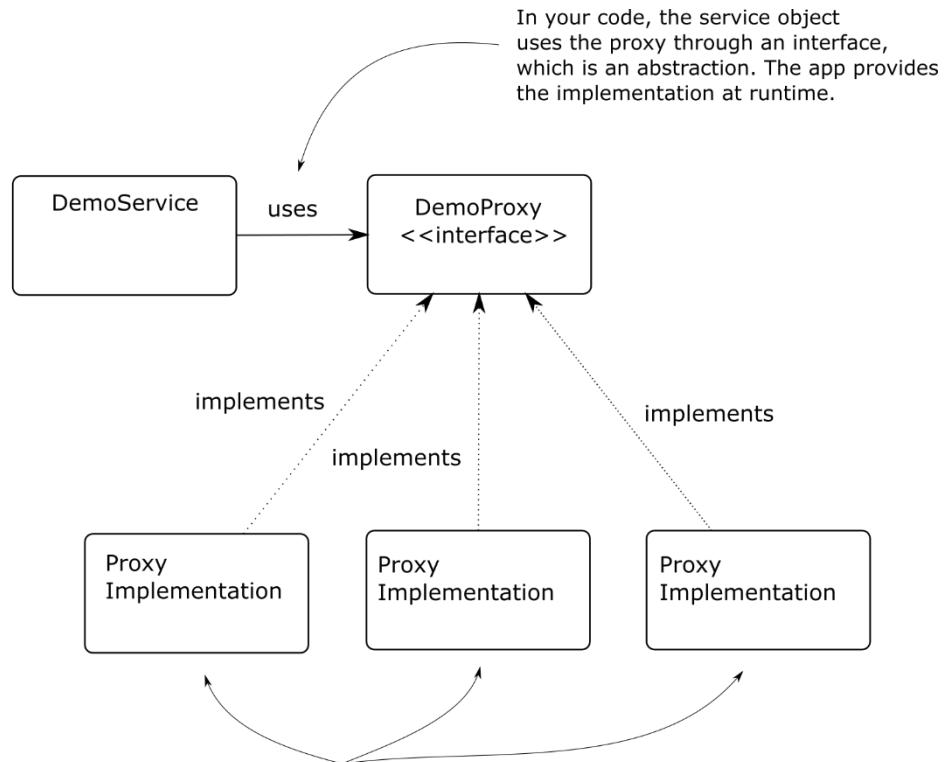


Figure 7.7 The framework keeps the implementations for an abstraction separate and provides them dynamically during execution. Because the implementation is decoupled and the app provides it during runtime, it's more difficult to find it by reading the code.



One of my personal uses cases for sampling is when **learning a new framework or library**. Sampling helps me understand what executes behind the scenes in a new functionality I learn to use. I applied this approach when learning Hibernate and Spring Security, which have complex functionality, and the approach helped me understand much faster how to work with the given capabilities.

7.2 Profiling to learn how many times a method executed

Finding what code executes is essential, but sometimes not enough. Often, we need to get more details to understand precisely a given behavior. For example, one of the essential details you'll miss when sampling is the number of method invocations. You might have a method that takes only 50 milliseconds to execute, but if the app calls it a thousand times, then you will observe it takes 50 seconds to execute when sampling. To demonstrate getting these details about the execution with a profiler and situations where this is useful, we'll again use some projects provided with the book.

We'll start with project da-ch7-ex1, which we also used in section 7.1, but this time to discuss profiling for details about the execution.

Start the app provided with project da-ch7-ex1. One crucial detail is that when you profile an app, you shouldn't investigate the entire codebase. Instead, you need to filter only on what's essential to your investigation. Profiling is a very resource-consuming operation, so trying to profile everything unless you have a really powerful system would take a ton of time. That's one more reason why we always start with sampling – to identify what exactly to profile further if needed.



Never profile the entire app's codebase. You should always decide first, based on sampling, which part of the app you want to get more details on by profiling it.

For this example, we'll ignore the app's codebase (without dependencies) and only take OpenFeign classes from the dependencies. Be aware that you can't usually refer to all the app's code in a real-world app since the app might be extensive. For this small example, won't be a problem but for large apps always restrict to as much as possible the intercepted code when profiling.

In figure 7.8, you find out how to apply these restrictions. On the right side of the **Profiler** tab, you can specify which part of the app is intercepted. In this example, we use

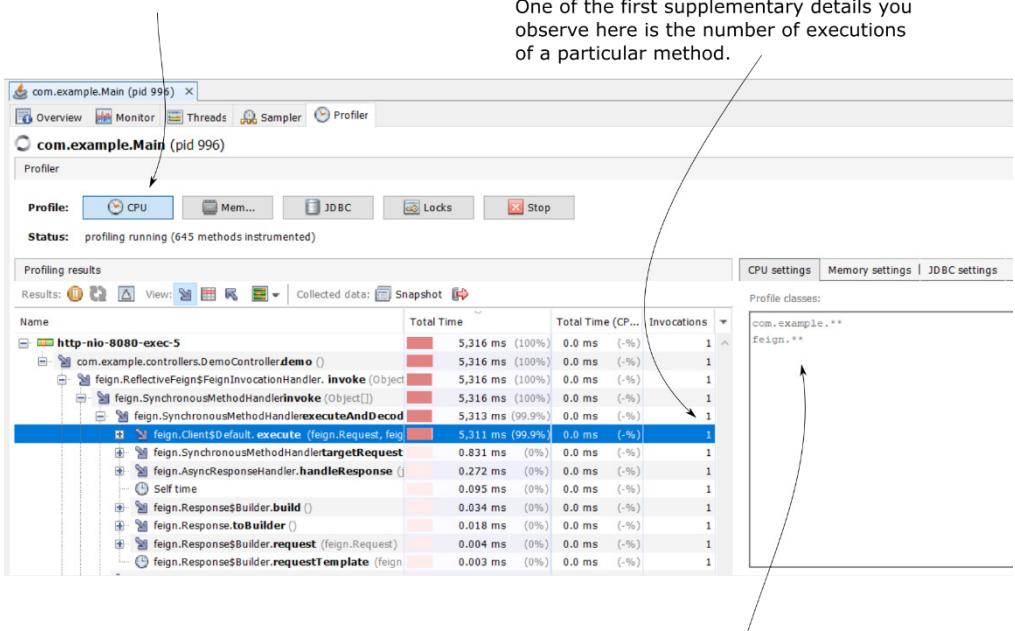
- com.example.** - meaning code in all the packages and subpackages of com.example
- feign.** - meaning code in all the packages and subpackages of feign

The syntax you can use to filter the packages and classes you want to profile has just a few simple rules:

1. Write each rule on a separate line.
2. Use one * to refer to a package – for example, we could have used com.example.* if we wanted to mean that we want to profile all classes in package “com.example”.
3. Use two * to refer to a package and all its subpackages – as we do in this case, using com.example.** we mean all classes in package “com.example” as well as any of its subpackages.
4. Write the full name of a class if you want to profile only that class – for example, we could have used com.example.controllers.DemoController to only profile this class.

I chose these packages after sampling the execution as discussed in section 7.1. Because I observed that the call of the method with the latency problem comes from classes of the feign package, I decided to add this package and its subpackages to the list, to get more info about it.

Select **CPU** to start profiling the app.



Always profile only a small number of packages. Before starting to profile the execution, define the filters to tell the tool which classes need to be intercepted.

Figure 7.8 Profiling a part of the app during execution to get details about the times a given method was invoked. We observe that the method causing the 5 seconds latency is invoked only once, meaning the number of invocations doesn't cause a problem here.

In this particular case, the number of invocations doesn't seem to cause issues – the method executes only once and takes about 5 seconds to finish its execution. A small number of method invocations implies at least that we don't have repeated unnecessary executions (which, as you'll learn later in this chapter is a common problem in many apps).

In another scenario, you might have observed that the call to the given endpoint only takes 1 second, but the method is (because of some lousy design) called five times. Then, the problem would have been in the app, and we knew what we needed to solve and where. In section 7.3, we'll also analyze such a problem having as a cause multiple calls to a method.

7.3 Using a profiler to identify SQL queries an app executes

In this section, you'll learn how to identify SQL queries an app sends to a DBMS using a profiler. This subject is by far one of my favorites. Today, almost every app uses at least one relational database, and almost all scenarios encounter latencies caused by SQL queries from

time to time. Moreover, the apps use fancier ways to implement the persistence layer today: in many cases, the SQL queries the app sends are created dynamically by a framework and a library. These dynamically generated queries are hard to identify, but a profiler can do some magic and simplify your investigation a lot.

We'll use a scenario implemented with project da-ch7-ex2. This app runs some queries on a relational database. With this example, you'll learn how to observe how many times a method executes and intercepts a SQL query the app runs. We'll then demonstrate that the executed SQL queries can be retrieved even when the app works with a framework and doesn't handle the queries directly – we'll discuss this subject later with a couple of examples.

7.3.1 Using a profiler to retrieve SQL queries not generated by a framework

This section uses an example to demonstrate how to use a profiler to obtain the SQL queries an app executes. In this example, we'll use a simple app that sends the queries to a Database Management System (DBMS) directly without using any framework. Once this example is clear, we'll demonstrate that this technique is even more powerful in the next sections. You can also apply it when the app uses particular frameworks to work with the data in a database.

Let's start project da-ch7-ex2 and use the profiler tab as you learned in section 7.2. Project da-ch7-ex2 is also a small app. It configures an in-memory database with two tables (product and purchase) and populates the tables with a few records.

The app exposes all purchased products when calling the endpoint /products. By "purchased products", I mean products that have at least one purchase record in the purchase table. The exercise is to analyze the app's behavior when calling this endpoint without analyzing the code first. This way, we'll see how much we can get just by using the profiler.

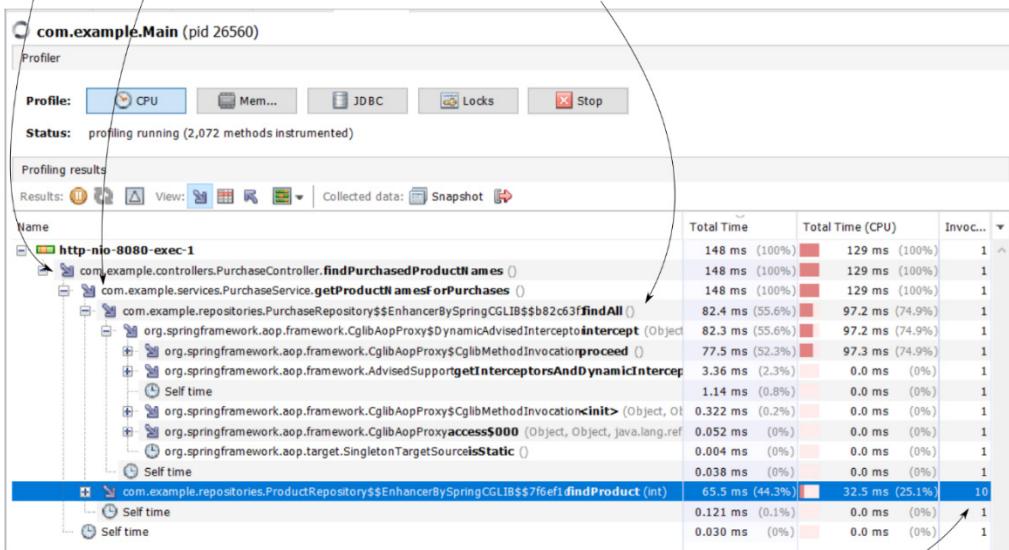
In figure 7.9, we directly use the profiler tab since you already learned sampling in section 7.1, but remember that in any real-world scenario, you start with sampling. We start the app, and using cURL or Postman, and we call the /products endpoint. The profiler shows us precisely what happens when calling the /products endpoint:

1. A method `findPurchasedProductNames()` that belongs to the `PurchaseController` class was called.
2. This method delegated the call to a method `getProductNamesForPurchases()` in class `PurchaseService`.
3. The method `getProductNamesForPurchases()` in `ProductService` calls `findAll()` in `PurchaseRepository`.
4. The method `getProductNamesForPurchases()` in `ProductService` calls `findProduct()` in `ProductRepository` 10 times.

1. The execution starts with the `findPurchasedProductNames()` method in the `PurchaseController` class.

2. Further, the `getProductNamesForPurchases()` method in `PurchaseService` is called.

3. The method in `PurchaseService` class calls `findAll()` in `PurchaseRepository`.



4. After calling `findAll()` in `PurchaseRepository`, the method calls `findProduct()` in `ProductRepository` 10 times.

Figure 7.9 Profiling the app, we observe that one of the methods is called 10 times. We now need to ask ourselves if this is a design issue. We have the big picture over the entire algorithm, and since we know what code is executed, we could also debug it if we can't figure out straightforwardly what happens.

Isn't this amazing? We didn't even look into the code, and we already know a lot of things about this execution. These details are fantastic, because now, you know exactly where to go into the code and also what you expect to find. The profiler gave you class names, method names, and how they call each other. Let's now take a look into the code and figure out where all this happens. Using the profiler, we can see that most things happen in the `getProductNamesForPurchases()` method in `PurchaseService` class, so that's most likely the place we need to analyze. Listing 7.2 shows the `PurchaseService` class that contains the algorithm's implementation.

Listing 7.2 The algorithm's implementation in the PurchaseService class

```

@Service
public class PurchaseService {

    private final ProductRepository productRepository;
    private final PurchaseRepository purchaseRepository;

    public PurchaseService(ProductRepository productRepository,
                          PurchaseRepository purchaseRepository) {
        this.productRepository = productRepository;
        this.purchaseRepository = purchaseRepository;
    }

    public Set<String> getProductNamesForPurchases() {
        Set<String> productNames = new HashSet<>();
        List<Purchase> purchases = purchaseRepository.findAll();      #A
        for (Purchase p : purchases) {      #B
            Product product =
                productRepository.findProduct(p.getProduct());      #C
            productNames.add(product.getName());      #D
        }
        return productNames;      #E
    }
}

```

#A Getting all the purchases from the database table.

#B Iterating through each product.

#C Getting the details about the purchased product.

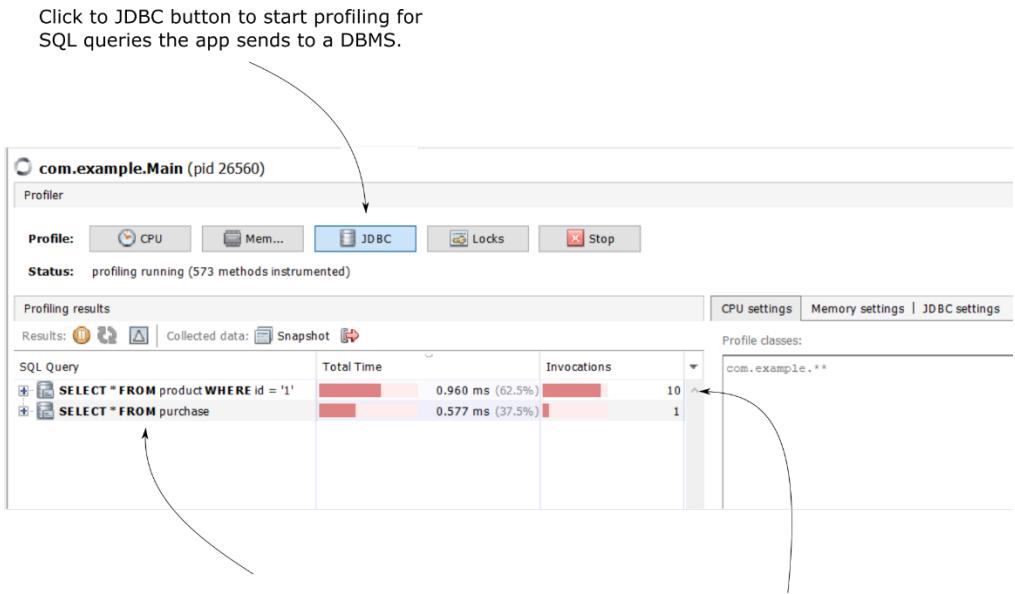
#D Adding the product into a set.

#E Returning the set of products.

Observe the implemented behavior: the app fetches some data in a list and then iterates over it to get more data from the database. Whenever I find such an implementation where some data is retrieved from a database table, the app iterates over it and gets more data from another table usually indicates a design issue. The reason is you can usually reduce the execution of so many queries to only one. Obviously, the fewer queries executed, the more efficient the app is.

In this example, it's effortless to retrieve the queries directly from the code. Since the profiler shows us exactly where they are, and the app is tiny, it won't be a problem finding the queries. But real-world apps are not small, and in many cases, it's not easy to retrieve the query from the code directly. But fear no more! You can use the profiler to retrieve all the SQL queries the app sends to a DBMS.

You find this demonstration in figure 7.10. Instead of selecting the CPU button, you now select the JDBC button to start profiling for SQL queries.



When the app sends an SQL query to a DBMS, the profiler intercepts it and shows it in this list. The SQL query appears complete, including the parameters' values.

We can observe this query executed 10 times. Usually, we avoid running the same query multiple times to improve the app's performance.

Figure 7.10 The profiler intercepts the SQL queries the app sends to the DBMS through the JDBC driver. This capability provides you an easy way to get the queries and run them, observe what part of the codebase runs them and how many times a query is executed.

What the tool does behind the scenes is pretty simple. Any Java app sends the SQL queries to a DBMS through a JDBC driver. The profiler intercepts the driver and copies the queries before the driver sends them to the Database Management System (DBMS). Figure 7.11 shows visually this approach. The result is fantastic, as you can simply copy-paste the queries in your database client where you can run them or investigate their explain plan.

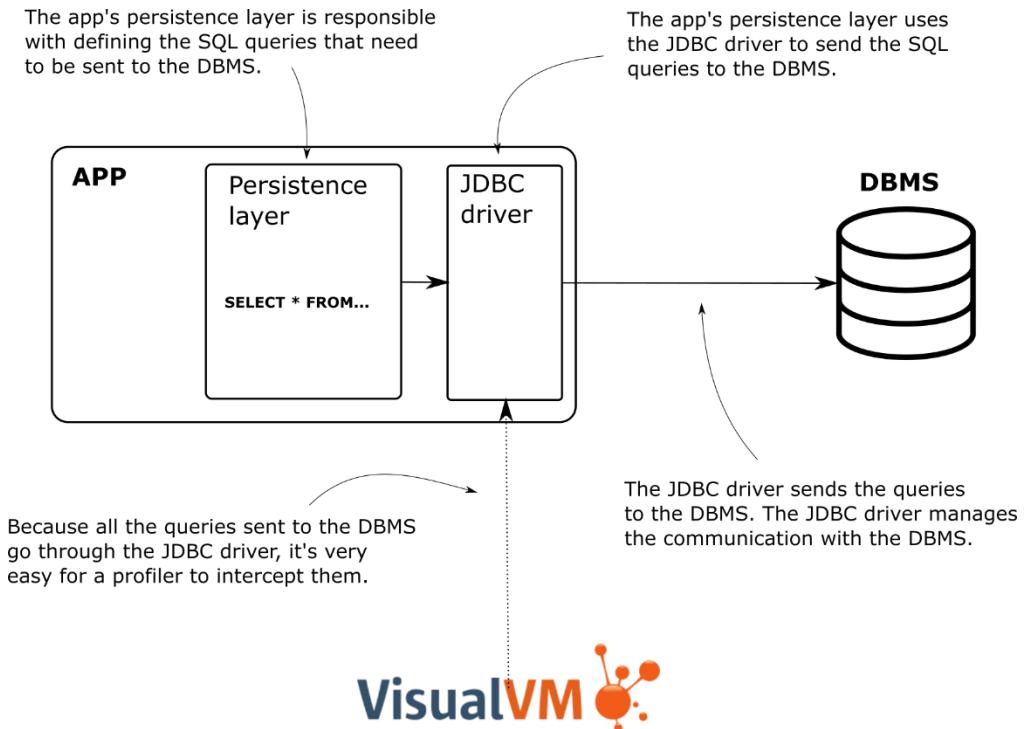


Figure 7.11 In a Java app, the communication with a relational DBMS is done through the JDBC driver. A profiler can intercept all method calls, including those of the JDBC driver, and retrieve the SQL queries the app sends to a DBMS. You can get the queries and use them in your investigations.

The profiler also shows you how many times a query was sent. In this case, the app sent the first query ten times. This design is definitely a faulty one since it repeats the same query multiple times, spending unnecessary time and resources. The developer who implemented the code tried to obtain the purchases and then get the product details for each purchase. But a straightforward query with a join between the two tables ("product" and "purchase") would have solved the problem in one step. Fortunately, using VisualVM, you identified the cause, and you know exactly what to change to improve this app.

Figure 7.12 shows you how to find the part of the codebase that sent the query. You can expand the execution stack and usually find the first method in the app's codebase.

Clicking on the small + button shows the full stack trace that caused the executing of a certain SQL query.



In the stack trace, you find the methods in the app's code base that caused the execution of a certain query. This way, you identify where the problem comes from in your app.

Figure 7.12 For each query, the profiler also provides the execution stack trace. You can use the stack trace to identify which part of your app's codebase sent the query.

Listing 7.2 shows the code behind the scenes whose call we identified using the profiler. Once you identify where the problem comes from, it's time to read the code and find a way to optimize the implementation. In this example, everything could have been merged in only one query. It may look like a dummy mistake, but trust me, you'll find such cases even in larger apps implemented by powerful organizations.

Listing 7.2 The algorithm's implementation in the ProductService class

```

@Service
public class PurchaseService {

    // Omitted code

    public Set<String> getProductNamesForPurchases() {
        Set<String> productNames = new HashSet<>();
        List<Purchase> purchases = purchaseRepository.findAll();      #A
        for (Purchase p : purchases) {          #B
            Product product = productRepository.findProduct(p.getProduct());      #C
            productNames.add(product.getName());
        }
        return productNames;
    }
}

```

#A The app gets a list of all products.

#B Iterates through each product.

#C Gets the product details.

Example da-ch7-ex2 is using JDBC to send the SQL queries to a DBMS. The app has the SQL queries in their native shape directly in the Java code (listing 7.3), so, somehow, you could say that copying the queries directly from the code is not that difficult. But, in today's apps, you'll encounter less often the use of native queries in the code. Today, many apps use frameworks such as Hibernate (the most used Java Persistence API [JPA] implementation) or Java Object Oriented Querying (JOOQ). You find more details about JOOQ on their GitHub repository here: <https://github.com/jOOQ/jOOQ>. What's different with these frameworks is that you won't find the native queries directly in the code anymore (native queries can be used with these frameworks, but it's quite unusual and mostly not recommended to use them).

Listing 7.3 The repositories use native SQL queries

```

@Repository
public class ProductRepository {

    private final JdbcTemplate jdbcTemplate;

    public ProductRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public Product findProduct(int id) {
        String sql = "SELECT * FROM product WHERE id = ?";      #A
        return jdbcTemplate.queryForObject(sql, new ProductRowMapper(), id);
    }
}

```

#A A native SQL query the app sends to the DBMS.

7.3.2 Using the profiler to get the SQL queries generated by a framework

Now, let me show you something even more extraordinary. To prove the usefulness of a profiler even more in investigating SQL queries, let's investigate project da-ch7-ex3. From an algorithm point of view, this project does the same thing as the previous project did – returns the name of the purchased products. I intentionally kept the same logic to simplify understanding the example and be able to make a comparison.

Take a look at listing 7.4, which shows the definition of a Spring Data JPA repository. The repository is now a simple interface, and you don't find the SQL queries anywhere. With Spring Data JPA, the app generates the queries behind the scenes either based on method's names or on a particular way to define the queries, named Java Persistence Query Language (JPQL), based on app's objects. Either way, there's no simple way to copy-paste the query from the code anymore.

Some frameworks generate the SQL queries behind the scenes based on the code and configurations you write. In these cases, it's particularly more challenging to get the executed queries. But a profiler helps you get the queries by extracting them from the JDBC driver before they are sent to the DBMS.

Listing 7.4 The Spring Data repository doesn't use native SQL queries

```
public interface ProductRepository
    extends JpaRepository<Product, Integer> { }
```

But the profiler comes to the rescue. Since the tool intercepts the queries before the app sends them to the DBMS, we can still use it to find exactly what queries the app uses. Start app da-ch7-ex3 and use VisualVM to profile the SQL queries same as we did for the previous two projects.

Figure 7.13 shows you what the tool displays when profiling the /products endpoint call. The app sent two SQL queries. Observe that the aliases in the query have strange names because the queries are framework generated. Also, observe an interesting aspect: even if the logic in the service is the same, and the app calls the repository method ten times, the second query is executed only once. This behavior happens because Hibernate optimizes the execution where it can. Now you can copy and investigate this query with an SQL development client if needed. In many cases, investigating a slow query requires running it in a SQL client to observe which part of the query puts the DBMS in difficulty.

The query is executed only once even if the method is called 10 times. Do these persistence frameworks usually do these kinds of tricks? Persistence frameworks are smart, but sometimes what they do behind the scenes can also add complexity. Someone not understanding properly the framework could write code that causes problems. This is another reason to use a profiler to check the queries the framework generated and make sure the app does what you expected to happen.

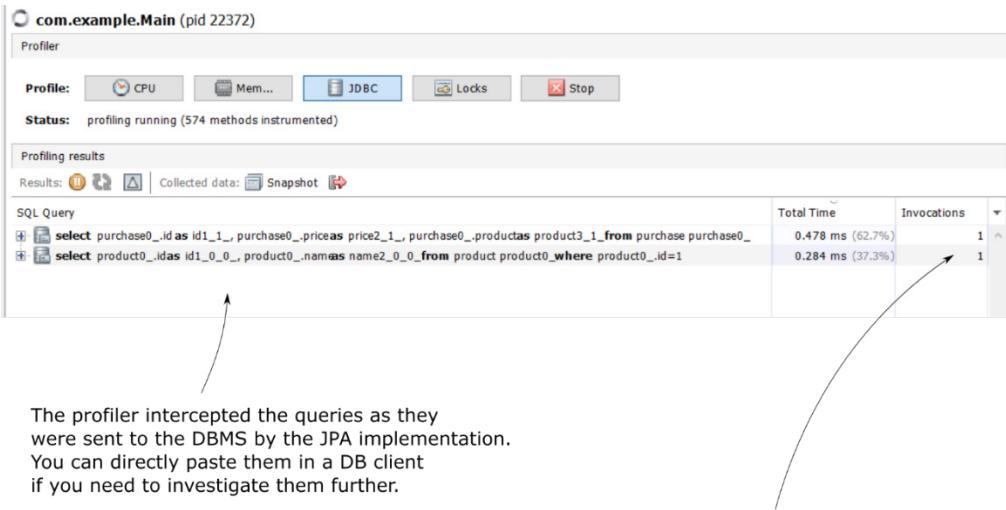


Figure 7.13 Even when working with a framework, the profiler can still intercept the SQL queries. This aspect makes your investigation a lot easier since you can't copy the query from the code directly as you may do when directly using JDBC and native queries.

The issues that I mostly encountered with frameworks and had to investigate are:

1. **Slow queries causing latencies** – easy to spot using the execution time with a profiler.
2. **Multiple unneeded queries generated by the framework** (usually caused by what the developers call the N+1 query problem) – easy to spot using the number of execution of a query with a profiler.
3. **Long transaction commits generated by poor app design** - easy to spot using CPU profiling.

When a framework needs to get data from multiple tables, it usually knows to compose one query and get all the needed data in one call. However, if you don't correctly use the framework it might take only a part of the data with an initial query, and, then, for each record initially retrieved it runs a separate query. So, instead of running only one query, the framework will send an initial one plus N others (one for each of the N records retrieved by the first) – we call this an **N+1 query problem**. An N+1 query problem usually causes significant latency by executing many queries instead of only one.

I observed that most developers seem to be tempted to work with logs or the debugger for investigating such problems. But also, in my experience, logs or the debugger are not the best options for such problems to identify the problem's root cause.

The first problem of using the logs for this case is that it's challenging to identify which query causes a problem. In real-world scenarios, the app may send dozens of queries – some of these multiple times and in most cases long and using a large number of parameters. With the profiler, which displays all the queries in a list together with their execution time and the number of executions, you can almost instantaneously spot the problem in most cases. The second difficulty is that even if you identify the potential query causing the problem (say you observe a query the app takes long to execute a given query while monitoring the logs), it's not straightforward taking the query and running it. In the log, you'll find parameters separated from the query.

You can configure your app to print the queries generated by Hibernate in the logs by adding some parameters to the application.properties of app da-ch7-ex3 file as shown in the next snippet.

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
logging.level.org.hibernate.type.descriptor.sql=trace
```

Beware that you'll have to use different ways to configure the logging depending on what technologies you use to implement the app. In the example provided with the book, we use Spring Boot and Hibernate. The next snippet shows how the app prints the query in the logs.

```
Hibernate:
  Select      #A
    product0_.id as id1_0_0_,
    product0_.name as name2_0_0_
  from
    product product0_
  where
    product0_.id=?
```

```
2021-10-16 13:57:26.566 TRACE 9512 --- [nio-8080-exec-2]
  o.h.type.descriptor.sql.BasicBinder      : binding parameter [1] as [INTEGER] - [1]
  #B
2021-10-16 13:57:26.568 TRACE 9512 --- [nio-8080-exec-2]
  o.h.type.descriptor.sql.BasicExtractor   : extracted value ([name2_0_0_] :
  [VARCHAR]) - [Chocolate]    #C
```

#A The query generated by the app.

#B The first parameter's value.

#C The second parameter's value.

The logs show us the query and give us the input and even the output of the query. But you need to bind yourself the parameter values to the query if you need to run it separately. And, when multiple queries are logged, looking for what you need might be really frustrating. Logs also don't show you straightforwardly which part of the app runs the query, which might make your investigation even more challenging.



I recommend you always start with a profiler when investigating latency issues. Your first step should be sampling. When you suspect SQL query-related problems, continue profiling for JDBC. From there, you'll find any problem straightforward to understand, and you can also use a debugger or the logs to enforce your speculations as needed.

7.3.3 Using the profiler to get programmatically generated SQL queries

For a complete reference, let's work on one more example that proves how a profiler works with scenarios where the app programmatically defines the queries. We'll investigate a performance problem with a query generated by Hibernate (the framework our example uses) in an app using criteria queries. Criteria queries are a programmatic way of defining the app's persistence layer with Hibernate. You never write a query with this approach, neither native nor JPQL.

As you observe in listing 7.5, which presents the `ProductRepository` class reimplemented with criteria query, this approach is more verbose. It's usually considered more difficult by developers and also leaves more room for mistakes. The implementation in project `da-ch7-ex4` contains a mistake, which can cause nasty performance problems in real-world apps. Let's see if we can find this issue together and how the profiler helps us understand what's wrong.

Listing 7.5 The repository defined with criteria query

```
public class ProductRepository {
    private final EntityManager entityManager;

    public ProductRepository(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);      #A
        Root<Product> product = cq.from(Product.class);           #B
        cq.select(product);          #C

        Predicate idPredicate = cb.equal(cq.from(Product.class).get("id"), id);      #D
        cq.where(idPredicate);          #E

        TypedQuery<Product> query = entityManager.createQuery(cq);
        return query.getSingleResult();      #F
    }
}
```

#A Creating a new query.

#B Specifying the query selects products.

#C Selecting the products.

#D Defining the condition that becomes part of the WHERE clause on the next line.

#E Defining the where clause

#F Running the query and extracting the result.

We use JDBC profiling to intercept the queries the app sends to the DBMS. Analyzing the query, you observe it contains a cross join between the product table and itself. This is a huge problem! With the ten records that our table contains, we don't observe anything suspicious here. But in a real-world app, where the table would definitely contain more records, this cross join would create huge latencies and eventually even wrong output (duplicated rows). But simply intercepting the query with VisualVM and reading it shows us the issue.

The query contains a useless cross join.
In a real-world app, this can cause
performance issues and even wrong
output behavior.

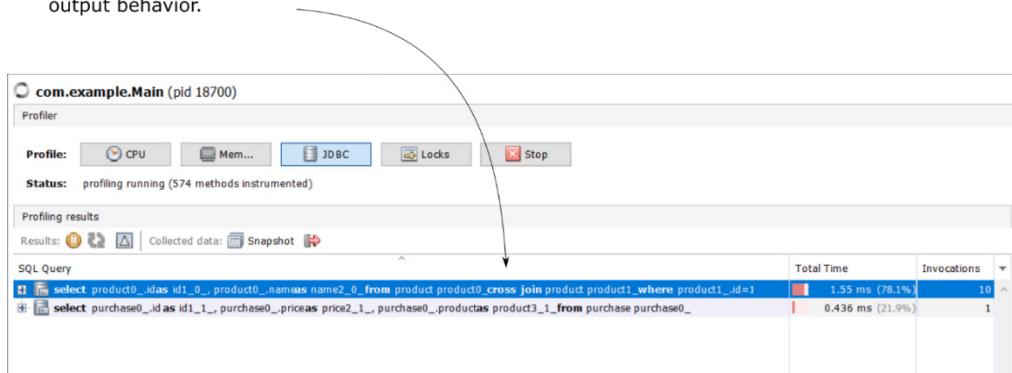


Figure 7.14 The profiler can intercept any SQL query sent to the DBMS through the JDBC driver. Here, we spot an issue in the generated query – an unneeded cross join that causes performance problems.

Of course, the next question is, why did the app generate the query this way? I like a statement about JPA implementation, such as Hibernate: “The excellent aspect is that they make the query generation transparent and minimize work. The bad aspect is that they make the query generation transparent making the app more prone to errors.” Generally, when working with such frameworks, I recommend developers profile the queries as part of the development process to discover such issues up-front. In such a case, using a profiler is more for auditing purposes than finding issues, but doing so is a good safety measure.

In this example, I intentionally introduced this tiny error with a significant impact. I called the `from()` method twice (listing 7.6), instructing Hibernate to make a cross join. I chose this example because I found this confusion among developers often when using criteria queries.

Listing 7.6 The cause of the cross join issue

```
public class ProductRepository {
    // Omitted code

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);

        Root<Product> product = cq.from(Product.class);      #A
        cq.select(product);

        Predicate idPredicate = cb.equal(cq.from(Product.class).get("id"), id);      #B
        cq.where(idPredicate);

        TypedQuery<Product> query = entityManager.createQuery(cq);
        return query.getSingleResult();
    }
}
```

#A Calling the CriteriaQuery from() method once

#B Calling the CriteriaQuery from() method again

Solving this problem is easy: use the product instance instead of calling the CriteriaQuery from() method the second time, as shown in listing 7.7.

Listing 7.7 Correcting the cross join issue

```
public class ProductRepository {
    // Omitted code

    public Product findById(int id) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Product> cq = cb.createQuery(Product.class);

        Root<Product> product = cq.from(Product.class);
        cq.select(product);

        Predicate idPredicate = cb.equal(product.get("id"), id);      #A
        cq.where(idPredicate);

        TypedQuery<Product> query = entityManager.createQuery(cq);
        return query.getSingleResult();
    }
}
```

#A Use the already existing Root object

Once you make this small change, the generated SQL query won't contain the unneeded cross join anymore (figure 7.15). Still, the fact that the app runs multiple times the same query is not optimal. The algorithm the app runs should be refactored to get the data using only one query call preferably.

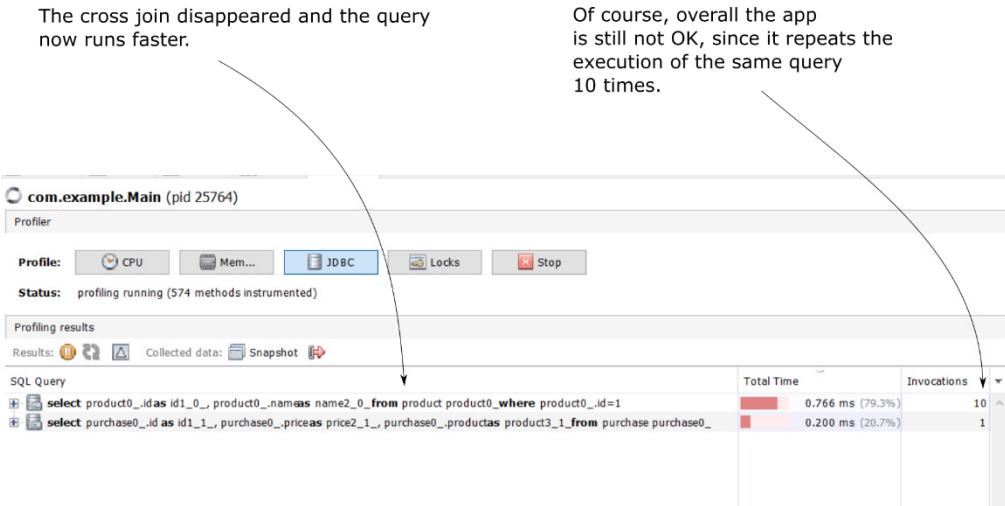


Figure 7.15 Eliminating the supplementary `select()` method call, the cross join disappeared. Overall, however, the algorithm for this app should be revised, since it still runs the same query multiple times, which is not optimal.

7.4 Summary

- A profiler intercepts the app's execution and provides essential details about the code in execution, such as the execution stack trace for each thread, how long it takes for each method to execute and how many times a certain method was called.
- When investigating latency problems, the first step to using a profiler is sampling. Sampling is a way the profiler intercepts the executing code without getting many details. Sampling is less resource-consuming and allows you to observe the big picture of execution over the whole process.
- Three essential details sampling gives you are
 - What code executes. Sometimes, when investigating an issue, you don't know what part of the code executes, and you can find this aspect by sampling.
 - Total execution time of every method. This detail helps you identify what part of the code causes potential latency problems.
 - Total CPU execution time. This detail helps you identify if your code spends the execution time "working" or waiting for something.
- Sometimes sampling is enough to understand where a problem comes from. But in many cases, you need more details about the execution. To get more details, you continue with profiling the execution.

- Profiling is a resource-consuming process. With a real-world app, it's almost always impossible to profile the whole codebase. For this reason, when profiling for details, you filter on specific packages and classes on which you want to focus your investigation. Usually, you understand what part of the app to focus on by sampling the execution first.
- An essential detail you get by profiling is the number of invocations of a method. When sampling, you know the total time a method spends executing, but not how often it was called. This aspect makes a difference between a method that is slow or wrongly used.
- You can use a profiler to get SQL queries the app sends to a DBMS. The profiler intercepts any queries, regardless of the technology used to implement the app's persistence layer. This tool's capability is invaluable when investigating slow queries for apps using frameworks (such as Hibernate) to work with a database.

8

Using advanced visualization tools for profiled data

This chapter covers

- Detecting problems with connections to relational databases
- Using call graphs to understand an app's design faster
- Using flame graphs to visualize an app's execution easier
- Analyzing queries an app sends to a NoSQL database server

In this chapter, we'll discuss valuable techniques that can make your life easier when you investigate specific scenarios. We'll start the chapter, in section 8.1, by examining an approach to identify connection problems between a Java app and a relational database server. We already discussed profiling SQL queries in chapter 7, but sometimes problems appear when an app establishes the communication with a database management system (DBMS). And such situations can even lead to an app not responding at all, which makes finding causes of such problems in real-world apps essential.

In section 8.2, I'll show you one of my favorite ways to understand the code behind a given execution scenario – a simple approach using call graphs. Call graphs are visual representations of the dependencies between an app's objects. I found call graphs helpful, especially when dealing with messy code I've never seen before. And since I'm sure most developers have to deal at some point in their careers with messy codebases, I consider knowing this approach will potentially save you plenty of time at some point.

Chapter 7 discussed one of the most used ways to visualize an app's execution – the execution stack. You learned how to generate an execution stack when sampling or profiling with VisualVM and identify execution latencies using it. In section 8.3, we'll use a different representation of the execution stack – a flame graph. Flame graphs are a way to visualize an app's execution that focuses on both the executed code as well as the execution time.

Sometimes, viewing the same data from an additional perspective helps you find what you're searching for more easily. As you'll learn in section 8.3, flame graphs offer you a different view of the app's execution, helping you identify potential latencies and performance problems.

In section 8.4, we'll discuss techniques for analyzing how an app works with its persistence layer when it doesn't use relational databases, but uses different persistent approaches from what we call "the NoSQL family of technologies".

For the subjects we discuss in this chapter, VisualVM is not enough. VisualVM is an excellent free tool that I use in more than 90% of the scenarios I need to investigate with a profiler. Still, VisualVM has its limitations.

To demonstrate the features we discuss in this chapter, we'll use JProfiler (<https://www.ej-technologies.com/products/jprofiler/overview.html>), a licensed profiler. JProfiler provides everything we discussed with VisualVM, but it also has capabilities VisualVM doesn't have (of course, everything for a small price). You can use the short trial period the software offers to try profiling the examples we use in this book with JProfiler and make your own opinion on the difference between these two tools. Also, you can use the short trial period to exercise the demonstrations in this chapter.

8.1 Detecting problems with JDBC connections

We discussed plenty of things about investigating problems with SQL queries in chapter 7. But what about the connection an app needs to establish first with a database management system (DBMS) to send the queries? It turns out negligence in the connection management can cause problems, and in this section, we discuss investigating such issues and finding their root causes.

Someone could argue that apps use frameworks and libraries that take care of the connection management in most cases, and such problems don't occur anymore. Experience, however, tells me that such problems still happen, mainly because developers rely on many things to be automatically taken care of by frameworks. And allowing frameworks to manage connections indeed works very well in most cases. However, sometimes we have to implement things that are less usual and require to go a bit more low-level than relying on something the framework offers, and that's where most problems such as the one we discuss in this section occur.

Let me tell you a story of an issue I had to deal with recently. In a particular service (implemented with Spring, a known and really appreciated framework today), developers had to implement a less common capability: a way to cancel the execution of a stored procedure (a procedure running at the database level). The implementation wasn't complex, but it required accessing the connection object directly. In most cases, it was enough to allow Spring to use the connection behind the scenes. Spring is a robust framework and easy to customize, and it straightforwardly enables you to access the connections it manages. However, the question is: does it still manage these connections after you access them directly? The answer is, sometimes it does. And this "sometimes" is what made things interesting (and also more challenging).

Soon, using the approaches you'll learn in this section, the developers found out that in a standard method execution where Spring manages the transactions, the framework also

closes the connections at the end. The process of canceling the procedure used a batching approach with a different project from the Spring family of technologies called Spring Batch. In such cases, the framework doesn't close the connections anymore, allowing you to manage them. The developers used the same approach in both cases and didn't realize that the connections were not correctly closed in one of the cases, which could have caused a big problem. Fortunately, the development error was found in time and caused no harm.

You might have never used Spring Batch, and didn't get all the details of this story. But don't worry, this chapter is not about Spring, Spring Batch, or any specific technology, so no worries if you don't have experience with any of these. This is just a story about something that really happened in the real world to give you some understanding of why this technique you'll learn is still relevant. It doesn't matter the name of the frameworks you use, you might never know everything that happens behind the scenes, so being prepared to investigate your app execution in any way will always be relevant.

To make the discussion straightforward, we'll use project da-ch8-ex1 provided with the book. This project defines a simple app with a huge problem: one of its methods "forgets" to close the JDBC connections it opens. An app creates a JDBC connection to send SQL queries to the DBMS. The JDBC connections must always be closed after the app doesn't need to use them any longer. Any DBMS offers to its client (the app) the possibility to get a limited number of connections (usually a small number such as 100). When an app opens all these connections but doesn't close them, it cannot connect to the database server (figure 8.1).

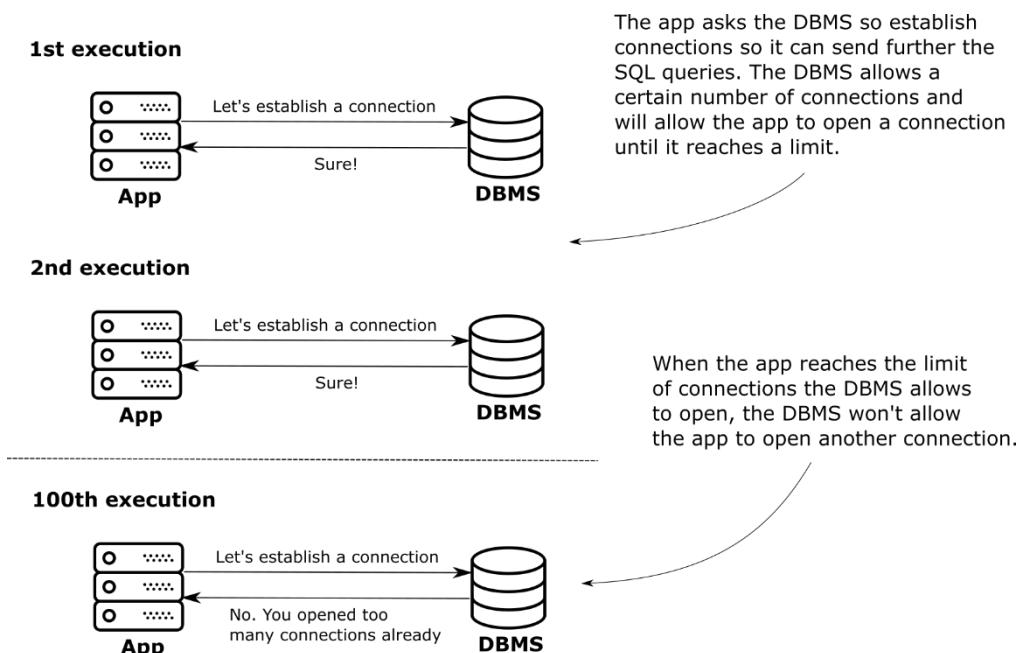


Figure 8.1 A DBMS only allows an app to open a finite and usually small number of connections. When the app reaches the limit of connections it can open, the DBMS doesn't allow the app to open other connections. In

such a case, the app could become unable to execute its work. An app should always close JDBC connections after using them to avoid such problems.

The DBMS doesn't always offer precisely 100 connections. This number is configurable at the database level. When working with a database, it's best to find out (usually by asking the database administrator) the maximum number of connections the app can open.



NOTE To simplify our demonstrationin this section, we'll use a persistence layer that limits the number of connections to 10.

Let's start project da-ch8-ex1 and analyze the app's behavior. This project defines a simple application that stores details about products in a database. The app exposes an endpoint at the /products path. Calling the endpoint, the app returns details based on data it stores in its database. When you call the endpoint the first time after starting the app, the app responds almost instantaneously. But sending the second request to the same endpoint, the app responds after 30 seconds with an error message, as shown in figure 8.2.

What the app actually does is not essential for our demonstration, so that I won't go into details about its functionality. Imagine a friend working on a separate project calls you for your help and shows such a problem. They won't give you many details about how their app works (in a real-world app, it could be a really complicated business case). So how can you still help them? We start by analyzing the behavior they show us.

When calling the endpoint the first time,
the app responds immediately.

```
$ curl http://localhost:8080/products
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload Upload Total   Spent   Left Speed
100     13     0    13     0      0  119      0 --:--:-- --:--:-- --:--:-- 120["Chocolate"]
```

But when calling the endpoint the second time,
the app throws an error after 30 seconds.

```
$ curl http://localhost:8080/products
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload Upload Total   Spent   Left Speed
100   109     0   109     0      0      3      0 --:--:-- 0:00:30 --:--:-- 28{"timestamp": "07:55:09.272+00:00", "status": 500, "error": "Internal Server Error", "path": "/products"}
```

Figure 8.2 When calling the /products endpoint the first time after starting the app, the app responds instantaneously with a list containing the word “Chocolate”. But when trying to call the endpoint the second time, the app looks to be stuck for about 30 seconds and then shows an error message.

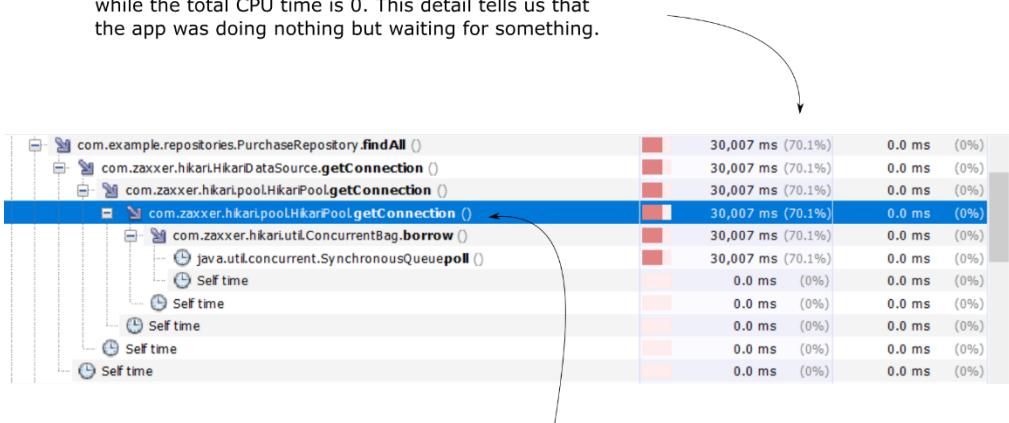
We want to find out what causes such behavior. If you have access to the logs, you already have a reason to suspect the problem is related to the JDBC connections. In the following snippet, you observe that the exception message tells us that the app couldn’t obtain a connection. The exception message tells you the app couldn’t get a JDBC connection, most likely because the DBMS doesn’t allow other connections to be opened. But, let’s assume we cannot always rely on logs. In the end, we can’t know if other frameworks or libraries that your app uses might generate the same decent exception messages.

```
java.sql.SQLTransientConnectionException: HikariPool-1 - Connection is not available,
request timed out after 30014ms.      #A
at com.zaxxer.hikari.pool.HikariPool.createTimeoutException(HikariPool.java:696)
at com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:197)
at com.zaxxer.hikari.pool.HikariPool.getConnection(HikariPool.java:162)
at com.zaxxer.hikari.HikariDataSource.getConnection(HikariDataSource.java:128)
at com.example.repositories.PurchaseRepository.findAll(PurchaseRepository.java:31)
at com.example.repositories.PurchaseRepository$$FastClassBySpringCGLIB$$d661c9a0.invoke
(<generated>)
```

#A The exception message.

As I recommended in chapter 7, every profiling investigation should start with sampling. Sampling gives you an overview of the execution and the details you need to continue the research. Say you use VisualVM; the sampling result would look as presented in figure 8.3.

We observe that the total spent time is about 30 seconds, while the total CPU time is 0. This detail tells us that the app was doing nothing but waiting for something.



By the method name, we can deduce that the app was waiting to establish a connection with the DBMS.

Figure 8.3 After sampling the execution, we have more reasons to suspect something is wrong with establishing a connection to the DBMS. We observe in the execution stack that the app waited for 30 seconds to get a connection.

After sampling and observing the exception stack trace in the logs, we know our app has problems connecting to the DBMS. But what causes this problem? This kind of problem could be caused usually by one of these two reasons:

1. The communication between the app and the DBMS fails because of some infrastructure or networking problems.
2. The DBMS doesn't want to provide a connection to our application
 - a) Because of an authentication problem.
 - b) Because the app already consumed all the connections the DBMS can offer.

Since in our case the issue always happens the second time a request is sent (so the issue has a defined pattern to reproduce it), we can exclude the communication problem (point 1). It must be that the DBMS doesn't provide a connection. But it can't really be an authentication problem since the first call worked well. It's unlikely something could have changed meanwhile with the credentials. So the most plausible cause is that our app sometimes doesn't close the connections. Now we just have to find where this happens. Remember that the method that encountered the problem isn't necessarily the one causing it. It might be that this method is just the "unlucky" one that tried to get a connection after someone else "ate" all the others.

But with VisualVM, you can't explicitly investigate JDBC connections, so we can't use it to precisely identify which connection stays open. Instead, we'll continue our discussion using JProfiler. Attaching JProfiler to a running Java process is very similar to using VisualVM. Let's follow the approach step by step.

First, you need to select **Start Center** in the left upper corner of JProfiler's main window, as presented in figure 8.4.

After opening JProfiler, select Start Center to attach the profiler to a process.

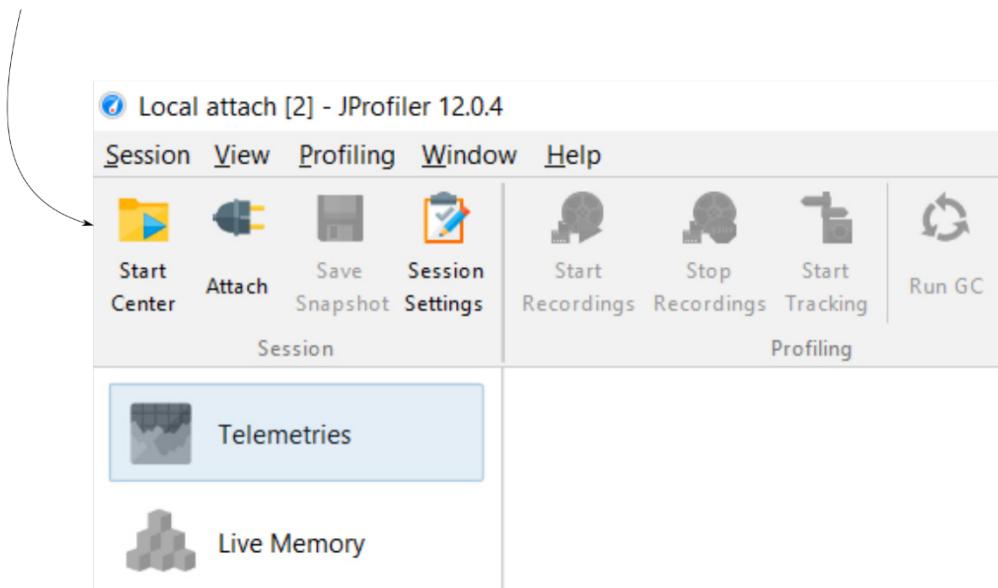


Figure 8.4 Start a sampling or profiling session with JProfiler by selecting the Start Center menu on the left upper side of the JProfiler window.

A popup window appears (figure 8.5), and you can select **Quick Attach** on the left to get a list of all Java processes running locally. You choose the process you want to profile and then select **Start**. Same as with VisualVM, you identify the process using the name of the main class or the process ID (PID).

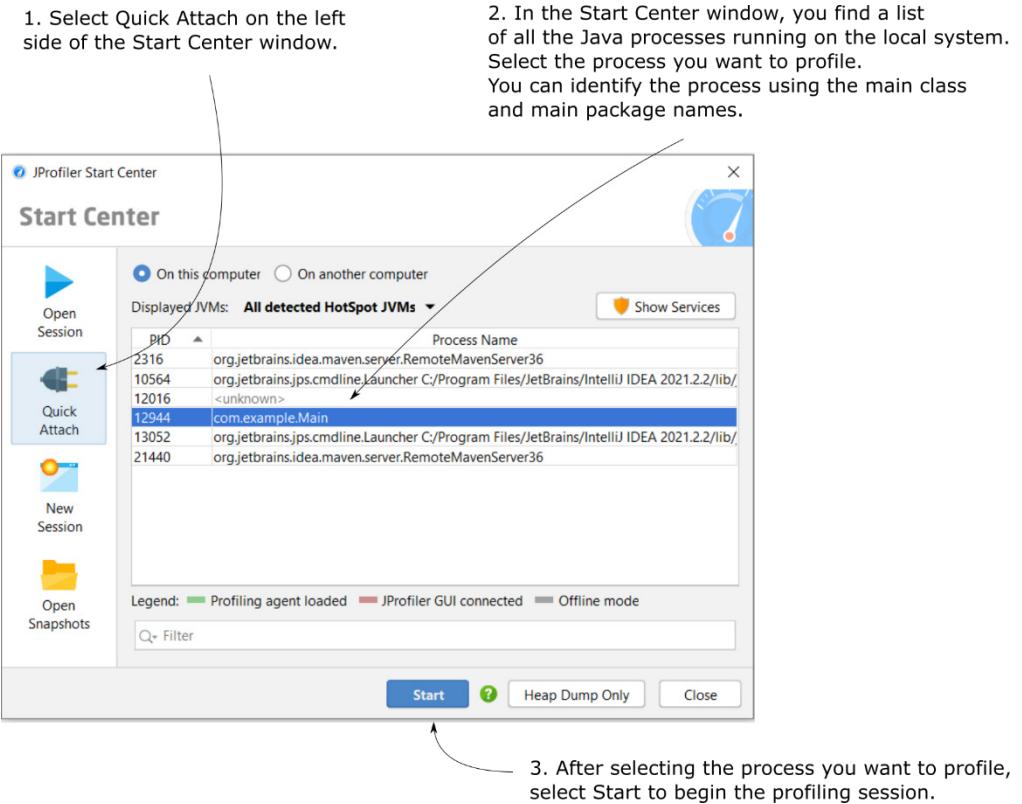


Figure 8.5 In the pop-up window, select Quick Attach, and then in the list, select the process you want to profile. After choosing the process you want to profile, select the Start button to begin the profiling session.

JProfiler will ask you further if you will use sampling or instrumentation (instrumentation is equivalent to what we called profiling with VisualVM), as shown in figure 8.6. We select instrumentation since we'll use the profiler to get details about the JDBC connections, so we want to analyze the execution more in-depth.

We continue with selecting Instrumentation.
 Instrumentation is equivalent to what we called "profiling" in VisualVM.
 Just as we discussed with VisualVM, when investigating a problem
 you first use Sampling to identify the area that's potentially containing
 the root cause, and then you use profiling (instrumentation) to investigate
 deeper. The approach we discuss is part of Instrumentation.

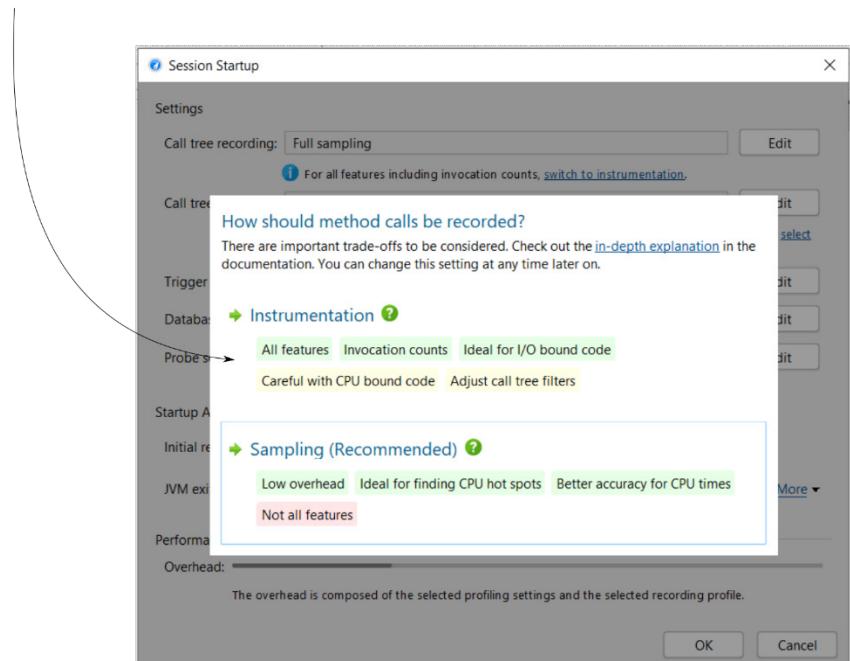


Figure 8.6 To analyze the execution in-depth, we need to select Instrumentation. Instrumentation is equivalent to what we call profiling with VisualVM.

Under the **Databases** item in the left menu, select JDBC. Then, start the JDBC profiling as presented in figure 8.7.

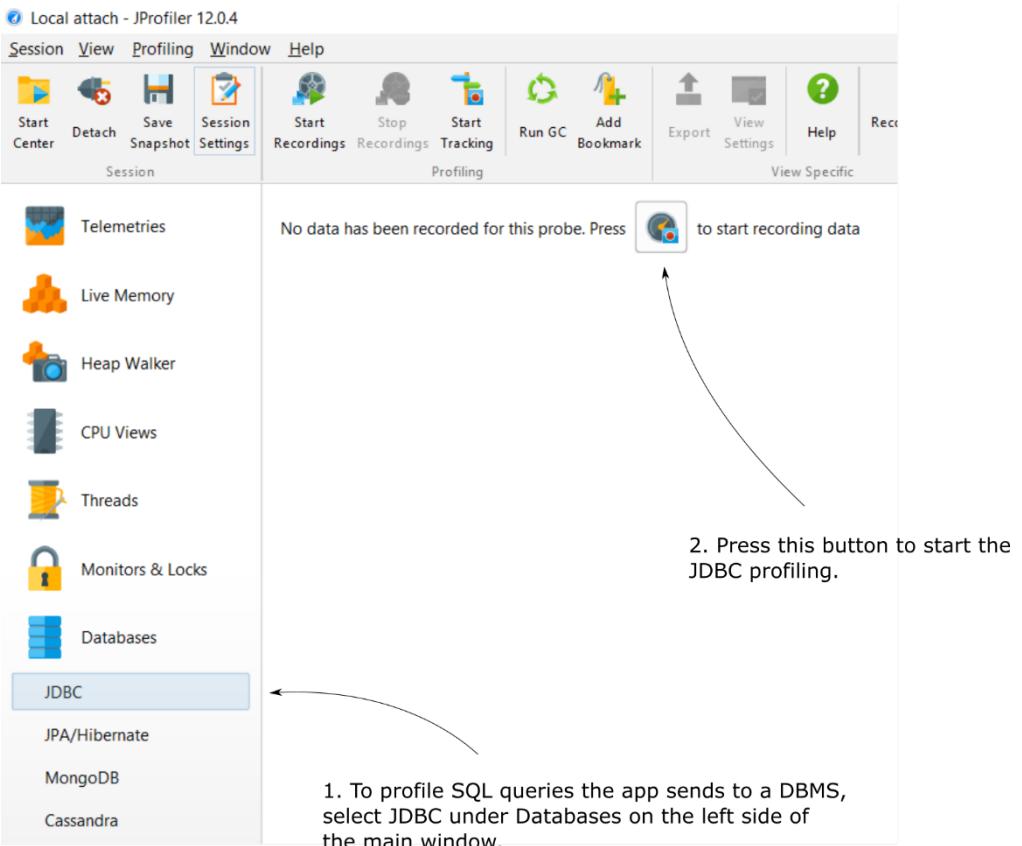


Figure 8.7 You can start JDBC profiling with JProfiler by first selecting JDBC in the left menu under profiling and then beginning the profiling process as presented in the visual.

Once the profiling process starts, we'll mostly be interested in two tabs from those that JProfiler displays: **Connections** and **Connection Leaks** (figure 8.8). These tabs will show us details about the connections to the DBMS the app opens, and we'll use them to identify the problem's root cause.

In this section, we are particularly interested in the Connections and the Connection Leaks tabs. Using these tabs, we'll identify problems with connections that the app doesn't close.

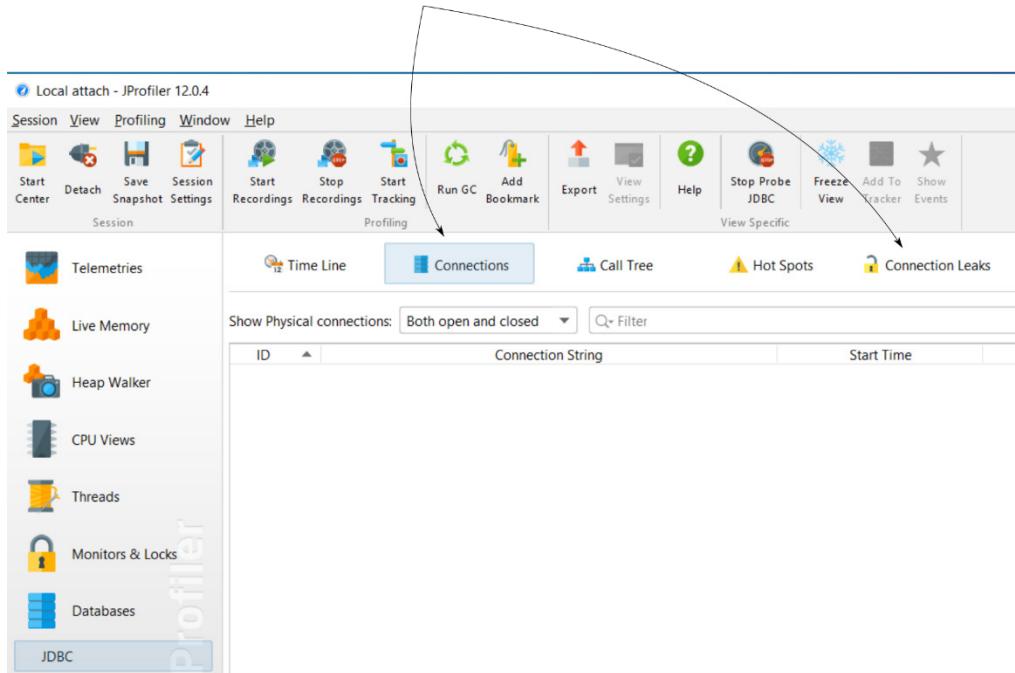


Figure 8.8 The Connections and Connection Leaks tabs will show details about the connections the app creates and the potential problematic connections. We'll use the details provided by these tabs to understand where the problem in our app comes from.

Now it's time to reproduce the problem and profile the execution. Send a request to the /products endpoint, and let's see what happens. The **Connections** tab will show that many connections are created, as presented in figure 8.9. Since we don't know what the app does, many connections don't necessarily mean problems. But we expect that the app closed these connections so the app can get other connections when needed. So what we need to figure out is if these connections are indeed closed correctly by the app.

When sending a request to the /products endpoint, the app opens a large number of connections. We use the Connections tab in the profiler to observe the connections when app opens.

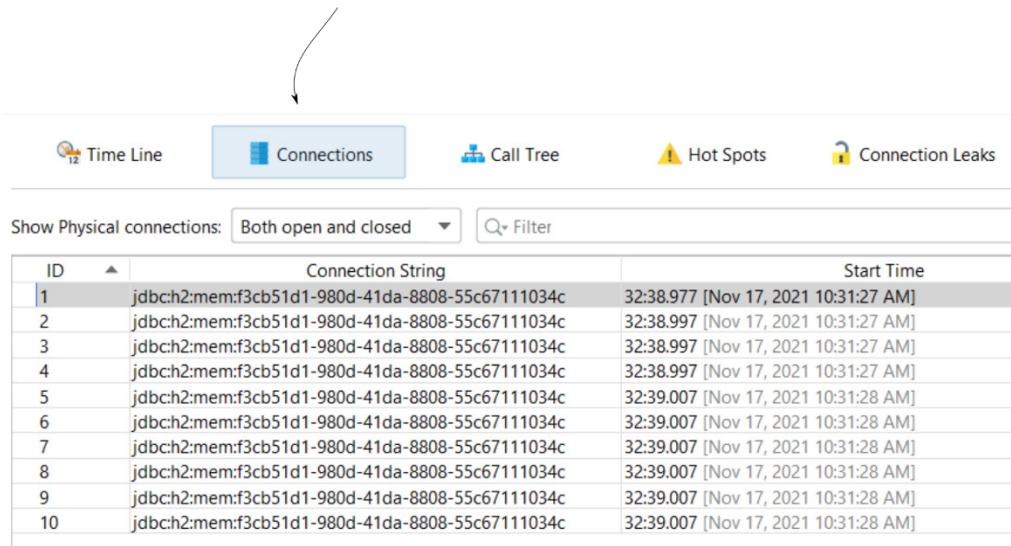


Figure 8.9 Sending a request to the /products endpoint, we observe the app creates many connections. We don't know exactly what the app does, but still, this can be an alarming point.

The **Connection Leaks** tab confirms our suspicion (figure 8.10), not only does the app open a lot of connections, the connections remain unclosed a long time after the endpoint responded. This is a clear sign of a connection leak. If we didn't explicitly start the CPU profiling (I'll show you in a moment how to do that), you'd only see the name of the thread that created the connection. Sometimes, the thread's name is enough, so in such a case, you don't need to start CPU profiling at all, but in other situations such as this one, it doesn't tell us enough to identify the code that created the connection.

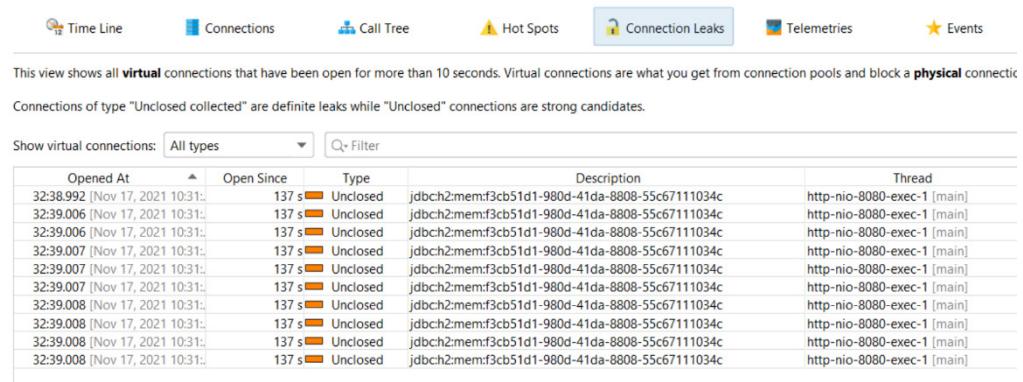


Figure 8.10 The Connection Leaks tab shows us the status of each connection. We are explicitly interested in the connections that close late or never. In our case, the connections the app opened are still alive even a long time after the endpoint sent back a response to the client, which strongly indicates a problem.

But that's not enough, isn't it? We were anyway suspecting that something is wrong with the app obtaining a connection to the DBMS. Does the profiler bring us other advantages? We'll use the profiler's CPU profiling capability to identify exactly the part of the codebase that creates the connections and forgets to close them.

We still need a way to identify the code that creates the leaking connections. Fortunately, JProfiler can help us with this as well. But we'll need to redo the exercise after also enabling the CPU profiling. With the CPU profiling active, JProfiler will show for each leaking connection the stack trace to the method that created the connection.

Figure 8.11 shows how to enable the CPU profiling and how you find the stack trace for each leaking connection.

The screenshot shows the JProfiler interface with the 'Session' tab selected. In the main pane, there's a message: 'Press [] to record CPU data'. Below it are four icons: Telemetries, Live Memory, Heap Walker, and CPU Views. The 'Call Tree' icon is highlighted with a blue border. A callout arrow points from the text 'The stack trace appears below the connections table when selecting the row that represents a certain connection.' to the 'Call Tree' icon.

The bottom half of the screenshot shows the 'JDBC' tab of the JProfiler interface. It displays a table of virtual connections:

Opened At	Open Since	Type	Description	Thread	Class Name
049.542 [Jan 8, 2022 4:3...	49,543 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.574 [Jan 8, 2022 4:3...	49,511 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.577 [Jan 8, 2022 4:3...	49,508 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.578 [Jan 8, 2022 4:3...	49,507 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.580 [Jan 8, 2022 4:3...	49,505 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.581 [Jan 8, 2022 4:3...	49,504 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.583 [Jan 8, 2022 4:3...	49,503 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.584 [Jan 8, 2022 4:3...	49,501 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...
049.585 [Jan 8, 2022 4:3...	49,500 ms	Unclosed	jdbch2:mem:755fc36c-f8f4-4c53-8138-bed09591a58d	http-nio-8080-exec-2 [main]	com.zaxxer.hikari.pool...

A callout arrow points from the text 'The stack trace appears below the connections table when selecting the row that represents a certain connection.' to the table. Another callout arrow points from the text 'If you want to see the exact stack trace that indicates the place that created the connection, you need first to start the CPU profiling. To start the CPU profiling, you select Call Tree from the left side of the window and then start recording CPU data.' to the 'Call Tree' icon. A third callout arrow points from the text 'Connections of type "Unclosed collected" are definite leaks while "Unclosed" connections are strong candidates.' to the 'Unclosed' column header in the table.

The stack trace for the selected row (id=90b7) is shown in a scrollable text area:

```

com.zaxxer.hikari.HikariDataSource.getConnection()
javax.sql.DataSource.getConnection()
com.example.repositories.ProductRepository.findProduct(int)
com.example.repositories.ProductRepository$$FastClassBySpringCGLIB$$69752884.invoke(int, java.lang.Object, java.lang.Object[])
org.springframework.cglib.proxy.MethodInterceptor.intercept(java.lang.Object, java.lang.reflect.Method, java.lang.Object[], org.springframework.cglib.proxy.MethodProxy)
com.example.repositories.ProductRepository$$EnhancerBySpringCGLIB$$de9c90b7.findProduct(int)
com.example.services.PurchaseService.getProductNamesForPurchases()
com.example.controllers.PurchaseController.findPurchasedProductNames()
HTTP:/products
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run()

```

Figure 8.11 With the CPU profiling enabled, JProfiler also shows the stack trace, which helps you identify which part of the app's code creates the leaking connections.

Let's go straight to that code in our example da-ch8-ex1. Listing 8.1 shows the method indicated by JProfiler. We observe that the method indeed creates a connection that seems not to be closed anywhere. We found the root cause!

Listing 8.1 Identifying the problem root cause

```

public Product findProduct(int id) throws SQLException {
    String sql = "SELECT * FROM product WHERE id = ?";
}

```

```

Connection con = dataSource.getConnection();      #A
try (PreparedStatement statement = con.prepareStatement(sql)) {
    statement.setInt(1, id);
    ResultSet result = statement.executeQuery();

    if (result.next()) {
        Product p = new Product();
        p.setId(result.getInt("id"));
        p.setName(result.getString("name"));
        return p;
    }
}
return null;
}

```

#A This line creates a connection that is never closed.

Project da-ch8-ex2 corrects the code as shown by listing 8.2. Adding the connection to the try-with-resources, the app will close the connection at the end of the try block when the connection isn't needed any longer.

Listing 8.2 Solving the problem by closing the connection

```

public Product findProduct(int id) throws SQLException {
    String sql = "SELECT * FROM product WHERE id = ?";

    try (Connection con = dataSource.getConnection();      #A
         PreparedStatement statement = con.prepareStatement(sql)) {
        statement.setInt(1, id);
        ResultSet result = statement.executeQuery();

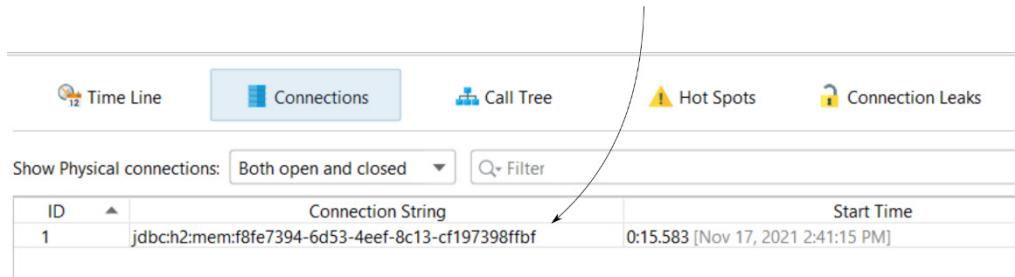
        if (result.next()) {
            Product p = new Product();
            p.setId(result.getInt("id"));
            p.setName(result.getString("name"));
            return p;
        }
    }
    return null;
}

```

#A The connection is declared in the try-with-resources block, closing the connection at the end of the try block.

We can profile the app again after applying the correction. Now, the **Connection** tab in JProfiler shows us that only one connection is created, and the **Connection Leaks** tab is empty, proving that the problem was indeed solved (figure 8.12). Testing the app, you also observe that you can send multiple requests to the /products endpoint.

Since the app now correctly closes the connections, the Connection tab only displays one row.



The Connection Leaks tab is now empty.

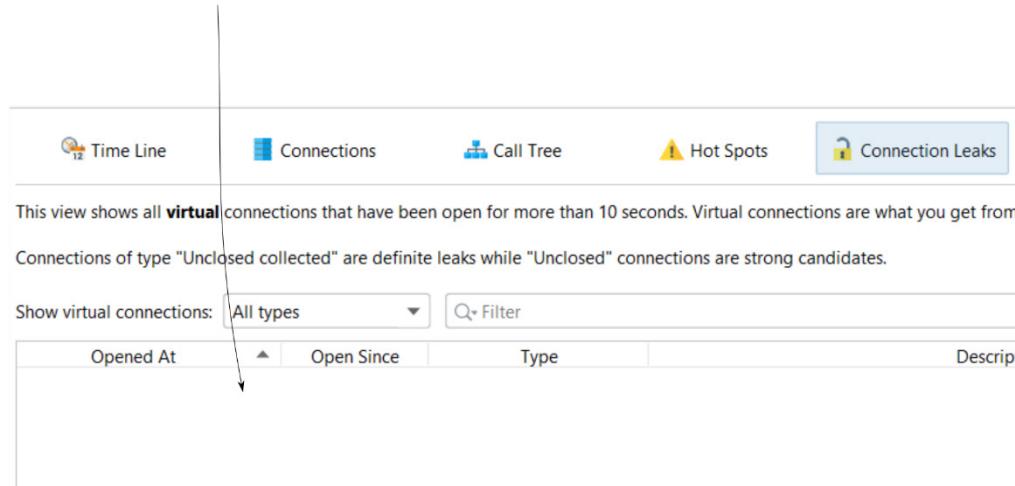


Figure 8.12 After fixing the error, we use JProfiler to confirm that no more connections are leaking. We observe that the app opens only one connection at a time and correctly closes the connection when not needing it any longer. The Connection Leaks tab doesn't show any other faulty connection.

Do you wonder if there is a best practice to avoid such problems in real-world scenarios? I recommend that developers take 10 minutes or so after each implementation or bug fix to test the capability they worked on with a profiler. This practice usually helps identify latency issues caused by wrong queries or faulty connection management from the early phases of development.

8.2 Understanding the app's code design using call graphs

In this section, we discuss one of my favorite techniques I use to understand an app's class design – visualizing the execution as a call graph. This technique especially helps me when I deal with messy code, eventually of a new app that I've never seen before.

Up to this point, we only used stack traces to understand the execution. The execution stack traces are valuable tools, and we've already seen how many things we can do with them in the past chapters. They are also helpful because of their straightforward way of representing as text, which gives them the possibility to be printed in logs (usually as exception stack traces). But from a visual point of view, they're not great at quickly identifying the relationship between objects and method calls. Call graphs are a different way to represent the data a profiler collects, which focuses more on the relationships between the objects and method calls.

To demonstrate how to obtain a call graph, we'll use example da-ch8-ex2 provided with the book. What we'll do, is to prove that we can use call graphs to quickly understand which objects and methods act behind an execution without even needing to analyze the code upfront. Of course, the idea isn't to avoid the code completely – you'll still end up digging code in the end, but using call graphs first, you'll have a better picture of what happens up front.

We continue using JProfiler for our demonstration. Since call graphs are a way to represent CPU profiler data, we need first to start CPU profiling. Figure 8.13 shows how to start CPU profiling which results in the first place into a stack trace (also named "call tree" in JProfiler). We'll investigate what happens when calling the /products endpoint the app exposes.

The screenshot shows the JProfiler 12.0.4 interface with the 'Session' tab selected. The left sidebar has 'Call Tree' highlighted. The main area displays a stack trace titled 'Press [] to record CPU data'. The stack trace shows method invocations with execution times and invocation counts. A callout points from the text '2. Same as in VisualVM, JProfiler provides a stack representation of the execution.' to the stack trace.

1. To profile the CPU data to obtain the execution stack, start CPU profiling in the Call Tree section.

2. Same as in VisualVM, JProfiler provides a stack representation of the execution.

3. In the stack representation, you find essential details such as the execution time and the number of invocations of methods.

The screenshot shows the JProfiler 12.0.4 interface with the 'Profiling' tab selected. The left sidebar has 'Call Tree' highlighted. The main area displays a detailed stack trace with various method invocations and their execution times. A callout points from the text '3. In the stack representation, you find essential details such as the execution time and the number of invocations of methods.' to the stack trace.

Figure 8.13 Select Call Tree from the left menu to start profiling the CPU (recording CPU data). Send a request to the /products endpoint, and the profiler will initially show the recorded data as a stack trace (also named call tree in JProfiler). In the stack trace representation, you find details about the number of invocations and the execution time.

Right-click on a line in the stack trace and select **Show Call Graph** to visualize the data collected about the execution as a call graph (figure 8.14).

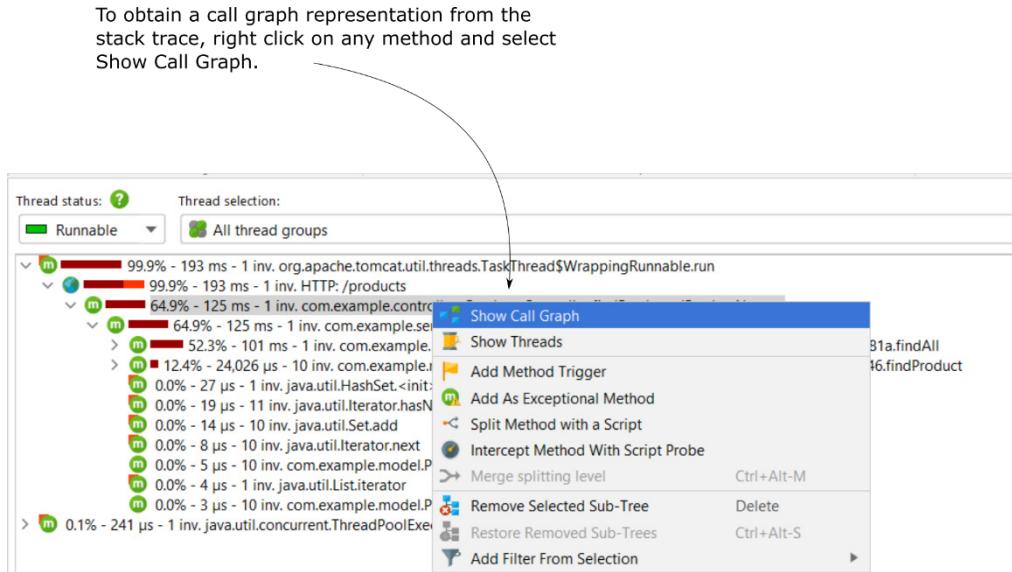


Figure 8.14 To get a call graph representation from an execution stack trace, right-click on a line in the stack trace and select Show Call Graph.

JProfiler will generate a call graph representation focusing on the method defined by the line you selected previously when generating the call graph. You'll initially only know where this method is called from and who this method calls. Further, as shown in figure 8.15, you can navigate further and observe the entire call chain. The call graph also provides details about the execution time and the number of invocations, but its focus is mainly on the relationship between the objects and method calls.

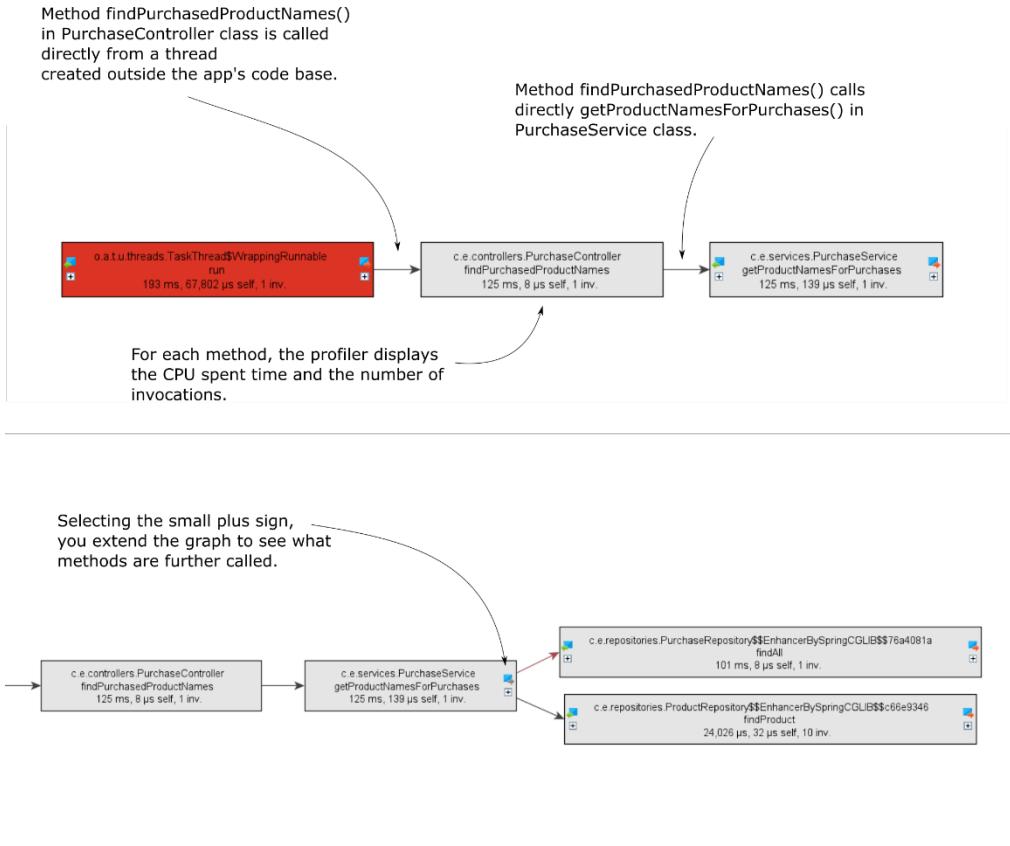


Figure 8.15 A call graph shows the execution focusing mainly on the relationship between objects and method calls. You can linearly navigate the method execution chain to determine where each method is called from and who that method calls further. The call graph shows objects and methods that are part of the app's codebase and those of libraries and frameworks the app uses.

8.3 Using flame graphs to easier spot performance problems

Another way to visualize the profiled execution is using a representation named flame graph. If call graphs (discussed in section 8.2) focus on the relationship between objects and method calls, flame graphs are most helpful in identifying potential latencies. Still, the flame graphs are just a different way to see the same details a method execution stack provides, but, as mentioned in the chapter introduction, other representations of the same data may help identify specific information.

We continue using example da-ch8-ex2 for this demonstration. We'll use JProfiler to change the execution stack representation (also called call tree in JProfiler) in a flame graph and discuss the new representation's advantages.

After generating a call tree as discussed in sections 8.1 and 8.2, you can change it to a flame graph using the Analyze item in the top menu bar, as shown in figure 8.16.

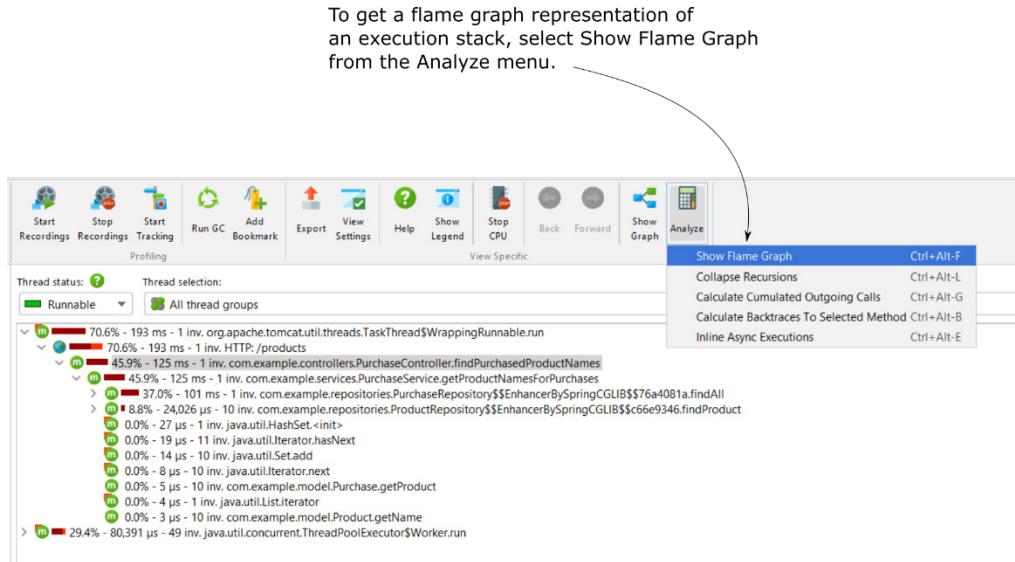
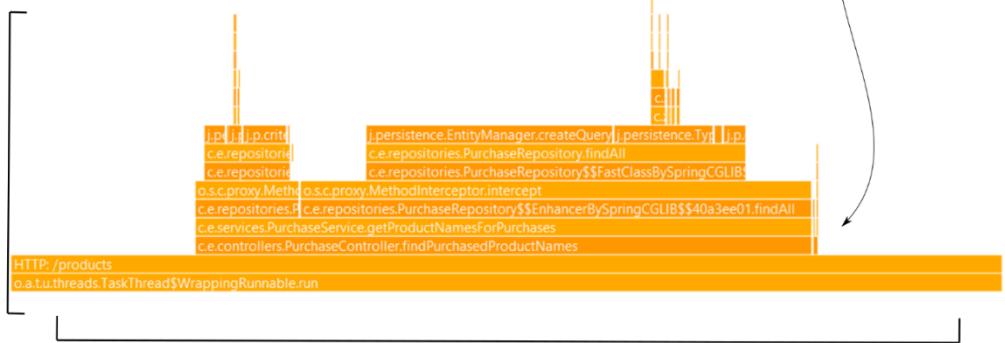


Figure 8.16 To change an execution stack (call tree) into a flame graph, select the Analyze item in the menu and then Show Flame Graph.

A flame graph is a way to represent the execution tree as a stack. The fancy name is just because the graph usually looks like a flame. The first level of this stack is the first method the thread executed. Then, every level above is shared by the methods called by the layer below. Figure 8.17 shows you the flame graph created for the execution tree in figure 8.16.

Each method is a level in the execution stack. The level below is who called that method, the levels above is who that method calls. For example, we observe here that method `findPurchasedProductNames()` in `PurchaseController` class calls `getProductNamesForPurchases()` in `PurchaseService` class. Further, the method in `PurchaseService` class calls `findAll()` of `PurchaseRepository`.

On the vertical, we observe the execution stack.



Horizontally, we have the execution time.

Figure 8.17 The flame graph is a stack representation of the execution tree. Each level shows the methods called by the level below. The first (bottom) level of the stack is the start of the thread. This way, vertically, we see the execution stack, while horizontally, the flame graph indicates the time spent by each level relative to the level below.

A method can call multiple other methods. So in the flame graph, they will appear on the same level. In such a case, the length of each is the time spent relative to the method calling it (the level below). In figure 8.17, you can see that method `findById()` in the `ProductRepository` class, and method `findAll()` in the `PurchaseRepository` class, were both called from the `getProductNamesForPurchases()` in the `ProductService` class.

In figure 8.18 we observe that `getProductNamesForPurchases()` in the `ProductService` class is the bottom level for both method `findById()` in the `ProductRepository` class, and method `findAll()` in the `PurchaseRepository` class. Moreover, the two methods `findById()` and `findAll()` share the same layer. But observe that they don't have the same length. The length is relative to the caller's execution, so in this case, it means that the execution time of `findById()` is less than the execution time of `findAll()`.

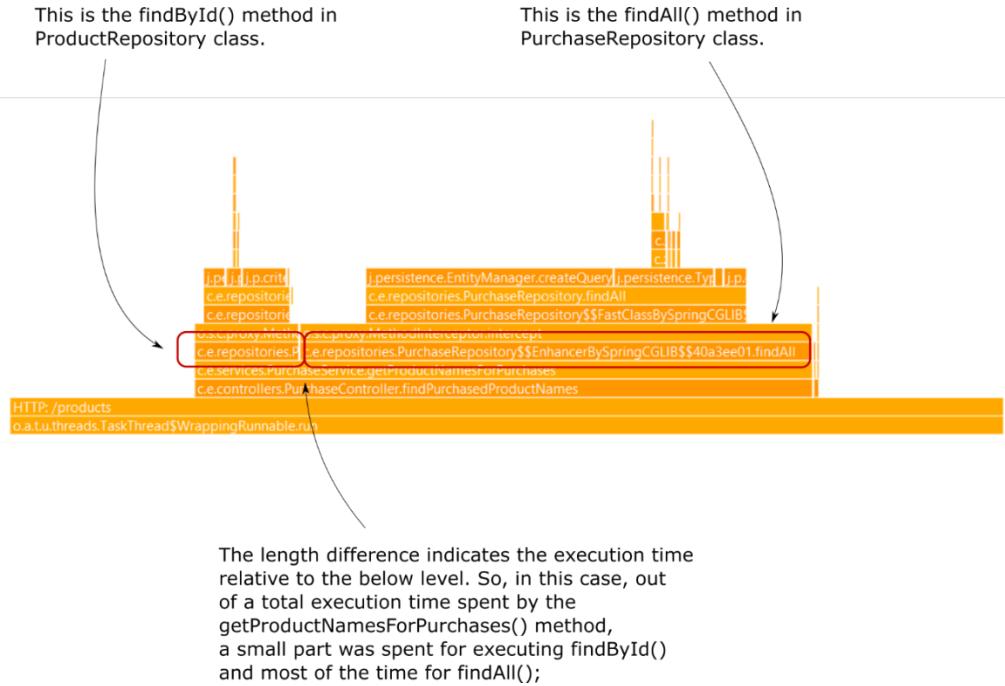


Figure 8.18 When multiple methods share the same level, it means they are all called by the method below them. The sum of lengths of the representation equals the length of the method under them. Each method's length is a relative representation of the execution time out of the total. In this case, it means that findAll() took much longer to execute than findById();

You might already have observed that one could get easily lost in this graph. And this is only a simple example for study purposes; in a real-world app, the flame might be much more complex. To mitigate this complexity, one of the things you can do in JProfiler is to color the layers based on the methods, class, or package name. Figure 8.19 shows how to use colorization to mark specific layers in the flame graphs. You use the **Colorize** item in the top menu to add colorization rules. You can add multiple colorization rules specifying which layers to be colored and the color you prefer.

1. Select Colorize

2. Add a new colorization rule.

3. Define the colorization rule and select OK.

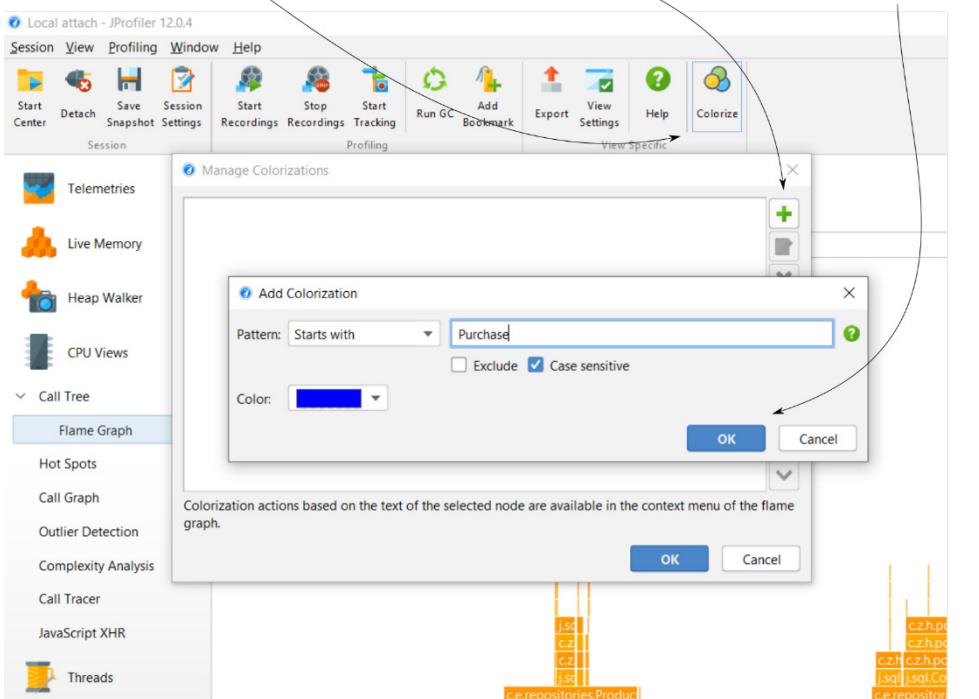


Figure 8.19 To colorize your flame graph and make it easier to read, you add colorization rules. You can add colorization rules using the Colorize item in the top menu. Any rule defines which layers in the flame graph should be colored and which color should be used.

In figure 8.20, you can observe how I highlighted the levels for the methods containing the word "Purchase" in their names, colorizing them in blue.

You can independently colorize the levels based on the class and package name to follow the graph easier. For example, in this case, I colorized with blue all the methods containing the word "Purchase" in their names.

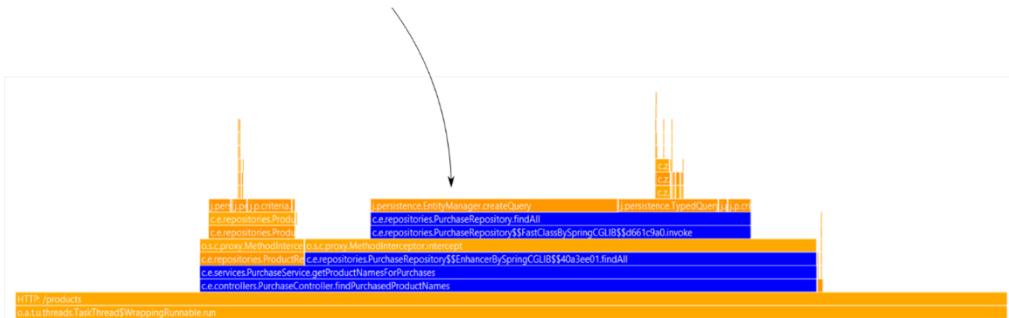


Figure 8.20 Colorizing levels helps highlight specific parts of the flame that you want to focus on. You can use multiple colors at the same time, which can help you compare execution times easier.

8.4 Analyzing queries on NoSQL databases

Applications often use relational databases, but in many cases, certain implementations need different persistence technologies. We name these NoSQL technologies and apps we implement can choose from a large variety of such implementations. Some of the most known examples are MongoDB, Cassandra, Redis, and Neo4J. Some profilers, such as JProfiler, can intercept queries the app sends to a specific NoSQL server to work with a database.

JProfiler can intercept events sent to MongoDB and Cassandra, and these details might help you save time when you investigate the behavior of an app using such a persistence implementation. For this reason, in this section, we'll use a small app to demonstrate the use of JProfiler to observe an app's activity with a MongoDB database.

Project da-ch8-ex3 works with MongoDB. The app implements a couple of endpoints, one that stores details of products in the database and another that returns a list of all previously added products. To make it simple, a product is only represented by a name and a unique ID.

If you want to follow me in this section, you'll first need to install locally a MongoDB server to which project da-ch8-ex3 will connect. You can download and install MongoDB Community Server from the official page:

<https://www.mongodb.com/try/download/community>

Once you have installed the server, you can start project da-ch8-ex3. We will also attach JProfiler to the process. To begin monitoring MongoDB events, you select the MongoDB section under **Databases** in the left menu and start recording. To observe how JProfiler presents the events, we'll call the two endpoints the app exposes. You can call the two endpoints using the cURL commands shown in the following snippet, or a tool such as Postman.

```
curl -XPOST http://localhost:8080/product/Beer      #A
curl http://localhost:8080/product      #B

#A Adds a product named "Beer" to the database
#B Gets all the products in the database
```

Figure 8.21 shows the two events intercepted by JProfiler. The tool shows the stack traces (call trees) associated with each event. We get details about the number of invocations and the execution time.

When you trigger the endpoint that generates an update in the database, you can find the event intercepted in JProfiler. You get the stack trace (call tree) and valuable details related to the operations such as the number of invocations and the execution time.

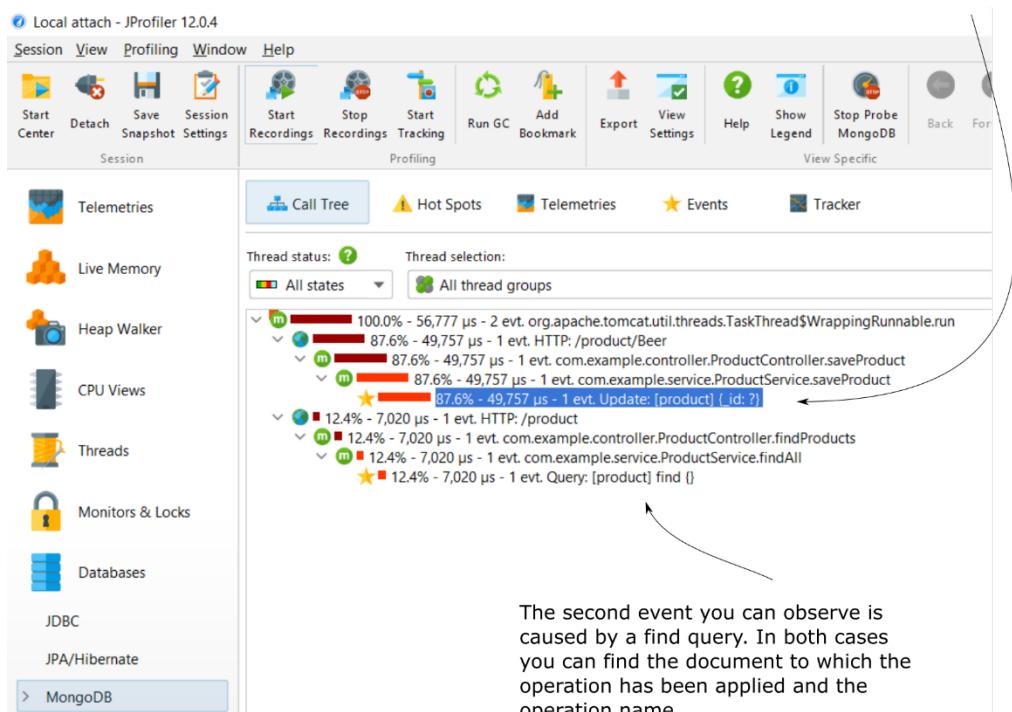


Figure 8.21 JProfiler can intercept the operations an app applies to a NoSQL database. In this example, JProfiler intercepts two events: an update and a read on a document named "product". This way, you can monitor the interaction between your app and a NoSQL database, considering the number of invocations for specific operations and the execution time. The profiler also gives you the complete stack trace for a particular operation to quickly find the code that caused a specific event.

8.5 Summary

- Free tools such as VisualVM offer plenty of widgets that help with any investigation. But licensed tools such as JProfiler can make the investigations even more comfortable and effective through different ways of representing the investigation data.
- Sometimes, apps encounter issues when connecting to a DBMS. Using JProfiler, you can more easily investigate issues with a JDBC connection to a relational database server. You can evaluate if connections remain open and identify the part of code that "forgets" to close them.
- Call graphs are a different way to visualize the execution stack. Call graphs focus mainly on the relationship between objects and method calls. For this reason, call graphs are an excellent tool you can use to understand easier the class design behind the app's execution.
- Flame graphs offer a different perspective for visualizing the profiled data. You can use flame graphs to spot easier places causing latencies in execution and long stack traces. You can color differently specific layers in a flame graph to easier understand the visual representing the execution.
- Some licensed tools also offer extended capabilities, such as investigating the communication between apps, or between your app and a NoSQL database server.

9

Investigating locks in multithreaded architectures

This chapter covers

- Monitoring an application's threads
- Identifying thread locks and what causes them
- Analyzing threads that are waiting

In this chapter, we discuss approaches to investigate the execution of apps that leverage multithreaded architectures. Generally, developers find implementing multithreaded architectures to be one of the most challenging things in app development. Not only that implementing functional capabilities where multiple threads collaborate using common resources is challenging, but making the app performant as well brings another dimension of difficulty. The techniques we'll discuss in this chapter will give you visibility into the execution of such apps, allowing you to more easily identify problems and optimize the app execution.

Before starting the chapter, you might need a refresher on the basics of threads in Java apps. To properly understand the content of this chapter, I expect you already know the basics of threading mechanisms in Java including thread states and synchronization. If you already know these, you can just proceed with this chapter's discussion. Alternatively, you should read appendix D. Appendix D won't give you all the possible knowledge on threads and concurrency in Java (that would need its own bookshelf) but will give you just enough details to understand this chapter's discussion.

9.1 Monitoring threads for locks

In this section, we discuss thread locks and how to analyze them to find eventual issues or opportunities to optimize an app's execution. Thread locks are caused by different thread

synchronization approaches usually implemented to control the flow of events in a multithreaded architecture. Thread locks are needed, but wrongly implemented they might affect performance or even make an app crash. Examples of such flows where locks are needed to control the executions are

- A thread wants to prevent other threads from accessing a resource while it's changing that resource.
- A thread needs to wait for another one to finish or reach a certain point in its execution before being able to continue its work.

Locks are necessary. They help an app control threads. But implementing thread synchronization leaves a lot of room for mistakes. Wrongly implemented locks may cause app freezes or performance issues. We need to use profilers to make sure our implementations are optimal. What we are looking for is making an app more efficient by minimizing the lock time.

In this section, we'll use a small application (project da-ch9-ex1) that implements a simple multithreaded architecture. We'll use a profiler to analyze the locks during the app's execution. What we want to achieve is to find out if the threads are locked and which is their behavior:

- Which thread locks another
- How many times a thread is locked
- What is the time a thread pauses instead of executing

These details allow us to understand if the app execution is optimal and if there're ways in which we can improve our app's execution. The app we use for our example implements two threads named the producer and the consumer. These two threads run concurrently. The producer generates random values and adds them to a list instance. Concurrently, the consumer removes values from the same collection used by the producer (figure 9.1).

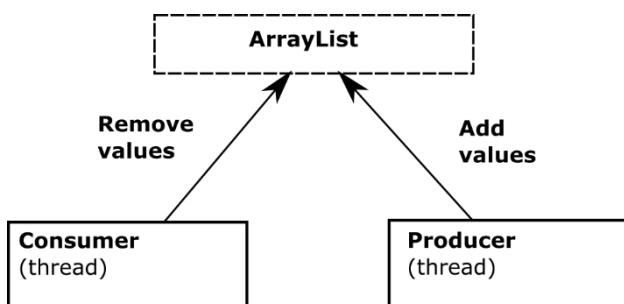


Figure 9.1 The app starts two threads that we'll refer to as “the producer” and “the consumer”. Both threads use a common resource: they change a list instance of type `ArrayList`. The producer generates random values and adds them to the list, while the consumer concurrently removes the values added by the producer.

Let's follow the app implementation in listings 9.1, 9.2, and 9.3 to have an idea of what to expect when we investigate the execution. In listing 9.1 you find the `Main` class which starts

the two thread instances. I made the app wait for 10 seconds before starting the threads to allow us some time to start the profiler and observe the entire threads' timelines. The app names the threads “_Producer” and “_Consumer” to allow us to easily identify them when working with the profiler.

Listing 9.1 App's main method starts two threads

```
public class Main {

    private static Logger log = Logger.getLogger(Main.class.getName());

    public static List<Integer> list = new ArrayList<>();

    public static void main(String[] args) {
        try {
            Thread.sleep(10000);      #A

            new Producer("_Producer").start();      #B
            new Consumer("_Consumer").start();      #C
        } catch (InterruptedException e) {
            log.severe(e.getMessage());
        }
    }
}
```

#A Waiting 10 seconds, in the beginning, to let the programmer start the profiling.

#B Starting a producer thread

#C Starting a consumer thread

In listing 9.2, you find the consumer thread's implementation. The thread iterates over a block of code one million times (this number of times should be enough for the app to run a few seconds and allow us to use the profiler to take some statistics). During every iteration, the thread uses a static list instance declared in the `Main` class. The consumer thread checks if the list has values and removes the first value in the list. The whole block of code implementing the logic is synchronized, using the list instance itself as a monitor. The monitor won't allow multiple threads to enter at the same time in the synchronized blocks it protects.

Listing 9.2 The consumer thread's definition

```
public class Consumer extends Thread {

    private Logger log = Logger.getLogger(Consumer.class.getName());

    public Consumer(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 1_000_000; i++) {      #A

            synchronized (Main.list) {      #B
                if (Main.list.size() > 0) {    #C
                    int x = Main.list.get(0);    #D
                }
            }
        }
    }
}
```

```

        Main.list.remove(0);      #D
        log.info("Consumer " +      #E
                  Thread.currentThread().getName() +
                  " removed value " + x);
    }
}

}
}

```

#A Iterating 1 million times over the consumer's synchronized block of code
#B Synchronizing the block of code using the static list defined in the Main class as a monitor.
#C Trying to consume a value only if the list is not empty.
#D Consuming the first value in the list and removing that value.
#E Logging the removed value.

Listing 9.3 presents the producer's thread implementation which is pretty similar to the consumer's. The producer also iterates one million times over a block of code. For each iteration, the producer generates a random value and adds it to a list statically declared in the `Main` class. This list is the same one from which the consumer removes the values. The producer adds new values only if the list size is smaller than 100.

Listing 9.3 The producer thread's definition

```

public class Producer extends Thread {

    private Logger log = Logger.getLogger(Producer.class.getName());

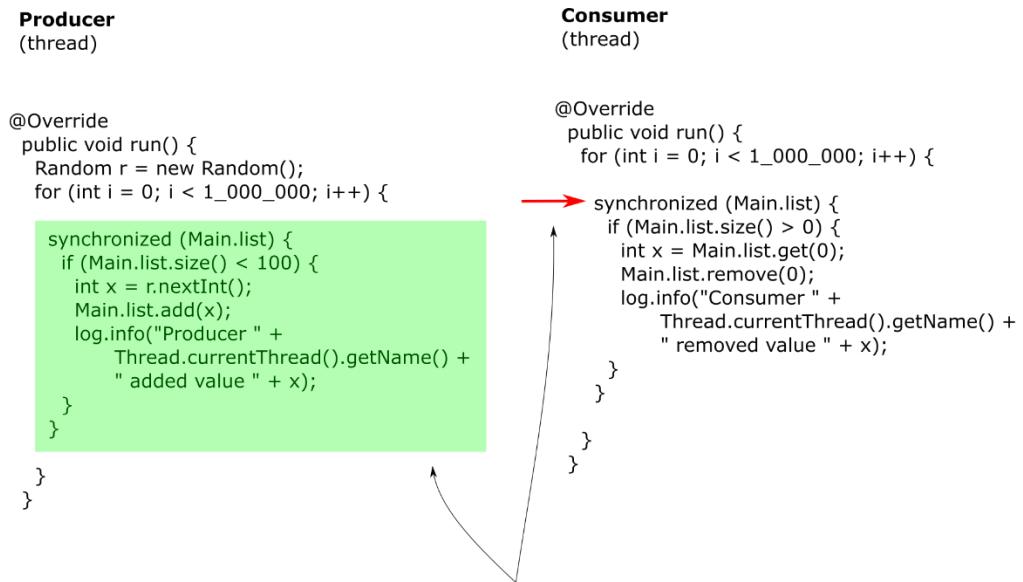
    public Producer(String name) {
        super(name);
    }

    @Override
    public void run() {
        Random r = new Random();
        for (int i = 0; i < 1_000_000; i++) {      #A
            synchronized (Main.list) {      #B
                if (Main.list.size() < 100) {      #C
                    int x = r.nextInt();      #D
                    Main.list.add(x);      #D
                    log.info("Producer " +      #E
                              Thread.currentThread().getName() +
                              " added value " + x);
                }
            }
        }
    }
}

```

#A Iterating 1 million times over the producer's synchronized block of code
#B Synchronizing the block of code using the static list defined in the Main class as a monitor.
#C Adding a value only if the list has under 100 elements.
#D Generating a new random value and adding it to the list.
#E Logging the value added to the list.

The producer's logic is also synchronized using the list as a monitor. This way, only one of the threads, the producer or the consumer, can change this list at a time. The monitor (the list instance) will allow one of the threads to enter its logic and keep the other thread waiting at the beginning of its block of code until the other finishes the execution of the synchronized block.



While the producer is executing the synchronized block (shaded rectangle), the consumer cannot access its synchronized block. The consumer waits for the monitor (list) to allow it enter its synchronized block.

Figure 9.2 Only one thread can be in either of the synchronized blocks at a time. Either the producer executes the logic defined in its run() method, or the consumer executes theirs.

Can we find out this app behavior and other details about the execution using a profiler? In a real-world app, the code might be much more complicated, so understanding what the app does just by reading the code would, in most cases, not be enough.



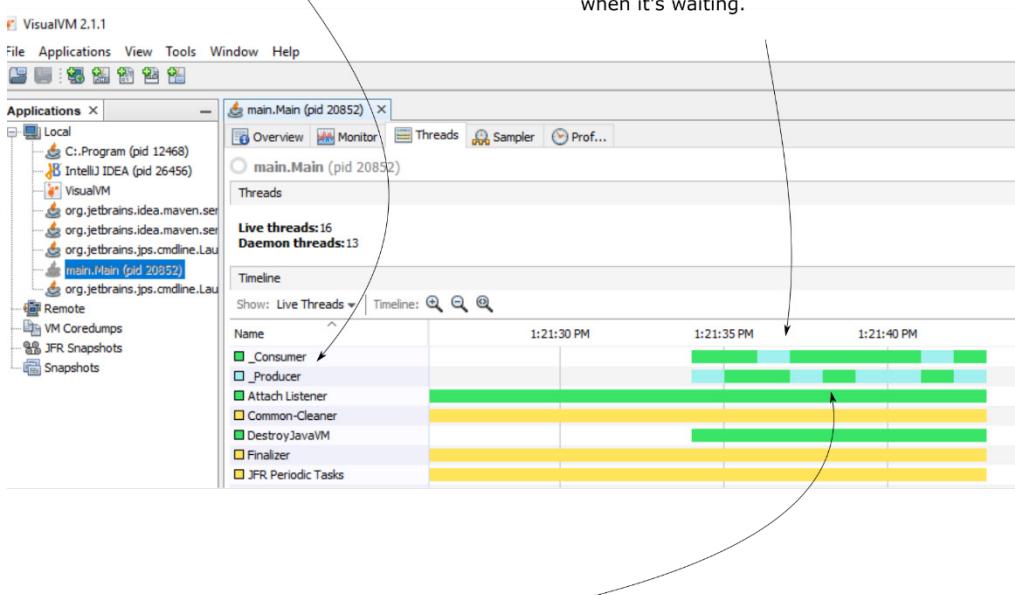
Remember that the projects we use in this book for our demonstrations are simplified and tailored to the purpose of our discussion. I don't want you to take them as best practices and apply them "as-is" in real-world apps.

Let's use VisualVM to observe how this looks like in the **Threads** monitoring tab (figure 9.3). Observe that the colors are mostly alternating since most of the code for each thread is synchronized. In most cases, either the producer is running and the consumer waits or the consumer is running and the producer waits.

Rarely, the two threads appear to execute code simultaneously. Since there're instructions outside the synchronized block, the two threads can be running simultaneously to execute the code. An example of such code is the "for" loop which in both cases is defined outside of the synchronized block.

You can observe the two threads (consumer and producer) executing on the timeline.

The timelines show alternative colors to indicate when the thread is running and when it's waiting.



Observe sometimes the threads run concurrently. They can run at the same time when they execute instructions that are outside the synchronized blocks.

Figure 9.3 In most cases, the thread will lock each other sequentially executing their synchronized blocks of

code. The two threads can still execute concurrently the instructions which are outside the synchronized block.

A thread can be blocked by a synchronized block of code, it can be waiting by another thread to finish its execution (joining), or it can be controlled by a blocking object. In cases where the thread is blocked by someone and it can't continue its execution, we say the thread is locked.

In figure 9.4 you can observe the same information presented in JProfiler. If you choose JProfiler instead of VisualVM you can still use the same approaches we discuss.

The executing threads are displayed on a timeline, using alternative colors to mark when they are running and when they are blocked.

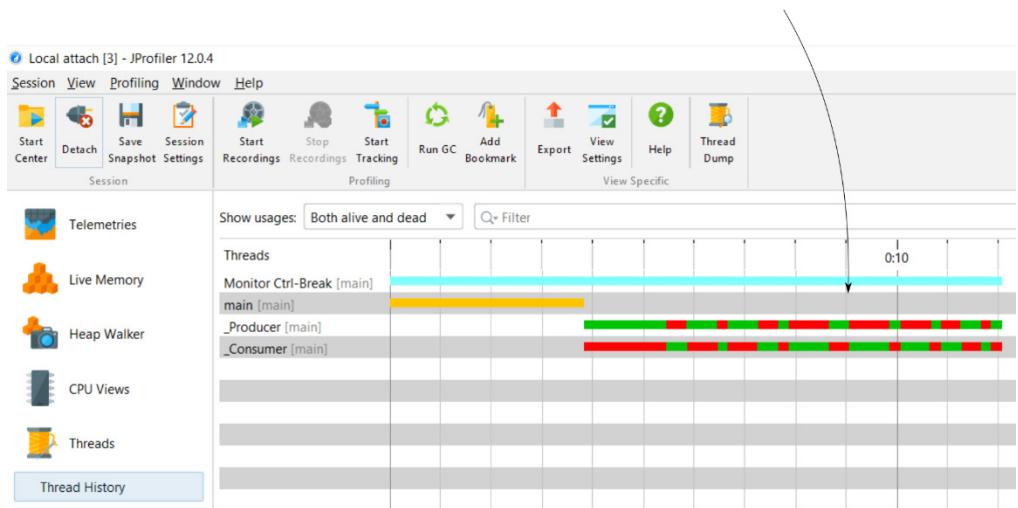


Figure 9.4 You can use other profilers instead of VisualVM. In this figure, you observe the way thread timelines are displayed in JProfiler.

9.2 Analyzing thread locks

When working with an app architecture that uses thread locks, we want to make sure that the app is optimally implemented. For that, we need a way to identify the locks to find out how many times threads are blocked and how long is the lock time. We also need to understand who causes a thread to wait in given scenarios. Can we collect all this information somehow? Yes, a profiler can tell us everything we need to know about the thread's behavior.

We'll continue using the same steps you learned in chapter 7 for profiling investigations:

1. **Sampling** to understand what happens during execution from the top-of-the-mountain and identify where to go into details further.
2. **Profiling** (instrumentation) to get the details on a specific subject we investigate.

Figure 9.5 shows the results of sampling the app's execution. When looking at the execution times, we observe that the total time is bigger than the total CPU time. In chapter 7 you saw a similar situation and we figured out that when something like this happens, it means the app waits for something.

For both threads, the total CPU time is much smaller than the total execution time. This indicates that the method was waiting for something.

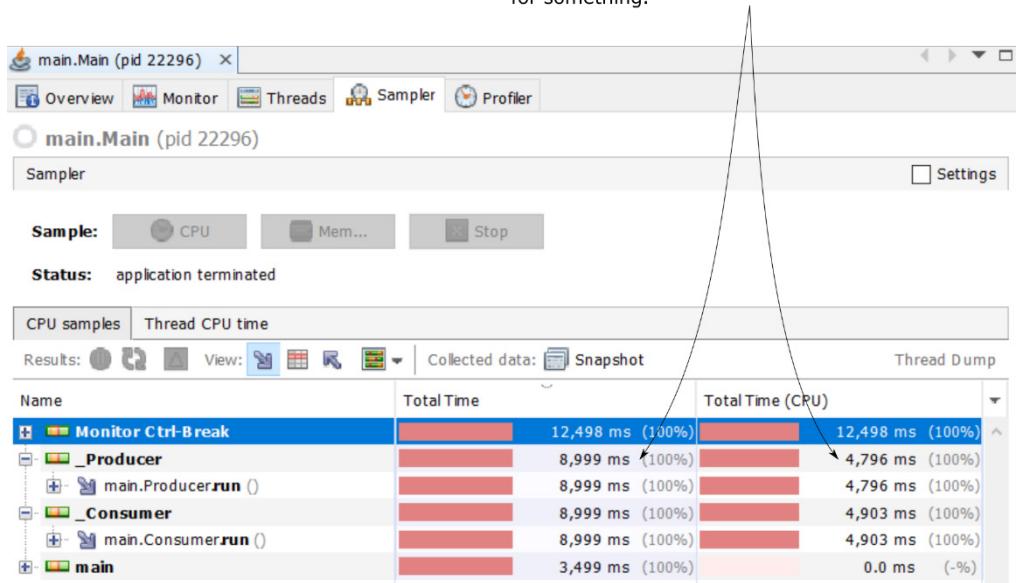


Figure 9.5 When the total CPU time is smaller than the total execution time, it means the app waits for something. We want to figure out what the app is waiting for and if this time can be optimized.

We go into more details to find out what is the method waiting for. In figure 9.6, we observe something interesting. The method waits, but as shown in the sampling data, it doesn't wait for something else. It simply seems it waits on itself. The row marked as "self time" tells us how much took the method itself to execute. Observe that the method spent only about 700 ms CPU time as "self time", but a much larger value of 4903 ms as total execution self time.

In chapter 7, we worked on an example where the app was waiting for an external service to respond. The app was sending a call and then waited for the other service to reply. In that case, it made sense why was the app waiting, but here the situation looks peculiar. What could cause such behavior?

You might wonder: "How can a method be waiting by itself? Is it too lazy to run?" When we observe such behavior, where a method is waiting but not for something external, most likely its thread has been locked. To get more details about what locks the thread, we need to analyze it more deeply by profiling the execution.

Going into more details, we observe that the method doesn't wait for something external. Its self execution time is very large even if the CPU time is small.

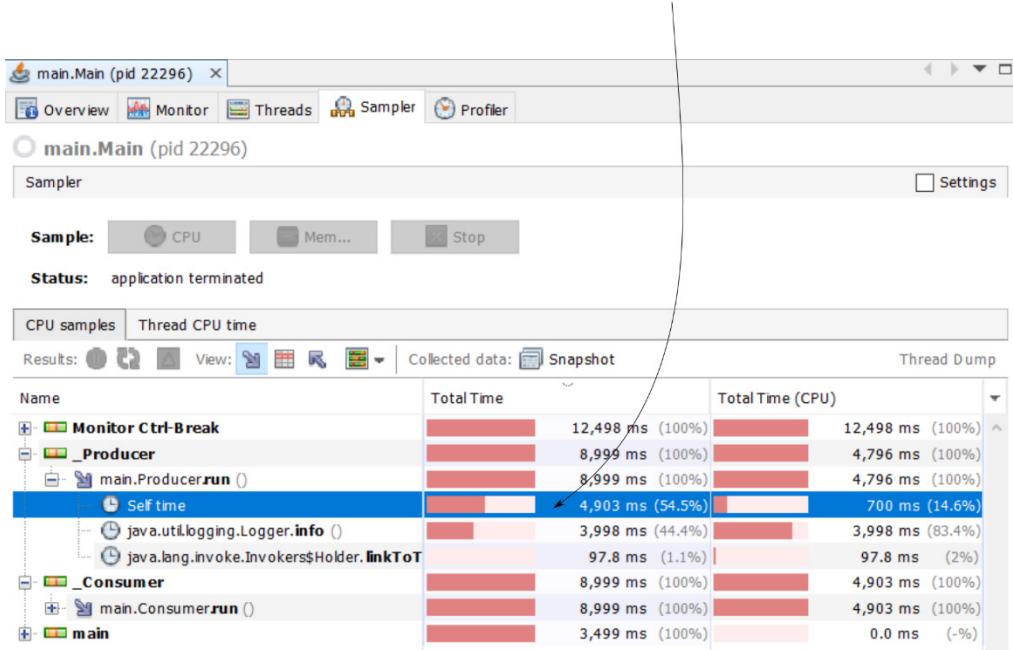


Figure 9.6 The method doesn't wait for someone, but instead it waits on itself. We observe that its self execution time is bigger than a total CPU time. Such a thing usually means that the thread was in a lock. The thread should have been blocked by another thread.

Sampling didn't answer all our questions. We observed the methods are waiting, but we don't know what are they waiting for. We need to continue with profiling (instrumentation) to get more details. In VisualVM, we use the **Profiler** tab to start locks monitoring. To start profiling for locks, you use the **Locks** button as presented in figure 9.7.

Figure 9.7 directly shows the profiling result after collecting the profiled data. The button appears disabled in the figure because the process was already stopped at the end of the profiling session.

To start profiling for data about locks, you use the Locks button. Once the session ends, the button becomes disabled.

We can observe that the threads have been blocked a large number of times. Each thread indicates over 3500 locks during the execution.

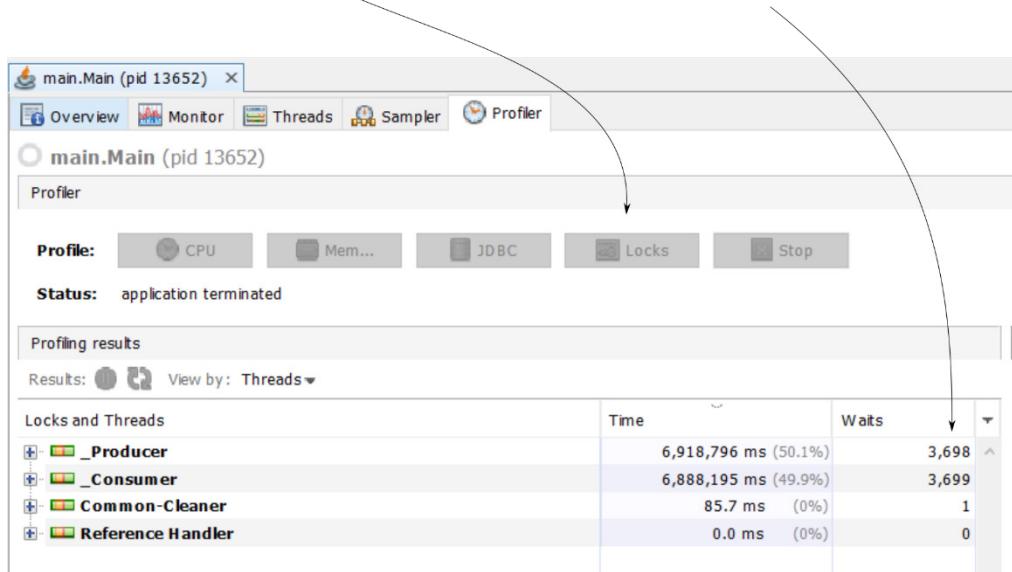


Figure 9.7 To start profiling for locks, use the Locks button in the Profiler tab. At the end of the profiling session, we observe more than 3600 locks on each of our producer and consumer threads.

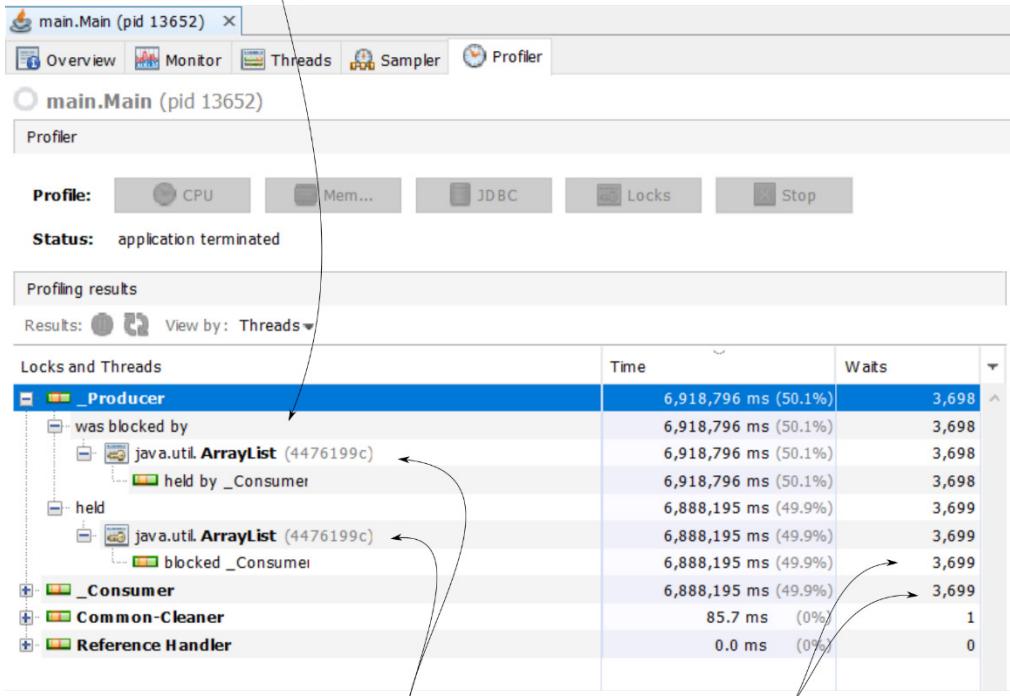
For each thread, we can go into detail by selecting the small **(+)** button on the left of the thread name. Now, you get details about each monitor object that affected the thread's execution. The profiler shows both details about the threads that were blocked by another thread, and also who blocked the thread.

In figure 9.8 you find these details. For the producer thread, we observed that it was blocked by a monitor instance of type `ArrayList`. The object reference (4476199c in the figure) helps us uniquely identify the object instance to figure out if the same monitor affected multiple threads and precisely identify the relationship between the threads and the monitor.

What we find in figure 9.8 can be read this way:

- The thread named “_Producer” was blocked by monitor instance with reference 4476199c – an instance of type `ArrayList`.
- The “_Consumer” thread blocked the “_Producer” thread 3698 times by acquiring the monitor 4476199c.
- The producer thread also held (owned) the monitor with reference 4476199c for 3699 times. This way, the thread with the name “_Producer” blocked the thread named “_Consumer” 3699 times.

Looking into detail, we find the objects (monitors), that caused the thread to be blocked as well as the monitors that the thread acquired.



In this case, we observe the same object (an instance of type ArrayList) blocked this thread but was also held by it.

Observe the number of locks the producer caused on the consumer is equal to the total number of times the consumer was locked, meaning only the producer locks the consumer.

Figure 9.8 The profiling results give us a good understanding of who creates and who is affected by locks. We observe there's only one monitor the producer thread works with. We can also observe that the producer thread was blocked by the consumer thread 3698 times using the monitor. Using the same monitor instance, the producer blocked the consumer also for a similar number of times: 3699.

Figure 9.9 extends the perspective to the consumer thread. You find that all data correlates. Throughout the whole execution, only one monitor instance, an instance of type ArrayList, locks either one of the threads or another. The consumer thread ends up being locked 3699 times while the producer thread was executing a block synchronized by the ArrayList object. The producer thread ended up being blocked 3698 times while the consumer thread was executing a block synchronized with the ArrayList monitor.



Remember that you won't necessarily get the same numbers when you execute the app on your computer. In fact, it's very likely you won't get the exact same numbers even when you repeat the execution on the same computer. You might get different values, but overall, you'll make similar observations.

Both threads (producer and consumer) held and were blocked by the same monitor. This shows that the threads alternatively block each other.

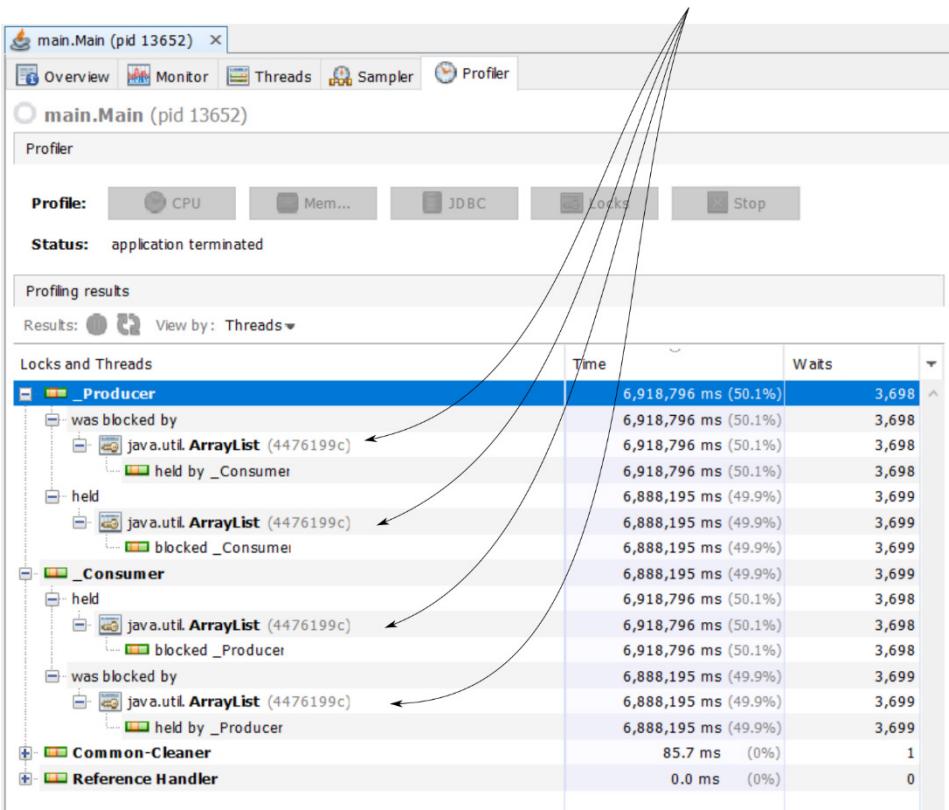


Figure 9.9 Both threads use the same monitor to block each other. While one thread executes the synchronized block with an ArrayList instance monitor, the other waits. This way, one thread ends up being locked for 3698 times and the other for 3699.

For this demonstration, I used VisualVM because it's free and it usually comes very comfortable for me to use anywhere. But the same approaches can be done with other tools as well. Let's use JProfiler to achieve the same details about locks.

After attaching JProfiler to a process (as discussed in chapter 8), make sure you set the JVM exit action to "Keep the VM alive for profiling" as presented in figure 9.10.

When you attach JProfiler to a process, configure the JVM exit action to keep the VM alive so you can still see the statistics after the profiled app ends its execution.

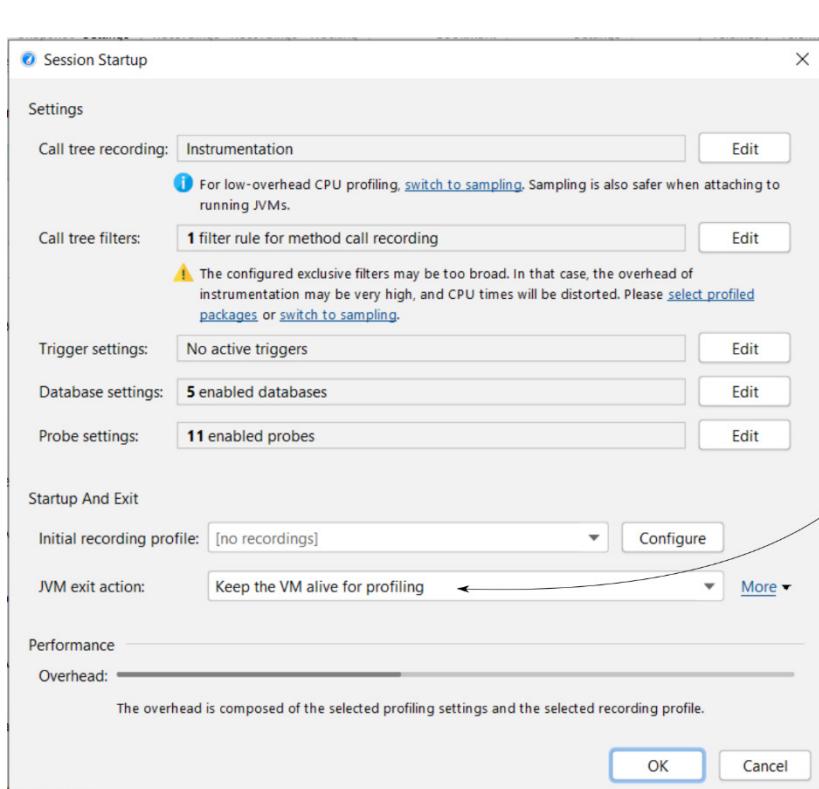


Figure 9.10 When starting the profiling session with JProfiler, remember to set the JVM action to keep the VM alive so you can see the profiling results after the app finishes its execution.

JProfiler offers multiple perspectives for visualizing the same details we also got with VisualVM. But in the end, as you'll observe, we'll get the same results.

JProfiler shows a complete history of the lock events. You find details about the lock duration, the monitor that has been used, the thread that acquired the lock (owning thread), the thread that was blocked (waiting thread) and the exact time of the event.

To access the lock history in JProfiler, select Monitor History under Monitors & Locks in the left menu.

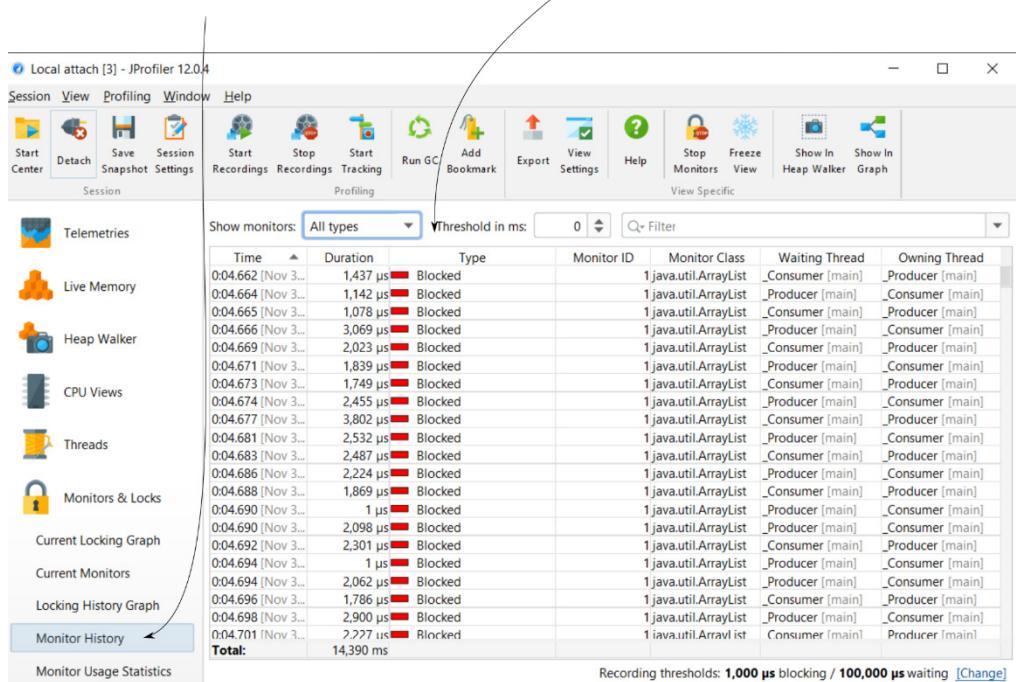


Figure 9.11 JProfiler shows a detailed history of all the locks the app's threads encountered. The tool displays the exact time of the event, the event duration, the monitor that caused the lock, and the threads that were involved.

In most cases, I didn't need such a detailed report. I prefer to group the events (locks) either by threads, or, less often, by the monitor. In JProfiler you can group the events as presented in figure 9.12. From the **Monitor Usage Statistics** in the left menu, you can choose to group the events either by threads that were involved, the monitors that caused the locks. JProfiler even has a more exotic option where you can group the locks by the monitor objects' classes – I can't say I used this one often or at all.

In JProfiler, you can use the Monitor Usage Statistics section to get information about locks grouped by affected threads or by the monitor that caused the lock.

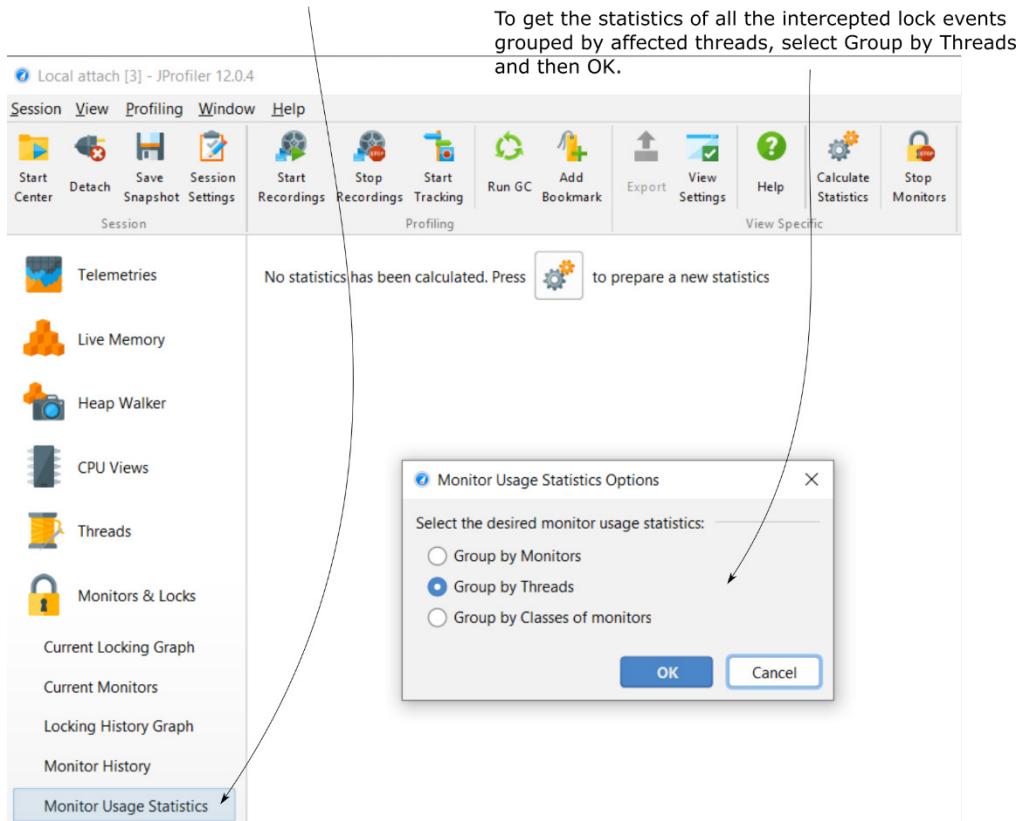


Figure 9.12 You can group the lock events by threads involved or monitors using the Monitor Usage Statistics section in the app. You can use the aggregated view to understand which threads are more affected and who affects them or which monitor causes the threads to stop more often.

If you group the lock events by involved threads, you'll get a statistic similar to the one VisualVM provided. Each thread is locked over 3500 times during the app's execution.

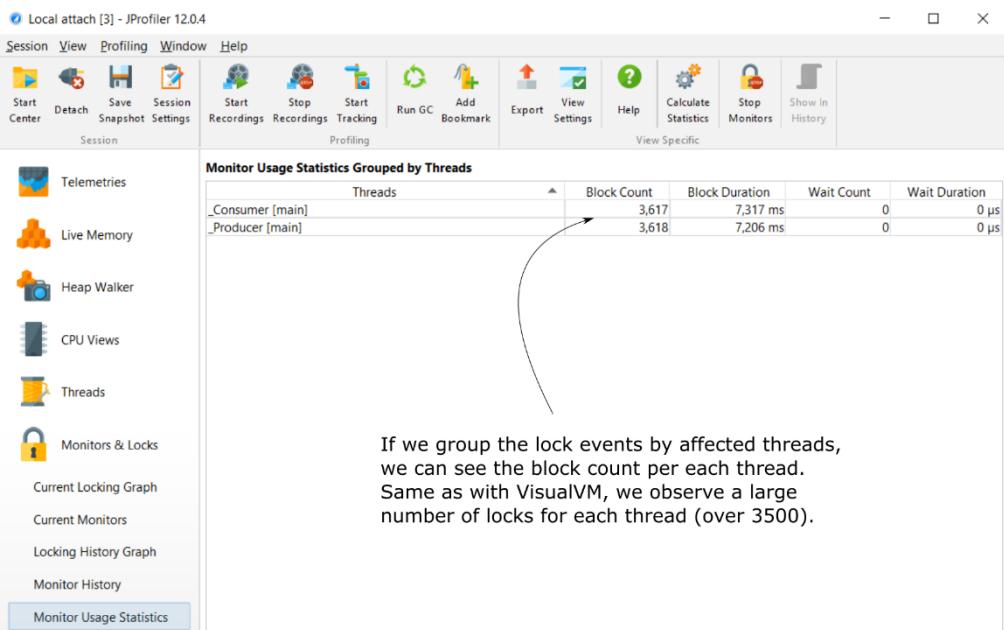


Figure 9.13 Grouping the lock events by threads you will get an aggregated view telling how many times was each of the threads locked during its execution.

Is the execution optimal? To answer such a question for an app we need to know what's the app's purpose. In our case, the app is a simple, demonstrative example that helps us focus on the investigation techniques we discuss. Not having a real purpose, it's difficult to make a real analysis on whether the results we got say that the app can be enhanced.

But, since the app uses two threads that are using a common resource (the list), if we consider they can't work at the same time with the shared resource, then what we expect is:

- The total execution time should be approximatively the sum of CPU execution times (because the threads can't work at the same time, they will mutually exclude each other).
- The threads should have a similar time allocated for execution and should be locked approximatively the same number of times. If one of the threads would be preferred, then the other could end up in starvation. **Starvation** is the situation in which a thread is blocked in an "unfair" way and doesn't get to execute because of this.

If you take a look again at the thread analysis, we observe that the two threads are fairly treated. They indeed get locked a similar number of times and they mutually exclude each other, but get to have a similar active (CPU time) execution.

In this case, the execution looks to be optimal without much we can do to enhance it. However, remember that we don't have one recipe we can apply to say if the execution is optimal or not for all the cases. It really depends on what the app does and our expectations about how should it execute.

Let me give you also an example of a different scenario where the app would not necessarily be considered optimal:

You can clearly see in our case that the producer only generates a random value and the consumer removes it. The threads don't really do a lot of things since this is just a small example designed to allow you to focus on the chapter's purpose which is using the profiler to analyze threads, and not on implementing concurrency.

But suppose now that you had an app that was actually processing values. Say that the producer needed more time to add each value to the list than the consumer would need to process the value afterward. In a real-world app, something like this can happen – the threads don't need to have equivalently difficult "work" to do.

So in such a case, you would maybe find a way to enhance the app by

- minimizing the number of locks for the consumer and making it wait, to allow the producer to work more.
- moreover, you could define more producer threads or you could make the consumer thread read and process the values in batches (multiple at a time).

Everything depends on what the app does, but understanding what you can do to make it better starts with analyzing the execution. Because you never have one approach that you can apply to all apps, I always recommend developers to use a profiler and analyze the changes in app execution anytime they implement a multithreaded app.

9.3 Analyzing waiting threads

In this section, we analyze threads that are waiting to be notified. Waiting threads are different than locked threads (which we discussed in section 9.2). A monitor locks a thread for the execution of a synchronized block of code. In this case, we don't expect the monitor to execute a specific action to "tell" the blocked thread to continue its execution. But a monitor can make the thread wait for an indefinite time and decide later when to allow that thread to continue its execution. Once a monitor makes a thread wait, the thread will return to execution, only after being notified by the same monitor. The ability to make a thread wait until being notified gives great flexibility to control threads, but it can also cause issues when not appropriately used.

To easily visualize the difference between locked and waiting threads, take a look at figure 9.14. Imagine the synchronized block is a restricted area managed by a police officer. The threads are cars. The police officer only allows one car at a time to run in the restricted area (the synchronized block). The cars that are blocked because they wait to be allowed to enter the restricted area – we say they are **locked**. The police officer can also manage the cars running in the restricted area. The police officer can order a car running inside this area to wait until they explicitly order them to continue further. When a car is ordered explicitly to take a break by a police officer, we say it's **waiting**.

These threads are in a blocked state. They cannot continue execution while another thread is running inside the synchronized block. We say they are **locked**.

Locked threads



(talking to the cars before the synchronized block)
You there! You have to wait. Someone else is currently executing the synchronized block.

`synchronized() {`



This thread is running inside the synchronized block. The monitor (police officer) won't allow other threads to go into the synchronized block until this one steps out of it.



`}`

Waiting threads

`synchronized() {`



(talking to the car inside the synchronized block)
You there! You'll have to wait until I tell you that you can continue your execution.



This thread is running inside the synchronized block. The monitor (police officer) pauses and moves it to the blocked state. Following the monitor's actions, we say the thread is **waiting**.

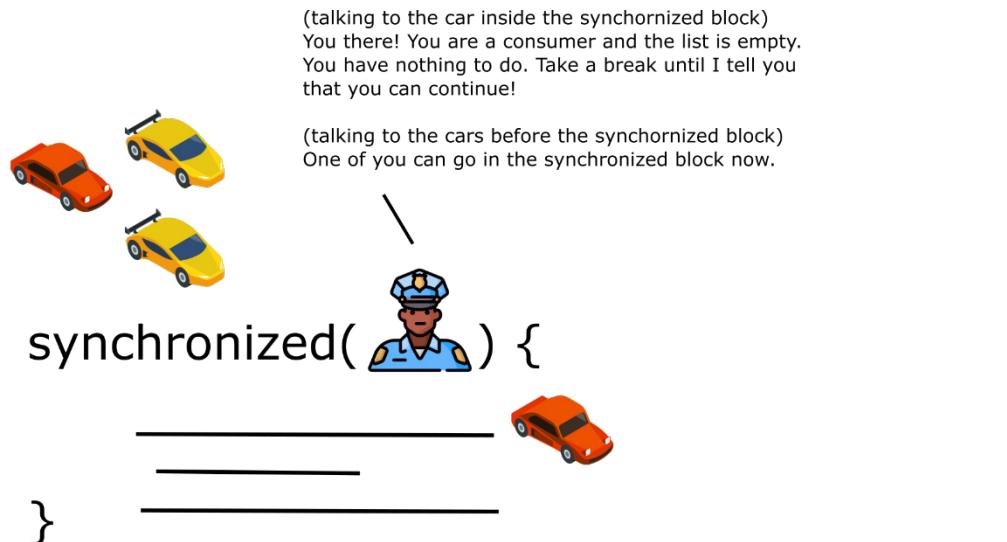
`}`

Figure 9.14 Locked threads vs. waiting threads. A locked thread is blocked at the entrance of a synchronized block. The monitor won't allow a thread to enter a synchronized block while another thread actively runs inside the block. A waiting thread is a thread that has been explicitly set to the blocked state by the monitor. The monitor can make any thread inside the synchronized block it manages to wait. The waiting thread can continue its execution only after the monitor explicitly tells it that it can proceed with its execution.

We'll use the same application we analyzed earlier in this chapter and we'll consider the following scenario: One of the developers working on the app thought about an improvement to our producer-consumer architecture. Now, the consumer thread can't do anything when the list is empty, so it just iterates multiple times over a false condition until the JVM puts it to sleep to allow a producer thread to run and add values to the list. The same thing happens

when the producer adds 100 values to the list. The producer thread will run over a false condition until the JVM allows a consumer to remove some of the values from the list.

Can we do something to make the consumer wait when it has no value to consume and make it run only when we know the list contains at least one value (figure 9.15)? Same for the producer, can we make it wait when there're already too many values in the list and allow it to run only when it makes sense it adds other values to the list. And would this approach make our app more efficient?



A few moments later after a producer adds a value to the list ...

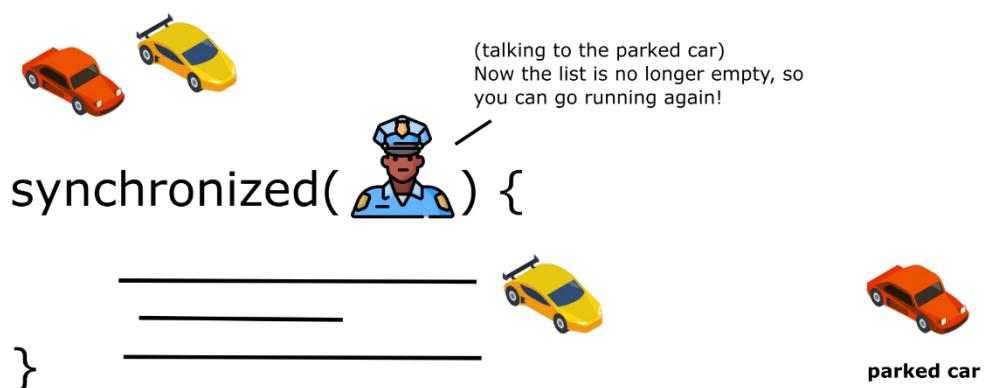
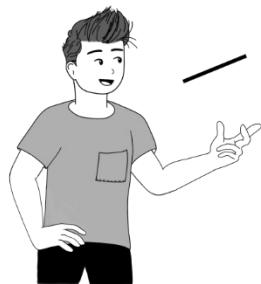


Figure 9.15 Some of the cars are consumer threads and others are producer threads. The police officer orders

a consumer to wait if the list doesn't have values that can be consumed, allowing this way producers to work and add values. Once the list contains at least a value that can be consumed, they order the waiting consumer to continue their execution.

We'll change the application to implement this new behavior, but we'll also demonstrate that for our scenario, the app isn't more efficient. On the contrary, the execution is less optimal.

It might look like a good idea to make the threads wait when they couldn't work anyway with the shared resource (the list). But analyzing the execution, you'll see that it rather badly affects the performance instead of helping the app run faster. When working with multiple threads, things don't always behave how we would expect.



I always recommend using a profiler during development to prove that the app executes optimally.

Listing 9.4 shows the new implementation of the consumer thread. The consumer thread waits when the list is empty since it would anyway have nothing to consume. The monitor makes the consumer thread wait, and will notify it to continue its execution only after a producer adds something to the list. We used the `wait()` method to tell the consumer to wait if the list is empty. At the same time, when the consumer removes values from the list, it notifies the waiting threads so if a producer is waiting, it now knows it can continue its execution because the list is no longer full. We use the `notifyAll()` method to notify the waiting threads. You find this implementation in project da-ch9-ex2.

Listing 9.4 Making the consumer thread wait when the list is empty

```
public class Consumer extends Thread {
    // Omitted code
    @Override
    public void run() {
        try {
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (Main.list) {
                    if (Main.list.size() > 0) {
                        int x = Main.list.get(0);
                        Main.list.remove(0);
                        log.info("Consumer " +
                                Thread.currentThread().getName() +
                                " removed value " + x);
                        Main.list.notifyAll();    #A
                    } else {
                }
            }
        }
    }
}
```

```

        Main.list.wait();      #B
    }
}
} catch (InterruptedException e) {
    log.severe(e.getMessage());
}
}
}

#A After consuming an element from the list, the consumer notifies the waiting threads telling them a change has
been made in the list contents.
#B When the list is empty, the consumer waits until it gets notified something has been added to the list.

```

Listing 9.5 shows the producer thread implementation. Similar to the consumer, the producer thread waits if there're too many values in the list. A consumer will eventually notify the producer and allow it to run again when it consumes a value from the list.

Listing 9.5 Making the producer thread wait if the list is already full

```

public class Producer extends Thread {

    // Omitted code

    @Override
    public void run() {
        try {
            Random r = new Random();
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (Main.list) {
                    if (Main.list.size() < 100) {
                        int x = r.nextInt();
                        Main.list.add(x);
                        log.info("Producer " +
                                Thread.currentThread().getName() +
                                " added value " + x);
                        Main.list.notifyAll();      #A
                    } else {
                        Main.list.wait();        #B
                    }
                }
            }
        } catch (InterruptedException e) {
            log.severe(e.getMessage());
        }
    }
}

```

#A After adding an element to the list, the producer notifies the waiting threads telling them a change has been made in the list contents.

#B When the list has 100 elements, the producer waits until it gets notified something has been removed from the list.

As you already know, we start our investigations by sampling the execution. We already observe something suspicious, the execution seems to take much longer (figure 9.16). If you go back to the previous observations we made in section 9.1, you'll see that the whole

execution was only about 9 seconds previously. Now, the execution takes about 50 seconds. A huge difference from the previous execution time.

The execution takes a longer time and there is still a big difference between the total time and the total CPU time, indicating that the app still waits a lot.

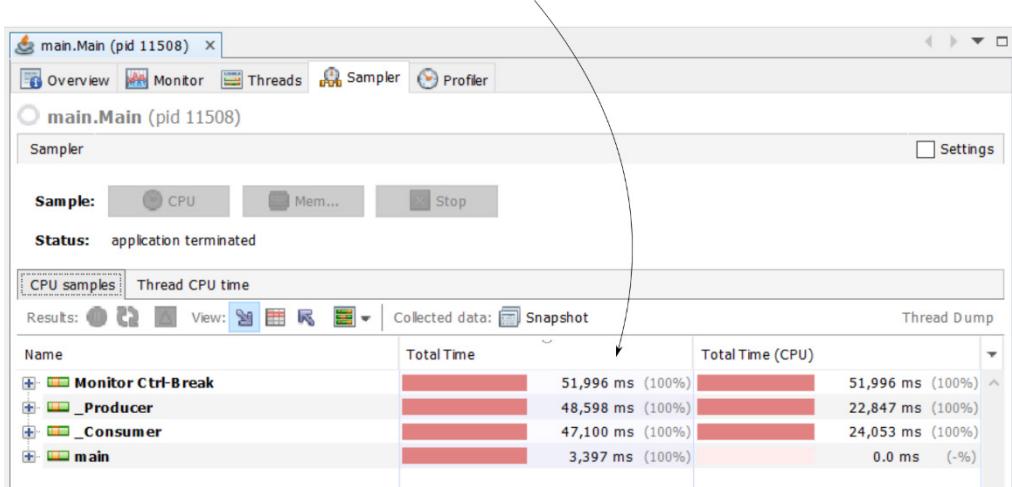


Figure 9.16 Sampling the execution, we observe that the execution time is slower than before we made threads wait.

Sample details (figure 9.17) show us that the `wait()` method we added caused most of the thread waiting time. The thread is not locked that much time anymore since the self-execution time is very close to the self CPU execution time. Still, our purpose was to make our app overall more efficient, it seems we only shifted the waiting from one side to another and we made the app slower overall.

The execution details indicate that most of the waiting time is caused by the wait() method the monitor invokes.

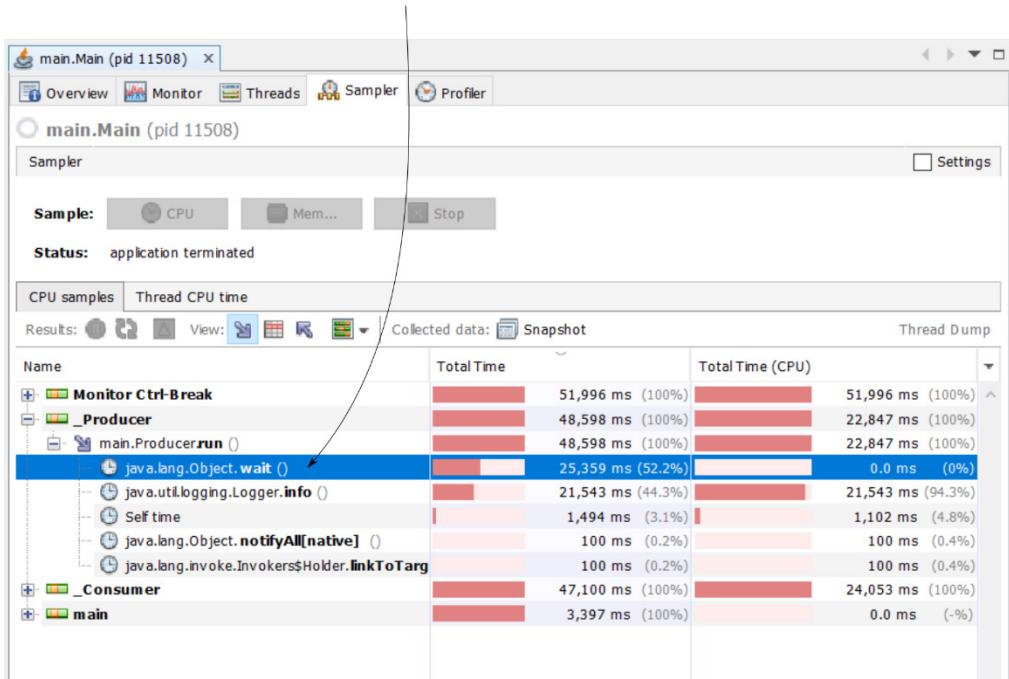


Figure 9.17 Looking into details, we observe that the self-execution time is not that much larger anymore, but the thread is blocked waiting for a longer time.

We continue the investigation profiling for more details (figure 9.18). Indeed, the profiling results show us fewer locks, but that doesn't help a lot, since the execution is much slower.

Observe that the number of locks decreased.
Even so, the total execution time grew.

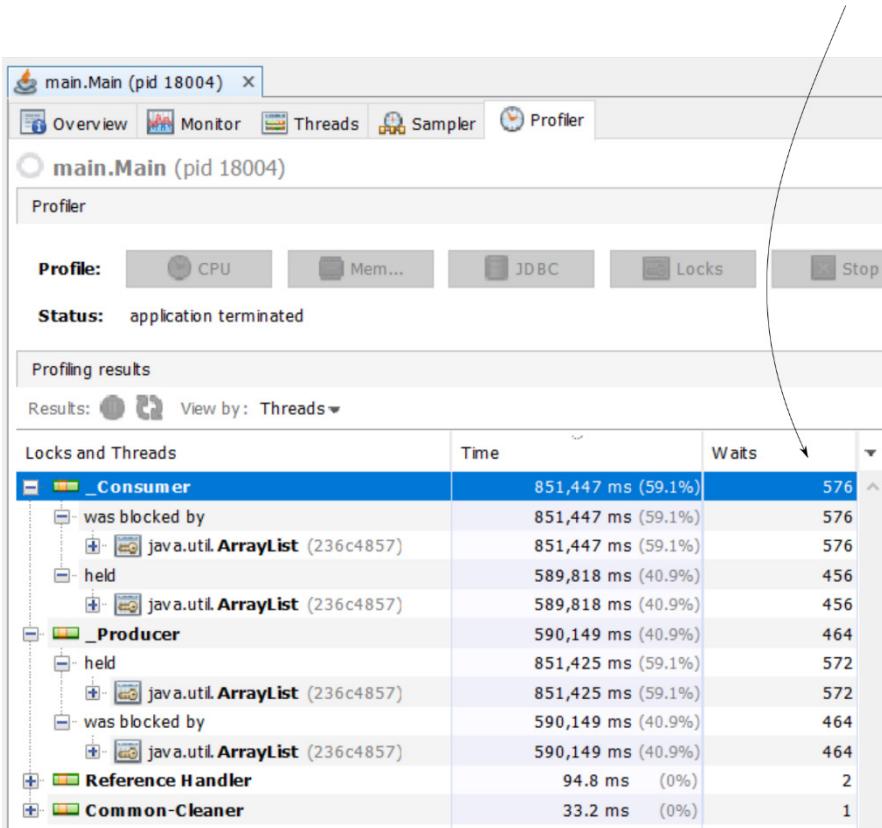


Figure 9.18 The lock pattern is similar to what we have before, but the threads are locked for a smaller number of times.

Figure 9.19 shows you the same investigation details obtained using JProfiler. In JProfiler, once grouping the lock events by threads, you will get both the number of locks and the waiting time. In the previous exercise, the waiting time was 0 but we had many more locks. Now, we have fewer locks but a larger waiting time. This tells us that the JVM changes more slowly between threads when using a wait/notify approach than allowing the threads to get naturally locked and unlocked by the monitor of a synchronized block.

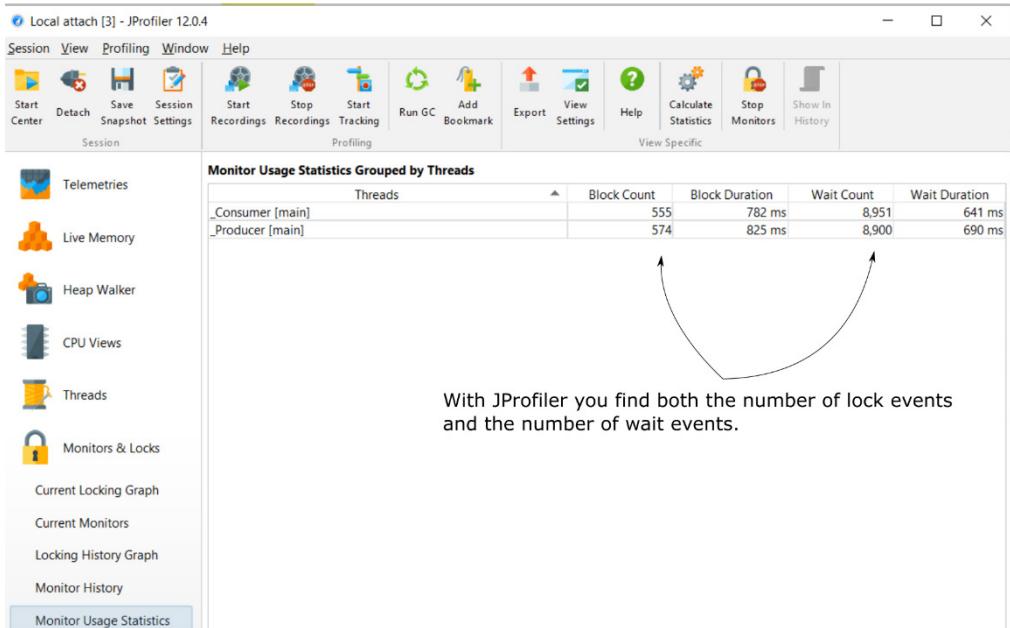


Figure 9.19 Using JProfiler, we observe the same details. The threads are locked for a smaller number of times, but now they are blocked in waiting for a much longer time.

9.4 Summary

- A thread can be locked and forced to wait by a synchronized block of code. Locks appear when threads are synchronized to avoid them changing shared resources at the same time.
- Locks are needed to avoid race conditions, but sometimes apps use faulty thread synchronization approaches which can lead to undesired results. When threads are not optimally synchronized, locks can lead to performance issues or even app freezes (in the case of deadlocks).
- Locks caused by synchronized code blocks slow down the app's execution because they force threads to wait instead of letting them "work". Locks may be needed in certain implementations, but we prefer to find ways to minimize the time an app's threads are locked.
- We can use a profiler to identify when locks slow down an app, how many locks the app encounters during execution, and how much they slow down the performance.
- When using a profiler, you should always sample the execution first. By sampling, you'll find out if the app's execution is affected by locks at all. You'll usually identify locks when sampling by observing that a method is waiting on itself when locks affect thread execution.
- If using sampling, you find out locks might affect the app's execution you can continue investigating using lock profiling (instrumentation). Lock profiling will show

the threads affected, the number of locks, monitors involved, and the relationship between locked threads and threads that cause the locks. These details help you decide if the app's execution is optimal or you can find ways to enhance it.

- Each app has a different purpose, so there's no unique recipe in understanding the thread lock analysis. In general, we are looking to minimize the time threads are locked or waiting and make sure threads are not unfairly excluded from execution (starving threads).

10

Investigating deadlocks with thread dumps

This chapter covers

- Getting thread dumps using a profiler
- Getting thread dumps in command line
- Reading thread dumps to investigate issues

In this chapter, we discuss using thread dumps for analyzing the thread execution at a given moment in time. Often, we use thread dumps in situations where the application becomes unresponsive, such as in the case of a deadlock. A deadlock is a situation where multiple threads pause their execution waiting for each other to fulfill a given condition. If a hypothetical thread A ends up waiting for thread B to do something and thread B waits for thread A, neither of them can continue their execution. In such a case, the app, or at least a part of it, will freeze. We need to know how to analyze such an issue to find its root cause and eventually solve the problem.

Because with a deadlock the process might completely freeze, you usually can't use sampling or profiling (instrumentation) anymore as we did in chapter 9. Instead, you can get a statistic of all the threads and their states for a given JVM process. This statistic that indicates the details of the thread is named a **thread dump**.

10.1 Getting a thread dump

In this section, we analyze ways to obtain a thread dump. We'll use a small application that implements a problem causing deadlocks on purpose. You find this app provided with the book as project da-ch10-ex1. We'll run this app and wait for it to freeze (this should happen

in a few seconds), and then we'll discuss multiple ways to get thread dumps. Once we know how to obtain the thread dumps, we discuss in section 10.2 how to read them.

Let's take a look at how the app that we'll use is implemented and why its execution causes deadlocks. The app uses two threads to change two shared resources (two list instances). A thread named the "producer" adds values to one list or another during execution. Another thread named the "consumer" removes values from these lists. If you've already read chapter 9, you might recall we worked there on a similar app. But since the app's logic is irrelevant for our example, I've omitted it from the listings and I only kept the part relevant to our demonstration – the synchronized blocks.

The example is simplified to allow you to focus on the investigation techniques we discuss. In a real-world app, the things usually get more complicated. Also, wrongly used synchronized blocks are not the only way to get into deadlocks. Faulty use of blocking objects such as semaphores, latches or barriers can also cause such problems. But the steps you'll learn to investigate the issues are the same.

In listings 10.1 and 10.2, you'll observe that the two threads use nested synchronized blocks with two different monitors named "listA" and "listB". The problem is that one of the threads uses monitor "listA" for the outer synchronized block, while "listB" is used for the inner. The other thread uses them the other way around. Such a code design leaves room for deadlocks and is shown visually in figure 10.1.

Listing 10.1 Using nested synchronized blocks for the consumer thread

```
public class Consumer extends Thread {
    // Omitted code

    @Override
    public void run() {
        while (true) {
            synchronized (Main.listA) {      #A
                synchronized (Main.listB) {      #B
                    work();
                }
            }
        }
        // Omitted code
    }
}
```

#A The outer synchronized block uses the listA monitor.

#B The inner synchronized block uses the listB monitor.

In listing 10.1 you observed that the consumer thread used "listA" as the monitor for the outer synchronized block. In listing 10.2 you see the producer thread uses the same monitor for the inner block, while the "listB" monitor is also swapped between the two threads.

Listing 10.2 Using nested synchronized blocks for the producer thread

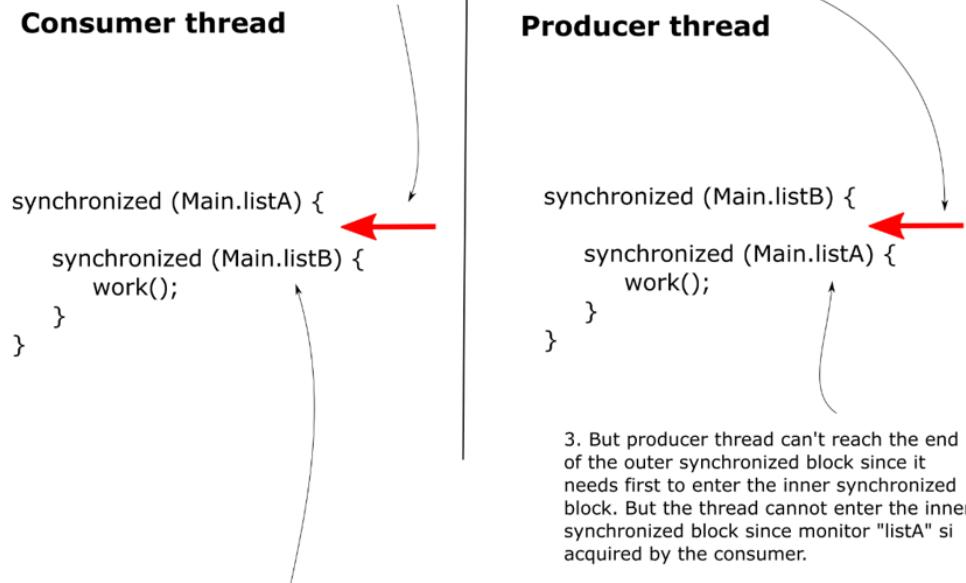
```
public class Producer extends Thread {  
    // Omitted code  
  
    @Override  
    public void run() {  
        Random r = new Random();  
        while (true) {  
            synchronized (Main.listB) {      #A  
                synchronized (Main.listA) {    #B  
                    work(r);  
                }  
            }  
            // Omitted code  
        }  
    }  
}
```

#A The listB monitor is used by the outer synchronized block

#B The listA monitor is used by the inner synchronized block

Figure 10.1 shows how the two threads can run into a deadlock.

1. Suppose that while running the two threads get into a situation where each of them entered the outer synchronized block but haven't gone into the inner synchronized block yet. The arrows indicate where each thread is during the execution.



2. In such a case, neither thread can continue its execution. The consumer cannot continue into the inner synchronized block, since monitor "listB" is acquired by the producer thread. Monitor "listB" should first be released, meaning that the producer thread should reach the end of the block.

3. But producer thread can't reach the end of the outer synchronized block since it needs first to enter the inner synchronized block. But the thread cannot enter the inner synchronized block since monitor "listA" is acquired by the consumer.

Figure 10.1 If both threads enter the outer synchronized block but not the inner one, they remain stuck waiting for one another. We say that they went into a deadlock.

10.1.1 Getting a thread dump using a profiler

What do we do when we have a frozen app and we want to identify the problem's root cause? Using a profiler to analyze the locks most likely doesn't work in such a scenario where the app or a part of it is frozen. Instead of analyzing the locks during execution, as we did in chapter 9, we'll take just a snapshot of the app's thread states. We'll read this snapshot, named **thread dump**, and find which threads affect each other and cause the app to freeze.

You can obtain a thread dump using either a profiler tool (for example VisualVM or JProfiler) or directly calling in a command line a tool provided by the JDK. In this section, we'll discuss how to obtain a thread dump using a profiler and we'll continue in section 10.1.2 by getting the same information in the command line.

We start our application (project da-ch10-ex1) and wait a few seconds for it to enter a deadlock. You'll know the app gets into a deadlock when it doesn't write any more messages in the console (it gets stuck).

Getting the thread dump using a profiler is a comfortable and simple approach. It's no more than a click on a button. Let's take for example using VisualVM to get a thread dump. Figure 10.2 shows the Visual VM interface. You can observe that VisualVM is smart enough and even figured out itself that some of the threads of our process ran into a deadlock. VisualVM indicates this aspect in the **Threads** tab.

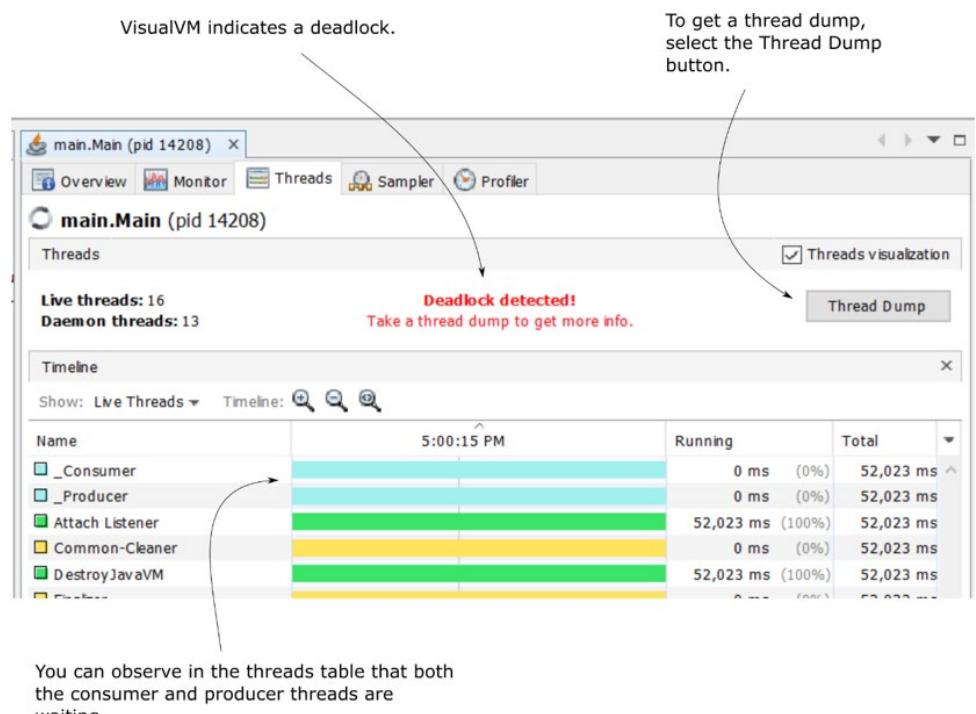
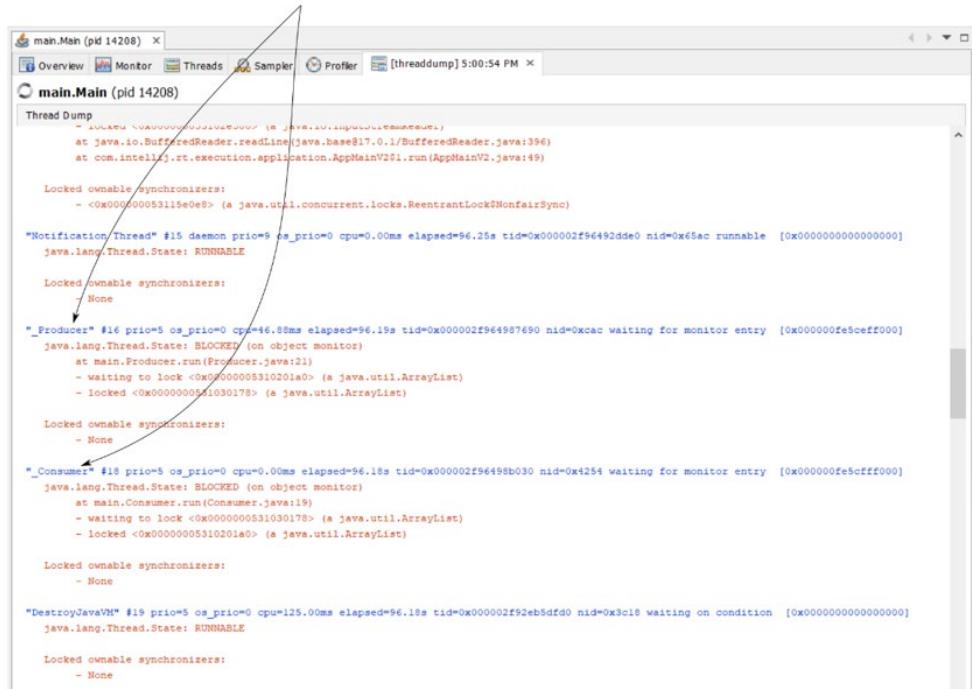


Figure 10.2 When some of the app's threads get into a deadlock, VisualVM indicates the situation with a message in the Threads tab. You can easily observe that both the `_Consumer` and `_Producer` threads are locked on the graphic timeline. To get a thread dump, you simply select the Thread Dump button in the window's right upper corner.

After collecting the thread dump, it appears as shown in figure 10.3. As you can observe, the thread dump is represented as plain text describing the app threads and details about them (such as their state in the life cycle, who blocks them, and so on). We'll discuss more about how to read the thread dump in section 10.2.

The thread dump shows information about each active thread.
You will find the producer and consumer threads in the generated thread dump.



```

main.Main (pid 14208) X
Overview Monitor Threads Sampler Profiler [threaddump] 5:00:54 PM X

main.Main (pid 14208)
Thread Dump
  - BLOCKED <0x000000053115e0e8> in java.util.concurrent.locks.ReentrantLock$NonfairSync
    at java.io.BufferedReader.readLine(BufferedReader.java:396)
    at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:49)

  Locked ownable synchronizers:
  - <0x000000053115e0e8> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

  "Notification Thread" #15 daemon prio=5 os_prio=0 cpu=0.00ms elapsed=96.25s tid=0x000002f96498dde0 nid=0x65ac runnable [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

  Locked ownable synchronizers:
  - None

  "_Producer" #16 prio=5 os_prio=0 cpu=46.88ms elapsed=96.19s tid=0x000002f964987690 nid=0xcac waiting for monitor entry [0x000000fe5ceff000]
  java.lang.Thread.State: BLOCKED (on an object monitor)
    at main.Producer.run(Producer.java:21)
    - waiting to lock <0x00000005310201a0> (a java.util.ArrayList)
    - locked <0x0000000531030178> (a java.util.ArrayList)

  Locked ownable synchronizers:
  - None

  "_Consumer" #18 prio=5 os_prio=0 cpu=0.00ms elapsed=96.18s tid=0x000002f96498b030 nid=0x4254 waiting for monitor entry [0x000000fe5cff000]
  java.lang.Thread.State: BLOCKED (on an object monitor)
    at main.Consumer.run(Consumer.java:19)
    - waiting to lock <0x0000000531030178> (a java.util.ArrayList)
    - locked <0x00000005310201a0> (a java.util.ArrayList)

  Locked ownable synchronizers:
  - None

  "DestroyJavaVM" #19 prio=5 os_prio=0 cpu=125.00ms elapsed=96.18s tid=0x000002f92eb5fdf0 nid=0x3c18 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

  Locked ownable synchronizers:
  - None

```

Figure 10.3 A thread dump is plain text describing an app's threads. In the thread dump we collected, we can find also the two deadlocked threads `_Consumer` and `_Producer`.



You might not understand this text at first.
Later in this chapter, you'll learn to read it.

10.1.2 Generating a thread dump from the command line

A thread dump can also be obtained using the command line. This approach is particularly useful when you need to get a thread dump from a remote environment. Most often, you don't have access to remote profile an app installed in an environment, not to say that remote profiling (and remote debugging as well) isn't even recommended for a production environment (as discussed in chapter 4). And since in most cases you can only access a remote environment through a command line you need to know how to get a thread dump this way too.

Fortunately, getting a thread dump using the command line is quite easy. Further in this section, we'll discuss and apply the steps to get a thread dump in the command line for our demonstrative app.

The steps we'll follow are:

1. Find the process ID for which we want to get a thread dump.
2. Get the thread dump as text data (raw data) and save it in a file.
3. Load the saved thread dump to read it easier in a profiler tool.

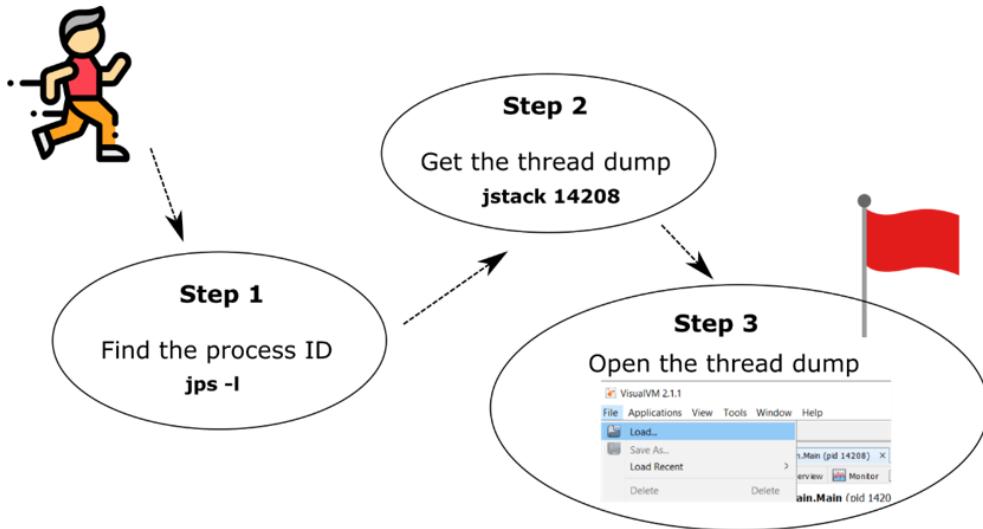


Figure 10.4 We'll follow three simple steps to get a thread dump using the command line. First, we'll find the process ID to get the thread dump for. Secondly, we'll use a JDK tool to get the thread dump. Finally, we'll open the thread dump in a profiler tool to read it.

STEP 1 – FIND THE PROCESS ID FOR THE PROCESS YOU WANT TO OBTAIN A THREAD DUMP

Up to now, using a profiler we identified the process to profiler using its name (represented as the main class's name). But when getting a thread dump using the command line, you need to identify the process using its ID. How to get a process ID (PID) for a running Java app? The simplest way is using the “jps” tool provided with the JDK. The next snippet shows the command you need to run. We use the “-l” (lowercase “L”) option to get the main class names associated with the PIDs. This way, we can identify the processes the same as we did in chapters 6 through 9 when you learned to profile an app’s execution.

```
jps -l
```

Figure 10.5 shows the result of running the command. The numeric values in the output’s first column are the process IDs (PIDs). The second column associates the main class name to each PID. This way, we get the PID that we’ll use in step 2 to get the thread dump.

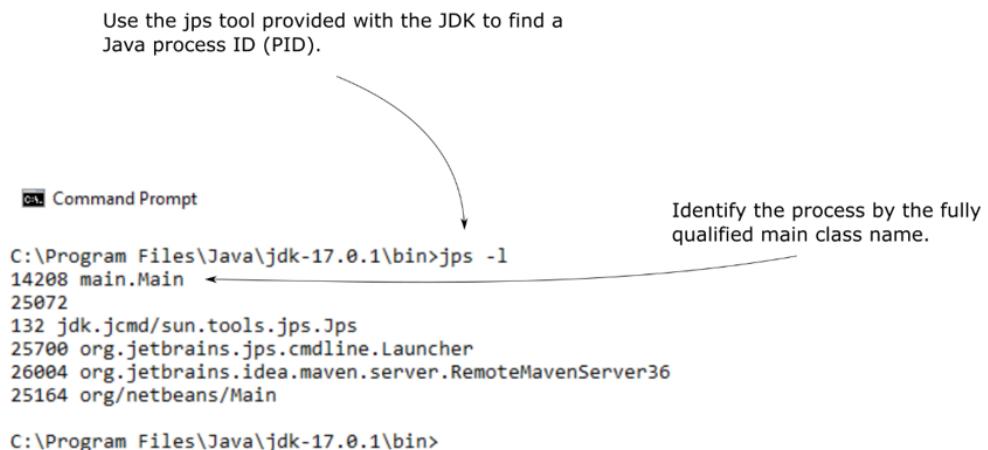


Figure 10.5 Using the `jps` tool provided with the JDK, we get the PIDs of the running Java processes. These PIDs are necessary further to get thread dumps for a given process.

STEP 2 – COLLECT THE THREAD DUMP

Once you can identify (by its PID) the process for which you want to collect a thread dump, you can use another tool the JDK provides named “`jstack`” to generate a thread dump. Using `jstack`, you just need to provide as a parameter the process ID as presented in the next snippet (in your case, instead of <<PID>> you need to use the PID value you collected in step 1).

```
jstack <<PID>>
```

An example of such a command execution would be:

```
jstack 14208
```

Figure 10.6 shows you the result of running the `jstack` command followed by a PID. The thread dump is provided as plain text that you can save in a file to move it or load into a tool for investigation.

Use the jstack tool provided with the JDK to get a thread dump. The only mandatory parameter is the process ID (PID) for the process you want to generate the thread dump.



```
C:\Program Files\Java\jdk-17.0.1\bin>jstack 14208
2021-12-01 17:10:51
Full thread dump OpenJDK 64-Bit Server VM (17.0.1+12-39 mixed mode, sharing):

Threads class SMM info:
    _java_thread_list=0x000002f96843b110, length=25, elements={...
0x000002f9646faaf0, 0x000002f9646fc110, 0x000002f96470ec20, 0x000002f9647107e0, ...
0x000002f964713a40, 0x000002f964718090, 0x000002f964719cb0, 0x000002f96471a750, ...
0x000002f96472e0b0, 0x000002f9647c0b0, 0x000002f964921580, 0x000002f96492dde0, ...
0x000002f964987690, 0x000002f96498b030, 0x000002f92eb5df0, 0x000002f965cb8ce0, ...
0x000002f966f07490, 0x000002f967087840, 0x000002f9666ba6a0, 0x000002f96747e060, ...
0x000002f96747f870, 0x000002f96747fd40, 0x000002f964a76010, 0x000002f964a746c0, ...
0x000002f967480210
}

"Reference Handler" #2 daemon prio=10 os_prio=2 cpu=0.00ms elapsed=692.87s tid=0x000002f9646faaf0 nid=0x6888 waiting on condition [0x00000fe5c1ff000]
    java.lang.Thread.State: RUNNABLE
        at java.lang.ref.Reference.waitForReferencePendingList(java.base@17.0.1/Native Method)
        at java.lang.ref.Reference.processPendingReferences(java.base@17.0.1/Reference.java:253)
        at java.lang.ref.Reference$ReferenceHandler.run(java.base@17.0.1/Reference.java:215)

"Finalizer" #3 daemon prio=8 os_prio=1 cpu=0.00ms elapsed=692.87s tid=0x000002f9646fc110 nid=0x5fa8 in Object.wait() [0x00000fe5c2ff000]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(java.base@17.0.1/Native Method)
        - waiting on <0x000000531818640> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(java.base@17.0.1/ReferenceQueue.java:155)
        - locked <0x000000531818640> (a java.lang.ref.ReferenceQueue$Lock)
```

Figure 10.6 The jstack command followed by a PID will generate a thread dump for the given process. The thread dump is shown as plain text (which we also say is the raw thread dump). You can collect the text in a file to import it and investigate it later.

STEP 3 – IMPORT THE COLLECTED THREAD DUMP INTO A PROFILER TO READ IT EASIER

Usually, you'll save the output of the jstack command which is the thread dump into a file. Storing the thread dump in a file gives you simple ways to move it, store it or import it in tools that help you investigate its details.

Figure 10.7 shows you how you can put the output of the jstack command in a file in the command line. Once you have the file, you can load it in VisualVM using the **File > Load** menu.

A generally good approach is putting the contents jstack outputs into a file so you can save it, send it, and investigate it easier.

```
C:\ Command Prompt
C:\Program Files\Java\jdk-17.0.1\bin>jstack 14208 > C:\MANNINGS\stack_trace.tdump
C:\Program Files\Java\jdk-17.0.1\bin>
```

You can open a saved thread dump in any profiler to easily read it. For example, in VisualVM, you can open it using File > Load.

VisualVM 2.1.1

File Applications View Tools Window Help

Load... ←

Save As... >

Load Recent

Delete Delete

main.Main (pid 14208) ×

View Monitor

main.Main (pid 1420)

Figure 10.7 Once you save the thread dump into a file, you can open it in various tools to investigate it. For example, to open it in VisualVM, you select File > Load.

10.2 Reading thread dumps

In this section, we discuss reading thread dumps. Once you collected a thread dump as we discussed in section 10.1, you need to know how to read it and how to efficiently use it to identify issues. We'll start discussing how to read plain text thread dumps in section 10.2.1 – meaning you'll learn to read raw data as provided by jstack (see section 10.1.2). Afterward, in section 10.2.2, we'll use a tool named **fastThread** (<https://fastthread.io/>) that will offer us a simpler way to visualize the data in a thread dump.

Both approaches (reading plain text thread dumps and using advanced visualization for thread dumps) are useful. Of course, we would always prefer to use advanced visualization since it's more comfortable, but if you can't, you need to know how to rely on raw data as well.

10.2.1 Reading plain text thread dumps

When you collect a thread dump, you get a description of the threads in plain text format (which we also say it's raw data). Although we have tools you can use to easily visualize the data (as we'll discuss in section 10.2.2), I always considered it important for a developer to understand the raw representation as well. You might get into a situation where you can't take out the raw thread dump from the environment where you generated it. Say you connect to a container remotely and you can only use the command line to dig into the logs

and investigate what happens with the running app. You suspect a thread-related problem so you want to generate a thread dump. If you can read the thread dump as text, you need nothing more than the console itself.

Let's take a look at listing 10.3 which presents the representation of one of the threads in the thread dump. The thread dump is nothing more than similarly displayed details for each thread that was active in the app when the dump was taken. So if you learn how to read the representation in listing 10.3, you'll understand a thread dump.

Here're the details you get for a thread:

- Thread name
- Thread ID
- Native thread ID
- Priority of the thread at the operating system level
- Total and CPU time the thread spent
- State description
- State name
- Stack trace
- Who's blocking the thread
- What locks the thread acquires

You observe that the first thing displayed is the **thread name**, in our case, “_Producer”. The thread name is essential since it's one of the ways you identify the thread in the thread dump later if you need it. The JVM also associates the thread with a **thread ID** (observe in listing 10.3, tid=0x000002f964987690). Since the name is given by the developer, there's a small chance that some threads will get the same name. If this unlucky situation happens, you can still identify a thread in the dump by its ID (that is always unique).

In a JVM app, a thread is in fact a wrapper over a system thread. That means you can always identify the operating system (OS) thread running behind the scenes. If you ever need to do that, in a thread dump, you find the **native thread ID** (in listing 10.3 nid=0xcac).

Once you identified a thread, you're most likely interested in details about it. Three of the first such details you get in a thread dump are the **thread's priority**, the **CPU execution time**, and the **total execution time**. Every OS associates a priority to each of its running threads. I didn't find myself often using this value in a thread dump. But we could say that if you spot that a thread's not active as much as you would consider it should be, and you observe the OS associates it a lower priority, then this might be the cause. Usually, in such a situation, you'd also observe that the total execution time is much higher than the CPU execution time. Remember, we discussed in chapter 7 that the total execution time is how long was the thread alive while the CPU execution time is how much it “worked”.

State description is a valuable detail. The state description tells you shortly in plain English what happens to the thread. In our case, the thread is “waiting for monitor entry”, meaning is blocked at the entrance to a synchronized block. It could have been that the thread is “timed waiting on a monitor” which meant it's sleeping for a defined time or the thread simply is “running”. A **state name** (RUNNING, WAITING, BLOCKED, and so on) is associated with the state description. Remember that appendix E offers a good refresher on thread lifecycle and thread states in case you need it.

The thread dump shows a **stack trace** for every thread which shows exactly what part of the code the thread was executing when the dump was taken. The stack trace is valuable since it shows you exactly what the thread was working on. You could use the stack trace to find a specific piece of code you want to debug further, or in the case of a slow thread, it shows you exactly what delays or blocks that thread.

Finally, in the case of threads that acquire locks or are locked, we can find in the thread dump **which locks they acquire** and **which locks are they waiting for**. You'll use these details every time you investigate a deadlock. Also, these details can give you optimization hints. For example, if you observe that a thread acquires many locks, you might wonder why and how can you change its behavior such that it doesn't need to block so many other executions.

Listing 10.3 The anatomy of a thread's details in a thread dump

```
"_Producer" #16 prio=5 os_prio=0 cpu=46.88ms elapsed=763.96s      #A
  tid=0x000002f964987690 nid=0xcac waiting for monitor entry      #B
    [0x000000fe5ceff000]
  java.lang.Thread.State: BLOCKED (on object monitor)      #C
    at main.Producer.run(Unknown Source)      #D
    - waiting to lock <0x000000052e0313f8> (a java.util.ArrayList)      #E
    - locked <0x000000052e049d38> (a java.util.ArrayList)      #F
```

#A Thread name and details about resource consumption and execution time

#B Thread ID and state description

#C Thread state

#D Thread stack trace

#E Lock ID that blocks the current thread and type of the monitor object

#F Lock ID of the lock produced by the current thread

An important thing to observe and remember about a thread dump is that it gives you almost as many details as a normal lock profiling does (discussed in chapter 9). Lock profiling offers an advantage over a thread dump: it shows the dynamics of the execution. Just like the difference between a picture and a movie, the thread dump is just a snapshot at a given time during the execution, while profiling shows you how parameters change during the execution. But in many situations, a "picture" is just enough and much easier to obtain.



Sometimes it could be enough to use
a thread dump instead of a profiler.

If the only thing you search is what code executes at a given time, a thread dump can show you this as well. Up to now you learned to use sampling for this purpose, but it's good

to know a thread dump can do this too. Say you don't have access to remotely profile an app but you need to find out what code executes behind the scenes. You can alternatively get a thread dump.

Let's focus now on how can you find the relationship between threads with a thread dump. How can we analyze the way in which threads interact with one another? We are particularly interested in threads locking each other. In listing 10.4 I added the details from the thread dump for the two threads that we know are in a deadlock. But the question is, how would we find they are in a deadlock if we didn't know this detail upfront?

When suspecting a deadlock, you will focus your investigation on the locks the threads cause. The steps are the following:

1. You filter out all threads that are not blocked so you can focus on the threads that could cause the deadlock.
2. You start with the first candidate thread (thread you didn't filter in step 1), and search the lock ID that causes it to be blocked.
3. You find the thread causing that lock and check who blocks that thread repeating the same process. If at some point you return to the thread you started with, it means that all the threads you parsed are in a deadlock.

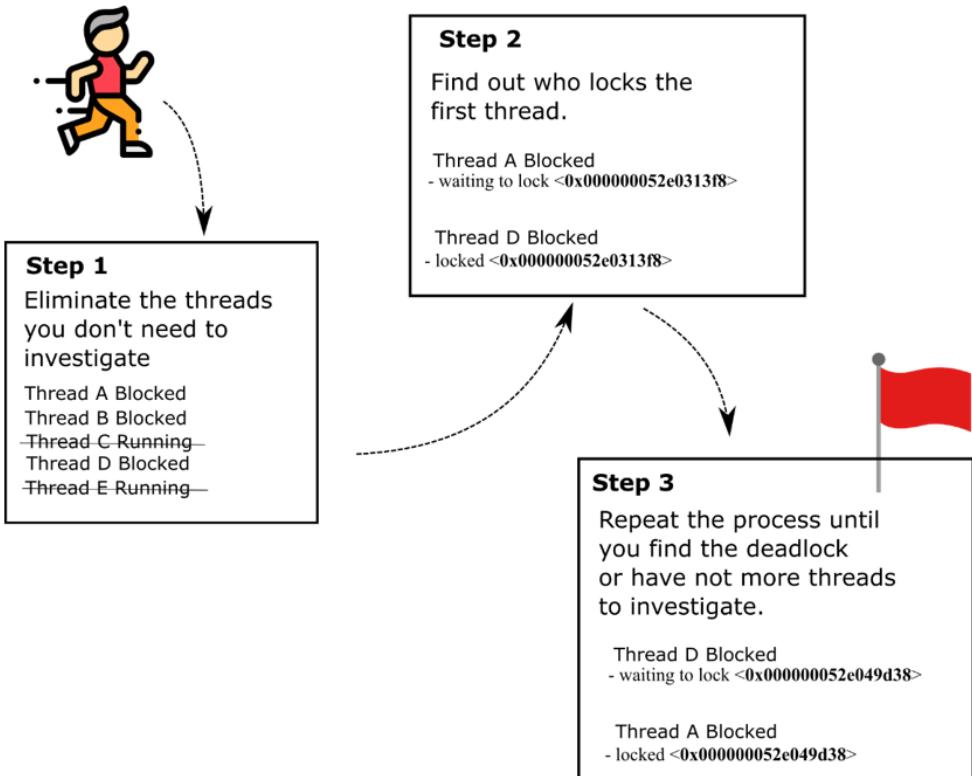


Figure 10.8 To find a deadlock with a thread dump you follow three easy steps. First, you remove all threads that are not blocked. Then, you start with one of the blocked threads and find out who is blocking it using the lock ID. For each thread, you continue the process. If through this process you return to a thread you already investigated, it means you found a deadlock.

STEP 1 – FILTER OUT THREADS THAT ARE NOT LOCKED

First, you filter out all the threads that are not locked so that you can focus only on the threads that are potential candidates for the situation you investigate – the deadlock. A thread dump can describe dozens of threads. You want to eliminate the noise and focus only on the threads that are blocked by someone.

STEP 2 – TAKE THE FIRST CANDIDATE THREAD AND FIND WHO BLOCKS IT

After eliminating the unnecessary thread details, start with the first candidate thread and search by the lock ID that causes a thread to wait. The lock ID is the one between angle brackets (in listing 10.4, “_Producer” thread waits for a lock with ID 0x000000052e0313f8).

STEP 3 – FIND OUT WHO BLOCKS THE NEXT THREAD

You repeat the process finding out for each thread who locks it. If at some point you get to a thread that was already investigated in your process, it means you found a deadlock.

Listing 10.4 Finding threads that lock each other

```
"_Producer" #16 prio=5 os_prio=0 cpu=46.88ms elapsed=763.96s tid=0x000002f964987690
    nid=0xcac waiting for monitor entry [0x000000fe5ceff000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at main.Producer.run(Unknown Source)
        - waiting to lock <0x000000052e0313f8> (a java.util.ArrayList)      #A
        - locked <0x000000052e049d38> (a java.util.ArrayList)      #B

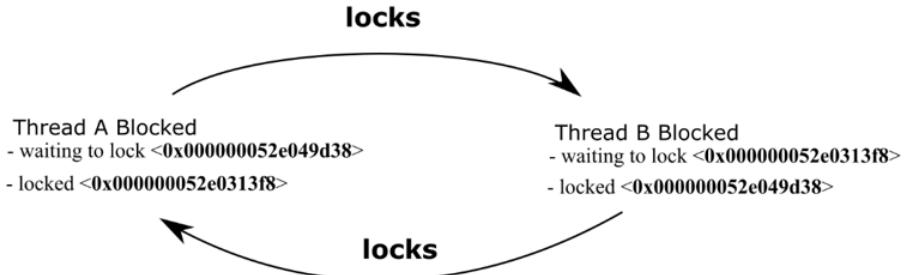
"_Consumer" #18 prio=5 os_prio=0 cpu=0.00ms elapsed=763.96s tid=0x000002f96498b030
    nid=0x4254 waiting for monitor entry [0x000000fe5cff000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at main.Consumer.run(Unknown Source)
        - waiting to lock <0x000000052e049d38> (a java.util.ArrayList)      #B
        - locked <0x000000052e0313f8> (a java.util.ArrayList)      #A
```

#A The _Producer thread waits for a lock initiated by the _Consumer thread

#B The _Consumer thread waits for a lock initiated by the _Producer thread

Our example demonstrates a simple deadlock. A simple deadlock assumes two threads lock each other. Following the three-step process discussed earlier, you'll find out that the "_Producer" thread blocks that "_Consumer" thread and the other way around. A complex deadlock happens when more than two threads are involved. For example, thread A blocks B, thread B blocks C and thread C blocks thread A. You can discover a long chain of threads that lock each other. The longer the chain of threads in the deadlock, the more difficult the deadlock is to find, understand and solve. Figure 10.9 shows visually the difference between a complex deadlock and a simple one.

Simple deadlock



Complex deadlock (more than 2 threads)

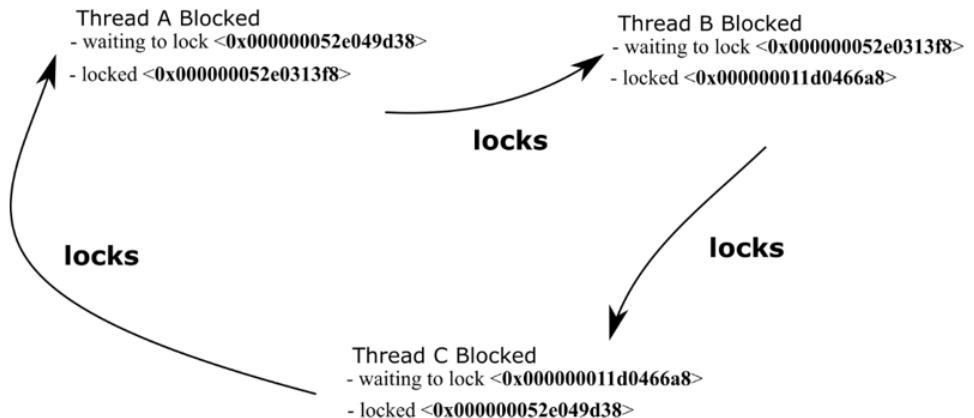


Figure 10.9 When only two threads are blocking each other, we say we have a simple deadlock. A deadlock can however be caused by multiple threads who block each other. With more threads, understanding the deadlock becomes more difficult. When more than two threads are involved we say we have a complex deadlock.

Sometimes, a complex deadlock can be confused with cascading blocked threads (figure 10.10). Cascading blocked threads (also known as cascading locks) are, however, a different issue you can spot using a thread dump. To find cascading threads, you follow the same steps as for investigating a deadlock. But instead of finding out that one of the threads is blocked by another in the chain (as in the case of a deadlock), in a cascade of locks you'll just find that one of the threads is waiting for an external event causing all others to wait for it.

Cascading blocked threads usually signal a bad design in the multithreaded architecture. When we design an app having multiple threads, we implement threading for allowing the app to process things concurrently. Having threads waiting for one another defeats the

purpose of a multithreaded architecture. While sometimes you need to make threads wait for one another, you shouldn't expect long chains of threads cascading locks.

Thread A Blocked

- waiting to lock <0x000000052e049d38>

blocked by

Thread B Blocked

- waiting to lock <0x000000052e0313f8>

- locked <0x000000052e049d38>

blocked by

Thread C Blocked

- waiting to lock <0x000000011aa45bdb2>

- locked <0x000000052e0313f8>

blocked by



Figure 10.10 Cascading locks appear when multiple threads enter a chain where they wait for one another. The last thread in the chain is blocked by an external event such as reading from a data source or calling an endpoint.

10.2.2 Using tools to easier grasp thread dumps

Reading the plain text raw representation of a thread dump is useful but sometimes can be quite difficult. You'd definitely prefer to have a more simple way to visualize the data in a thread dump if possible. Today we can use tools to help us easily understand a thread dump. Whenever I can get the thread dump out of the environment where I collect it, I usually prefer using **fastThread** (fastthread.io) to investigate the dump instead of dealing with the raw data myself.

fastThread is a web tool designed to help you easily read thread dumps. The tool offers both free and paid plans, but, in my case, the free plan was always enough with what I needed to investigate. All you need to do is upload a file containing the thread dump raw data and wait for the tool to extract the details you need and put them in a shape easier to grasp. Figure 10.11 shows you the starting page where you choose the file containing the thread dump raw data from your system and upload it for analysis.

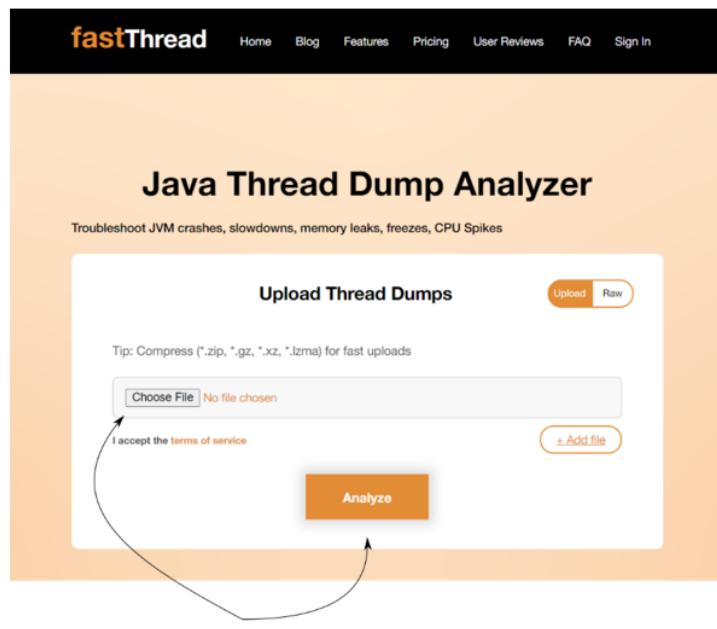
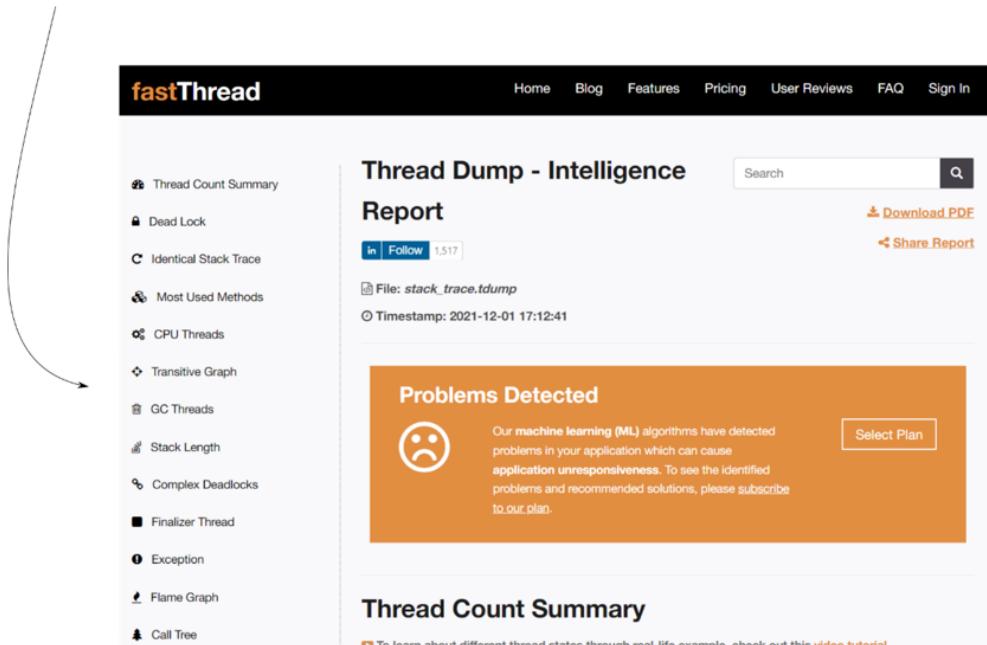


Figure 10.11 To analyze a thread dump, you upload a file containing the thread dump raw data to fastThread.io and wait for the tool to present the details in a simple-to-understand shape.

The fastThread analysis shows various details you can get from the thread dump. These details include deadlock detection, dependency graphs, stack traces, resource consumption, and even a flame graph representation of the execution (figure 10.12).

After analyzing the thread dump, the tool presents multiple visualization widgets such as: identifying deadlocks, CPU consumption per thread and even a flame graph representation of the process.



The screenshot shows the fastThread website interface. On the left, there's a sidebar with various analysis options: Thread Count Summary, Dead Lock, Identical Stack Trace, Most Used Methods, CPU Threads, Transitive Graph, GC Threads, Stack Length, Complex Deadlocks, Finalizer Thread, Exception, Flame Graph, and Call Tree. A curved arrow points from the text above to this sidebar. The main content area is titled "Thread Dump - Intelligence Report". It shows a file named "stack_trace.tdump" uploaded on 2021-12-01 17:12:41. Below this, a large orange box highlights "Problems Detected" with a sad face icon and a message about machine learning detecting application unresponsiveness. It includes a "Select Plan" button. At the bottom, there's a "Thread Count Summary" section with a link to a video tutorial.

Figure 10.12 With fastThread you can easily analyze the thread dumps. After uploading the raw data to the tool, you get various details in an easy-to-read shape. These details include deadlock detection, dependency graphs, resource consuming and even a flame graph representation of the execution.

Figure 10.13 shows how fastThread identified the deadlock in our thread dump.

The tool identifies the deadlock and the threads causing it.

Dead Lock

Learn more about [Deadlock](#)

Thread **_Producer** is in deadlock with thread **_Consumer**

_Producer	
PRIORITY : 5	NATIVE ID (DECIMAL) : 3244
THREAD ID : 0X0000002F964987690	STATE : BLOCKED
NATIVE ID : 0XCAC	
stackTrace:	
java.lang.Thread.State: BLOCKED (on object monitor)	
at main.Producer.run(Unknown Source)	
- waiting to lock <0x000000052e0313f8> (a java.util.ArrayList)	
- locked <0x000000052e049d38> (a java.util.ArrayList)	
_Consumer	
PRIORITY : 5	NATIVE ID (DECIMAL) : 16980
THREAD ID : 0X0000002F96498B030	STATE : BLOCKED
NATIVE ID : 0X4254	
stackTrace:	
java.lang.Thread.State: BLOCKED (on object monitor)	

Figure 10.13 After analyzing the thread dump raw data, fastThread identified and provides details about the deadlock caused by the **_Consumer** and **_Producer** threads.

10.3 Summary

- When two or more threads get blocked waiting for each other, we say they are in a deadlock. When an app gets into a deadlock, it usually freezes and can't continue its execution.
- You can identify the root cause of a deadlock using thread dumps. A thread dump shows the status of all threads of an app at the time the thread dump was generated. Using the information a thread dump provides, you can easily find which thread is waiting for another.
- A thread dump also shows details such as resource consumption and stack traces for each thread. Sometimes you can use a thread dump instead of instrumentation if these details are enough for your investigation. You can imagine the difference between a thread dump and profiling as the difference between a picture and a movie. With a thread dump, you only have a picture, so you'll miss the execution dynamics, but you can still get a lot of relevant and helpful details.
- The thread dump describes details about the threads that were executing in the app when the dump was taken. The thread dump shows essential details about the threads in a plain text format. Among these details, we find the resource consumption, thread state in its lifecycle, if the thread is waiting for something, and which locks it's causing or it's affected by.
- You can generate a thread dump either by using a profiler or a command-line tool. Using a profiling tool to get the thread dump is the easiest approach, but when you can't connect a profiler to the running process (for example, due to network constraints) you can use a command-line tool to get the dump. Getting the thread dump will allow you to investigate the running threads and the relationships among them.
- The plain text thread dump (also known as raw thread dump) can be challenging to read. Tools such as fastThread.io help you easily visualize the details in a thread dump.

11

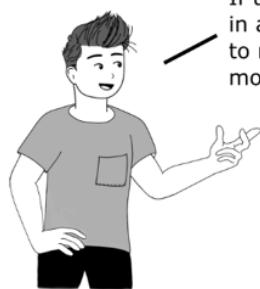
Finding memory-related issues in an app's execution

This chapter covers

- Sampling an execution to find memory allocation issues
- Profiling a part of the code to identify the root causes of memory allocation problems
- Obtaining and reading heap dumps

Every app processes data, and to process the data, the app needs to store that data somewhere while working with it. The app allocates a part of the system's memory to work with the data, but the memory isn't an infinite resource. All the apps running on a system share a finite amount of memory space that the system provides. If an app doesn't wisely manage its allocated memory, it might run out of memory, making it impossible to continue its work. Even if the app doesn't run out of memory, using too much of it might make the app slower – so faulty memory allocation can be a cause of performance issues.

An app can run slower just because it doesn't allocate data in the memory in an optimized way. If the app allocates more memory than the system provides, the app will stop working and throw an error. It's essential we write app capabilities that use their allocated memory in an optimized way.



If the app doesn't allocate the data it processes in an optimized way, it may force garbage collector to run more often, and so the app will become more CPU intensive.

One of the purposes of an app is to be as efficient as possible in managing its resources. When we discuss the resources of an app, we mainly think about CPU (processing power) and memory. In chapters 7 through 10, we discussed how to investigate issues with CPU consumption. In this chapter, we'll focus on identifying problems with how an app allocates data in memory.

When an app has memory management issues, these issues can have side effects, such as the slowness of the execution and even the app crashing entirely.

We'll start the chapter by discussing execution sampling and profiling for memory usage statistics in section 11.1. You'll learn how to identify if an app potentially has issues with memory usage and how to find which part of the application causes them.

Then, in section 11.2, we'll discuss how to get a complete dump (that we'll call heap dump) of the allocated memory to analyze its contents. In certain cases where the app crashes entirely because of faulty memory management and the app stops, you cannot profile the execution since the app doesn't execute anymore. But getting and analyzing the contents of the app's allocated memory at the moment the problem appeared can help you identify the problem's root cause.

You need to remember a few basic concepts about the way in which a Java app allocates and uses memory before continuing with this chapter. In case you need a refresher, appendix E gives you all the essential details that you'll need to know to properly understand this chapter's discussion.

11.1 Sampling and profiling for memory issues

In this section, we'll use a small application that simulates a faulty implemented capability that uses too much of the allocated memory. We'll use this app to discuss investigation techniques you can use to identify issues with memory allocation or places in code that can be optimized to use the system's memory more efficiently.

Suppose you have a real application and you observe some feature runs slowly. You use the techniques we discussed in chapter 6 to analyze the resource consumption and you observe that the app doesn't necessarily "work" a lot (consume CPU resources), but it uses an extensive amount of memory. When an app uses too much memory to achieve something, the JVM can trigger the garbage collector (GC) often and the GC will further

consume CPU resources also. Remember that the GC is the mechanism that automatically deallocates unneeded data from memory (see appendix E for a refresher).

Take a look at figure 11.1. When discussing how to analyze resource consumption in chapter 6, we used the **Monitor** tab in VisualVM to observe what resources the app consumes. You can use the memory widget in this tab to find out when the app uses an extensive amount of memory.

Under the Monitor tab you find the widget that allows you monitor the app's memory usage.

For a call of the app's endpoint you observe a large spike increase in the used memory. The JVM also adjusted the heap max size as a result of the increased memory usage.

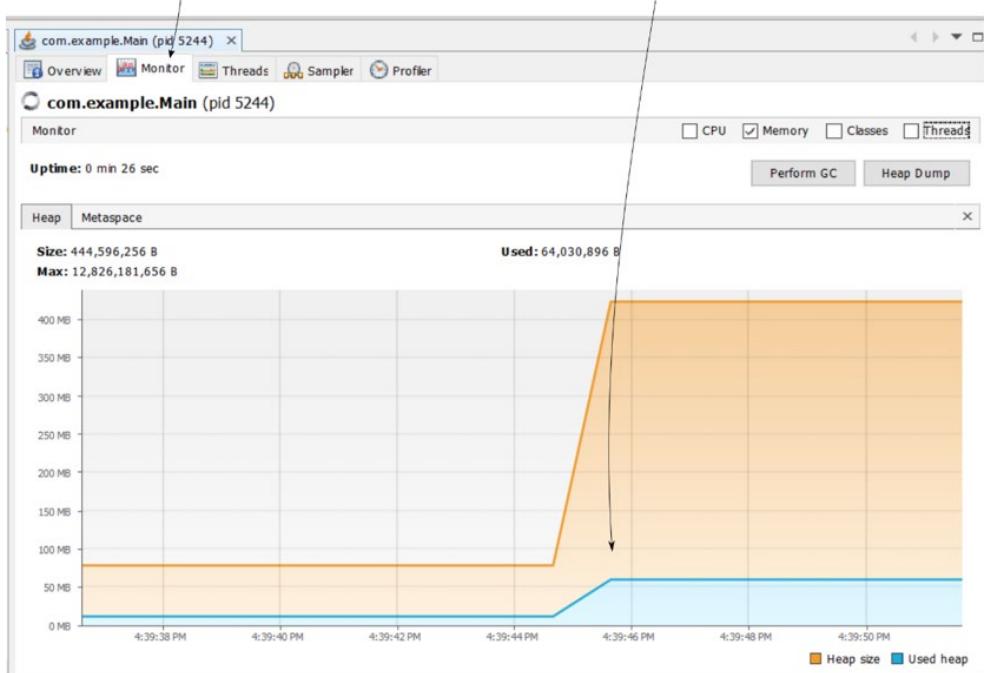
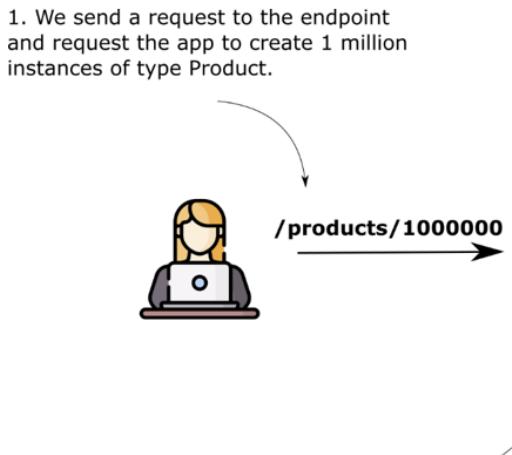


Figure 11.1 The memory widget in the Monitor tab in VisualVM helps you identify if the app spends more memory than usual at a given time. Often, widgets in the Monitor tab, such as the CPU consumption and the memory consumption, give us clues on how to continue our investigation. In such a case where we observe the app consumes an abnormal amount of memory, we may decide to continue with memory profiling the execution.

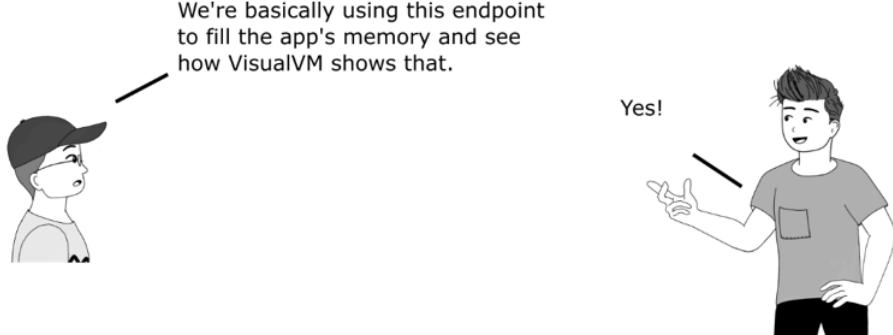
You find the application that we use in this chapter provided with the book in project data-ch11-ex1. This small web application exposes an endpoint. When calling this endpoint, you

give a number, and the endpoint creates that many object instances. We'll basically request to create one million objects (which is a large enough number for our experiment) and we'll find out what a profiler tells us for this request execution. This endpoint execution that we'll investigate simulates what would happen in a real-world situation where a given app capability spends a lot of the app's memory resources.



3. We'll analyze what happens in Visual VM.

Figure 11.2 When we call the endpoint exposed by the provided project `da-ch11-ex1`, the app creates a large number of instances that consume a large part of the app's memory. We'll analyze this scenario using a profiler.



You need to start the project, open VisualVM, and select the process we investigate in VisualVM. Open the **Monitor** tab, then, call the `/products/1000000` endpoints using either curl or Postman. The steps are to

1. Start project da-ch11-ex1
2. Start VisualVM
3. Select a process for project da-ch11-ex1 in VisualVM
4. Go to the **Monitor** tab in VisualVM
5. Call the /products/1000000 endpoint
6. Observe the memory widget in the **Memory** tab in VisualVM

In the **Monitor** tab, in the memory widget, you'll observe the app uses plenty of memory resources. The widget looks similar to what you see in figure 11.1. What should we do in such a case in which we suspect some app capability doesn't optimally use the memory resources? The investigation process further follows two major steps:

1. Memory **sampling** to get details about the object instances the app stores.
2. Memory **profiling (instrumentation)** to get additional details about a specific part of the code in execution.

Following the same approach you learned in chapters 7 through 9 for CPU resources consumption, the best way is first to get a top-of-the-mountain view of what happens using sampling. To sample an app execution for memory usage, you select the **Sampler** tab in VisualVM. Then, you select the **Memory** button to start a memory usage sampling session. Call the endpoint that we use for our demonstration and wait for the execution to end. You'll observe that the VisualVM screen will display the objects the app allocates.

What we are looking for is what occupies most of the memory. In most cases, you'll find one of these two situations:

1. Many object instances of certain types are created and they fill up the memory (this actually happens in our scenario).
2. There're not so many instances of a certain type, but each instance is very large.

Many instances filling up the allocated memory makes sense, but how could a small number of instances do this? Imagine this scenario: Your app processes large video files. The app maybe loads two or three files at a time, but since they are large, they will fill up the allocated memory. A developer could analyze if the capability can be optimized. Maybe the app doesn't need the full files loaded in memory but only fragments of them at a time.

When starting our investigation, we don't know which is the scenario in which we'll fall. So I usually sort descending first by the amount of memory occupied and then by the number of instances. Observe, as presented in figure 11.3, that for each sampled type, VisualVM shows you the memory spent and the number of instances. You just need to sort descending by each of the second and the third columns in the table.

In figure 11.3 you observe how I sorted the table descending by Live Bytes (space occupied). What we then look for is the first type belonging to our app's codebase that appears in the table. Don't look for primitives, strings, arrays of primitives, or arrays of strings. These will usually be at the top since they are created as a side effect. But in most cases, they won't give you any clue about what the problem really is.

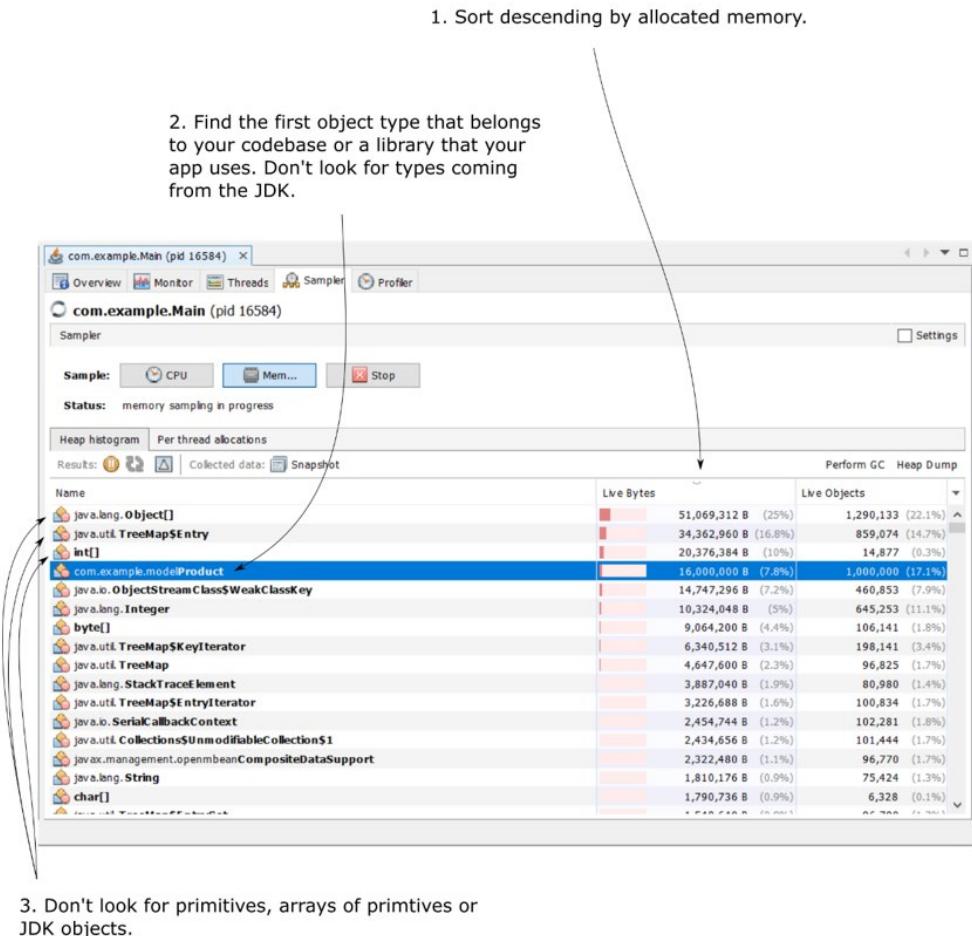


Figure 11.3 We sort the sampled results descending by memory occupied. This way, we observe which objects spend most of the memory. We don't usually look for primitives, strings, and arrays of strings (or in general objects types from the JDK). We are mostly interested in seeing which object directly related to our codebase causes the problem. In this case, we observe the `Product` type (which is part of our codebase) occupies a large part of the memory.

In figure 11.3 we can clearly observe that type `Product` is causing trouble. It occupies a big part of the allocated memory, and in the **Live Objects** column, we observe the app created 1 million instances of this type.



The profiling tool names them live objects because sampling only shows you the instances that still exist in the memory.

In case you need the total number of instances of that type that have been created throughout execution, you need to use profiling (instrumentation) techniques. We'll do this later in this chapter.

This app is just an example, made on purpose for our discussion, but in a real-world app, just sorting by the occupied space might not be enough. We need to figure out if the problem is a large number of instances or simply each instance takes a lot of space. I know what you think: Isn't it clear in this case? Yes, it is, but in a real-world app it might not be, so I always recommend that developers also sort descending by the number of instances to make sure. Figure 11.4 shows the sampled data sorted descending by the number of instances the app created for each type. Again, type `Product` takes a place at the top.

1. Sort descending by the number of object instances (live objects).

2. Find the first object type that belongs to your codebase or a library that your app uses. Don't look for types coming from the JDK.

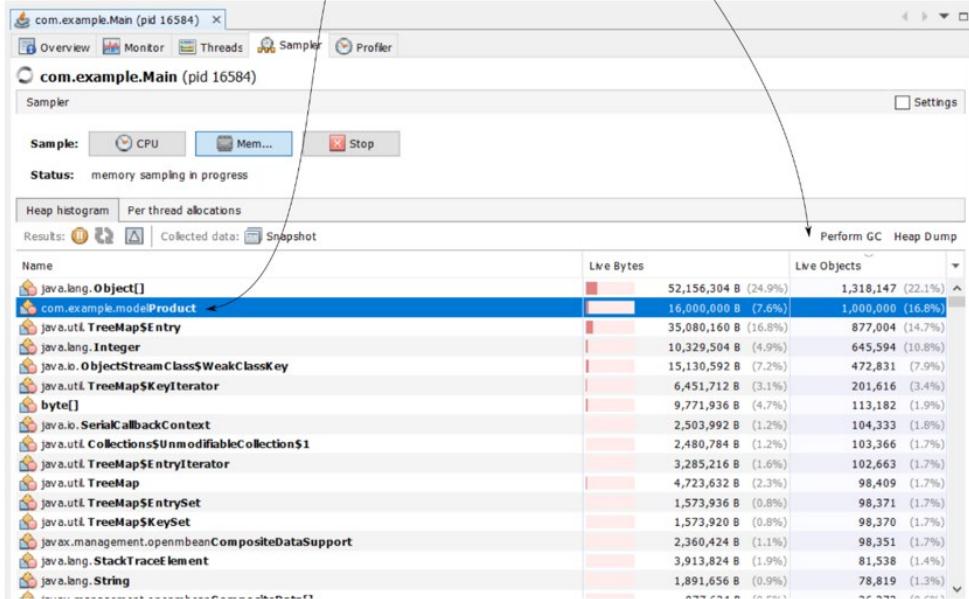


Figure 11.4 We also sort the sampled results descending by the number of instances (live objects). Sorting the results like this will give us clues on whether some capability creates a large number of objects that could negatively impact the memory allocation.

Sometimes sampling is enough to help you realize what the problem is. I saw in many cases developers directly figuring out the root cause only after sampling since they know very well what the app does. In some cases, they were directly aware of where the app creates the potentially problematic objects or they knew it was some new capability the team delivered.

But what if you can't figure out straightforwardly what part of the app creates those objects? In many cases, I've found myself investigating apps and not being able to figure out where an object is created just with sampling. As a consultant, I see many applications that are new for my eyes - some of them are big, some of them are old, and most of them messy.

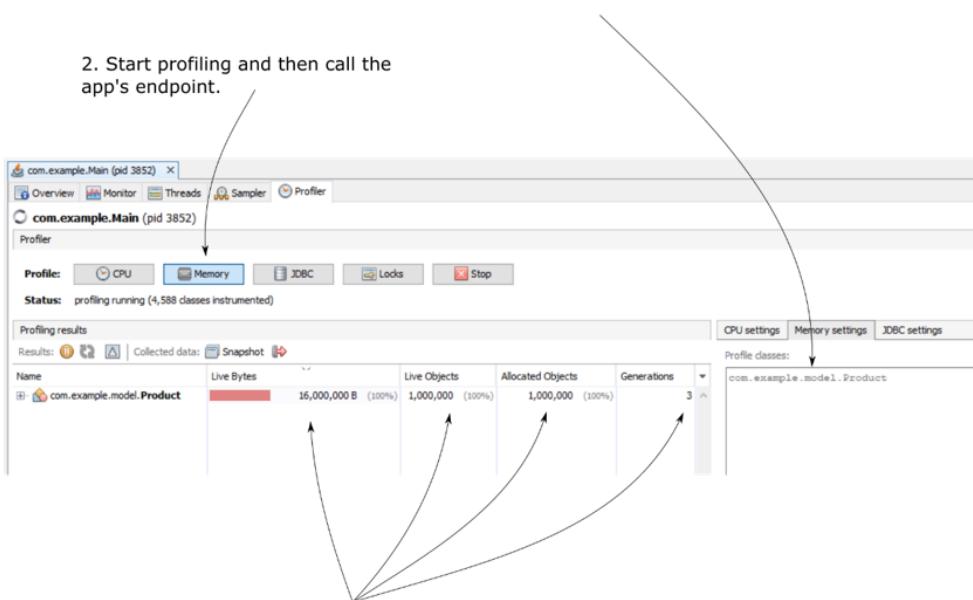
When you can't figure out what the problem is just by sampling the execution, your next step is profiling (instrumentation). Using profiling, you get more details, including what part

of the code created the potentially problematic instances. But remember the rule of thumb: When you use profiling, you need first to know what to profile. That's the reason why we always use sampling first.

Since we know the problem is with the `Product` type, we're going to profile for this type. Similar to how we did in chapters 7 through 9, you have to specify which part of the app you want to profile using an expression. In figure 11.5 you observe that I profile only for the `Product` type. I do this by using the fully qualified name (package and class name) of the class in the **Memory settings** textbox on the right side of the window.

1. Specify the expression that defines which objects you want to profile for memory usage.

2. Start profiling and then call the app's endpoint.



3. The profiler will indicate details about each object involved in execution during the profiling session. You'll find the allocated memory per object, the number of instances in memory for each object, how many objects have been garbage collected, how many still exist in the memory, and how many times the GC tried to remove them from the memory.

Figure 11.5 To profile for memory allocation, first specify which packages or classes you want to profile, and then start the profiling by selecting the Memory button. The profiler will give you relevant details about the profiles types including used memory, number of instances, the total number of allocated objects, and the number of GC generations.

Just as in the case of CPU profiling (chapter 8), you can profile for more types at a time or even specify entire packages. Some of the commonly used expressions are

- Strict type fully qualified name (for example, `com.example.model.Product`) – only searches for that specific type.
- Types in a given package (for example, `com.example.model.*`) – only searches for types declared in the package `com.example.model` but not in its sub-packages.
- Types in a given package and its sub-packages (for example, `com.example.**`) – searches in the given package and all its sub-packages.



To make your investigation easier, always remember to restrict as much as possible the types you profile. If you know Product causes the problem, then it makes sense only to profile this type.

Besides the live objects (which are the instances still existing in memory for that type) you will also get the total number of instances of that type that the app created. Moreover, you will also find how many times those instances “survived” the garbage collector (that’s what we call the “generations”).

These details are useful, but finding what part of the code creates the objects is often more useful than these values for me. In figure 11.6, you observe that for each profiled type, the tool also shows you where the instances were created. Just click on the **(+)** symbol on the left side of the line in the table. This capability that the profiler has quickly shows you where the root cause of the problem most likely is.

For each profiled object type, the profiler indicates the part of code that created it during execution. This way, you find the place with a potential problem or that could be optimized.

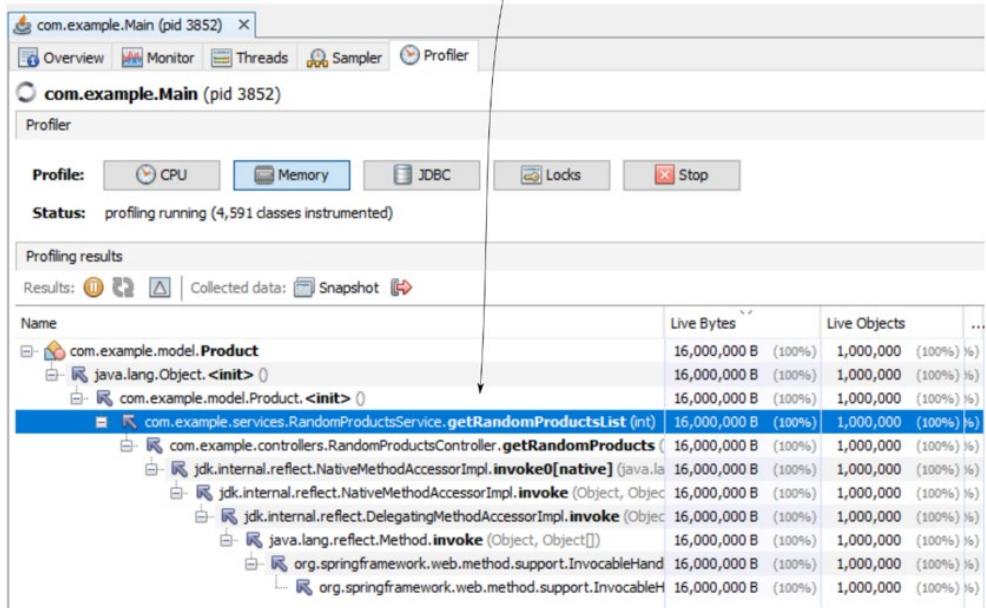


Figure 11.6 The profiler shows the stack trace of the code that created the instances of each of the profiled types. This way, you can easily identify what part of the app created the problematic instances.

11.2 Using heap dumps to find memory leaks

If the app's running, then you can profile to understand if any capability can be optimized. But what if the app crashed and you suspect this happened due to a memory allocation issue? In most cases, app crashes are caused by capabilities with memory allocation problems such as memory leaks. A memory leak is a situation where the app doesn't deallocate the objects it creates in memory even if it doesn't need these objects anymore. Since the memory is not infinite, continuously allocating objects will make the memory fill at some point causing the app to crash. In a JVM app, this will be signaled with an `OutOfMemoryError` thrown at runtime.

If the app's not running, you can't attach a profiler to investigate the execution anymore. But even so, you have alternatives to investigate the problem. You can collect a heap dump which is a snapshot of what the heap memory looked like when the app crashed. Well, you can collect a heap dump anytime, you don't need to have an app crashing for that, but you'll most likely find the heap dump useful when you can't profile the app for any reason - maybe

because the app crashed and it's not running anymore or you simply don't have access to profile the process and you want to investigate if it suffers from any memory allocation issues.

In section 11.2.1, we'll discuss three possible ways to get a heap dump, and in section 11.2.2, I'll show you how to use the heap dump to identify memory allocation issues and their root causes. In section 11.2.3, we'll discuss a more advanced way of reading a heap dump using a query language named Object Query Language (OQL). This OQL is similar to SQL, but instead of using it to query a database, you use OQL to query the data in a heap dump.

11.2.1 Obtaining a heap dump

In this section, we discuss three ways in which you can get a heap dump. Obviously, to use a heap dump for your investigation you need to get one first. There're three main ways in which you get a heap dump:

1. Configure the application to generate one automatically in a given location when the app crashes because of a memory issue.
2. Use a profiling tool (such as VisualVM).
3. Use a command-line tool (such as jcmd or jmap).

You could even get a heap dump programmatically. Some frameworks have capabilities you can straightforwardly use for getting a heap dump. And that allows developers to integrate their apps with app-monitoring tools. To learn more about this subject, you can check the `HotSpotDiagnosticMXBean` class in the Java official API documentation.

Link to the `HotSpotDiagnosticMXBean` documentation:

<https://docs.oracle.com/en/java/javase/17/docs/api/jdk.management/com/sun/management/HotSpotDiagnosticMXBean.html>

Project da-ch11-ex1 implements an endpoint you can use to generate the heap dump using the `HotSpotDiagnosticMXBean` class. Calling this endpoint using cURL or Postman (see next snippet) will generate a dump file.

```
curl http://localhost:8080/jmx/heapDump?file=dump.hprof
```

CONFIGURING AN APP TO GENERATE A HEAP DUMP WHEN IT ENCOUNTERS A MEMORY ISSUE

Most often, developers need a heap dump to investigate an app crash where they suspect the problem was caused by faulty memory allocation. For this reason, apps are most often configured to generate a heap dump of what the memory looked like when the app crashed. You should always configure an app to generate a heap dump when it stops because of a memory allocation problem.

Fortunately, configuring an app to generate a heap dump when the app stops due to a memory problem is easy. You just need to add a couple of JVM arguments when the app starts. The next snippet shows you the two arguments you need to add:

```
-XX:+HeapDumpOnOutOfMemoryError      #A
-XX:HeapDumpPath=heapdump.bin       #B
```

#A Parameter to tell the JVM to generate a heap dump if it encounters an OutOfMemoryError

#B Parameter to tell the JVM where to store the generated heap dump.

The first argument, `-XX:+HeapDumpOnOutOfMemoryError`, tells the app to generate a heap dump when it encounters an `OutOfMemoryError` (the heap gets full). The second argument, `XX:HeapDumpPath=heapdump.bin`, specifies the path in the file system where the dump will be stored. In this case, the file containing the heap dump will be named `heapdump.bin` and will be right near the app executable, from the root of the classpath, (because we used a relative path). Make sure the process has “write” privileges on this path to be able to store the file in the given location.

The full command for running an app would look like in the following snippet:

```
java -jar -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heapdump.bin app.jar
```

We'll use a demo app named `da-ch11-ex2` to demonstrate this approach. You can find this app in the projects provided with the book. The app continuously adds instances of a type `Product` to a list until the memory gets full. The next code snippet shows its very simple implementation.

```
public class Main {

    private static List<Product> products = new ArrayList<>();

    public static void main(String[] args) {
        Random r = new Random();
        while (true) {          #A
            Product p = new Product();
            p.setName("Product " + r.nextInt());
            products.add(p);    #B
        }
    }
}
```

#A The loop iterates forever.

#B Adding instances to the list until the memory gets full.

The next code snippet shows what the simple `Product` type looks like:

```
public class Product {

    private String name;

    // Omitted getters and setters
}
```

Maybe you're wondering why the random name for the product instances. We'll need it later when we discuss reading a heap dump in section 11.2.2. So wait a bit and you'll figure out why I designed the example like this. For the moment, the only thing we're interested in is that this app fills its heap memory in seconds and we'll need to get a heap dump to investigate why.

You can use the IDE directly to run the app and set the arguments. Figure 11.7 shows how to set the JVM arguments in IntelliJ. I also added the `-Xmx` argument to limit the heap memory of the app to just 100 Mb. That will make the heap dump file smaller and our example easier.

Set the JVM arguments in the Run/Debug configuration window.

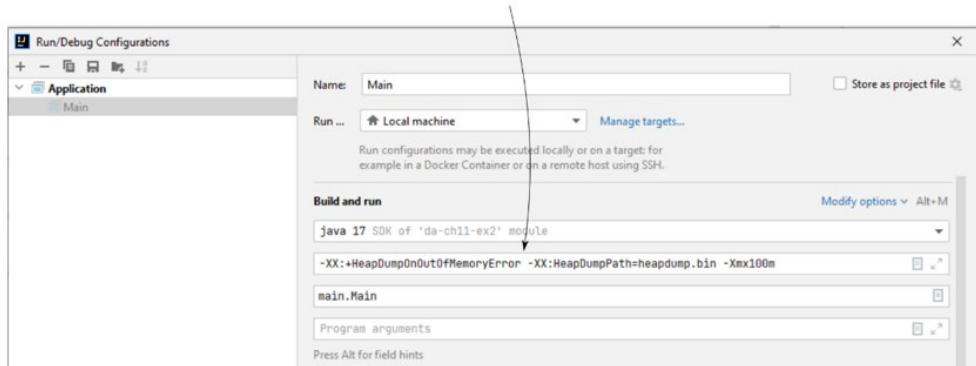


Figure 11.7 You can configure the JVM arguments from your IDE. Add the values in the Run/Debug configurations before starting the application.

When you run the application, wait a few seconds and the app will crash. With only 100Mb of heap space, the memory shouldn't take more than a few seconds to get full. In the project folder, you will find the file named `heapdump.bin`, which contains all the details about the data in the heap at the moment the app stopped. You can open this file with VisualVM now to analyze it as presented in figure 11.8.

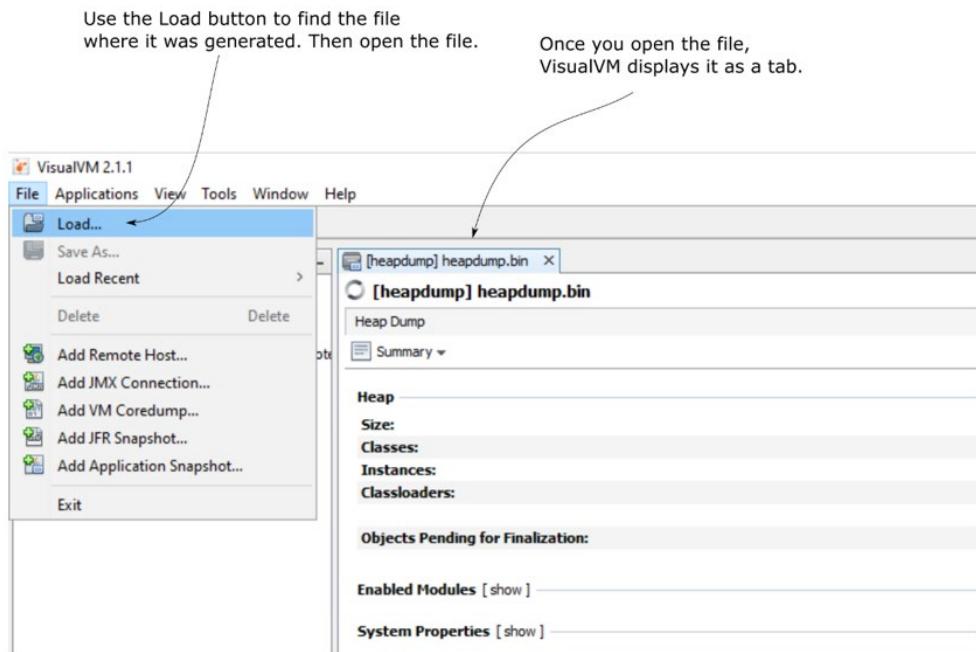


Figure 11.8 You can use VisualVM to open the heap dump file for analysis. Use the Load button in the menu to find the file where it was generated by the app. Then, open the file. VisualVM will display the heap dump as a tab.

OBTAINING A HEAP DUMP USING A PROFILER

Sometimes you need to get a heap dump for a running process. In this case, the easiest solution is to use VisualVM (or a similar profiling tool) to get the dump. Getting a heap dump with VisualVM is as easy as clicking a button. Just use the **Heap Dump** button in the **Monitor** tab for any process as shown in figure 11.9.

Select the Heap Dump button in the Monitor tab to get a heap dump. VisualVM opens the dump as a tab and you'll be able to investigate it or save it anywhere you want.

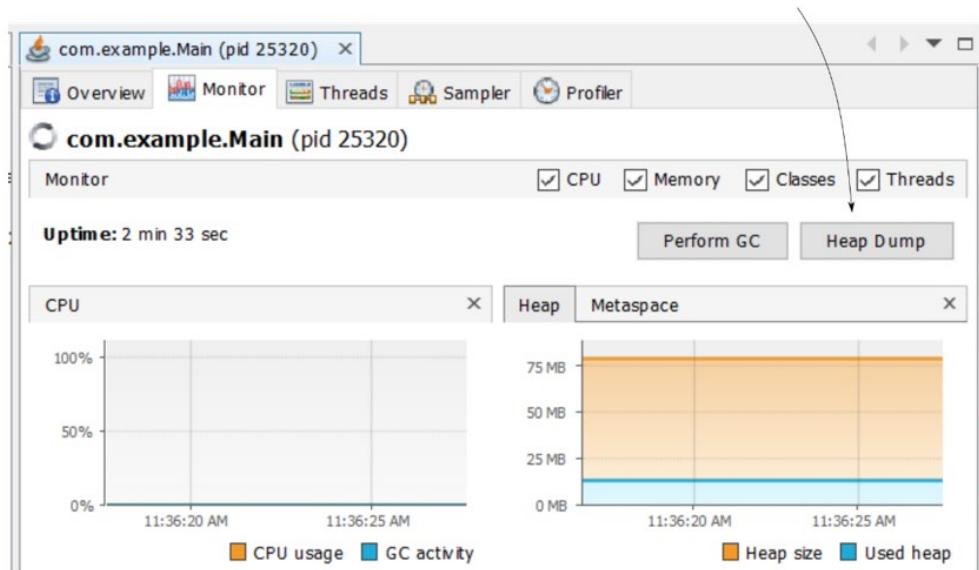


Figure 11.9 Select the Heap Dump button in VisualVM's Monitor tab to get a heap dump for the selected process. VisualVM opens the dump as a tab and you can further investigate it or save it anywhere you want.

OBTAINING A HEAP DUMP WITH THE COMMAND LINE

If you need to get a heap dump for a running process, but you cannot connect the profiler to it, don't panic, you still have options. You can use a command-line tool to generate the heap dump. So if your app is deployed in an environment where you don't have access to connect a profiling tool, you can directly connect via command line and get the dump using `jmap` – a tool provided with the JDK.

The two steps for collecting a heap dump with `jmap` are

1. Find the process ID (PID) of the running app for which you want to get the heap dump.
2. Use `jmap` to save the dump into a file.

To find the running process PID, you can use `jps` as we did in chapter 10. The next snippet reminds you of the command.

```
jps -l
25320 main.Main      #A
132 jdk.jcmd/sun.tools.jps.Jps
25700 org.jetbrains.jps.cmdline.Launcher
```

#A The first column in the output provides the PID. The second column is the Java process name.

The second step is using jmap. To call jmap, you need to specify the PID and the location where the heap dump file will be saved. We'll also specify the output is a binary file using the `-dump:format=b` parameter. Figure 11.10 shows the use of this tool in the command line.

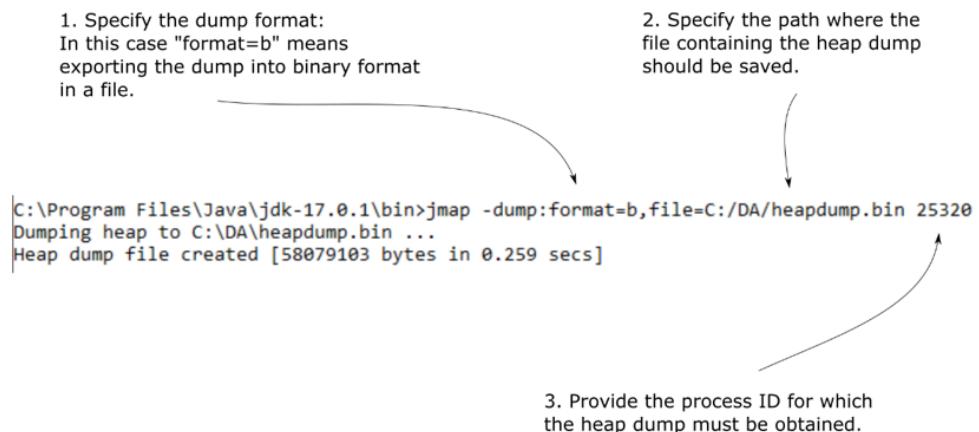


Figure 11.10 Using jmap in the command line to get a heap dump. You need to specify the path where the file containing the dump will be saved and the process ID for which you generate the dump. The tool saves the heap dump as a binary file in the requested location.

To copy the command more easily, you can find it in the next snippet also:

```
jmap -dump:format=b,file=C:/DA/heapdump.bin 25320
```

Now you can open the file that was saved by jmap in VisualVM for investigation.

11.2.2 Reading a heap dump

In this section, we focus on using a heap dump to investigate memory allocation issues. The heap dump is like a “picture” of the memory the way it looked when it was generated. The heap dump contains all the data the app had in the heap, which means you can use it to find out all the details about the data and the way it was structured. This way, you can find which objects were occupying a big part of the allocated memory and understand why the app couldn't deallocate them.



Remember that in the "picture" (heap dump), you can see everything once you took it. If unencrypted passwords or any kind of private data is in memory, someone having the heap dump will be able to get them.

Unlike a thread dump, you cannot analyze a heap dump as plain text. You can use VisualVM (and any profiling tool in general) to analyze a heap dump. In this section, we use VisualVM to analyze the heap dump we generated for project da-ch11-ex2 in section 11.2.1. You'll learn the approach for finding out the root cause of an `OutOfMemoryError`.

When you open a heap dump in VisualVM, the first view the profiling tool shows is the summary view for the heap dump (figure 11.11). The summary view shows you quick details on the heap dump file (such as the file size, the total number of classes, and the total number of instances in the dump). You can use these details to figure out if you have the correct dump in case you weren't the one to extract it.

In some cases, I was getting heap dumps I had to investigate from a support team that had access to the environments where the app was running. I couldn't access those environments myself, so I had to rely on someone else to get me the data. More than once I had the surprise that the heap dump they gave me was not the right one. I was able to identify that fact by looking at either the size of the dump and comparing it to what I know the maximum value was configured for the process, or even at the operating system or the Java version.

My advice is to first quickly check the summary page and figure out at least that you have the correct file before you lose time investigating some wrong file. On the summary page, you'll find also a top of the types that occupy a big amount of space. I usually don't rely on this summary but go directly to the objects view where I start my investigation. In most cases, the summary wasn't enough for me to draw a conclusion.

The summary shows quick details about the dump and the environment where the app was running.

For a real-world app, the heap dump is usually much larger than the one in our example.

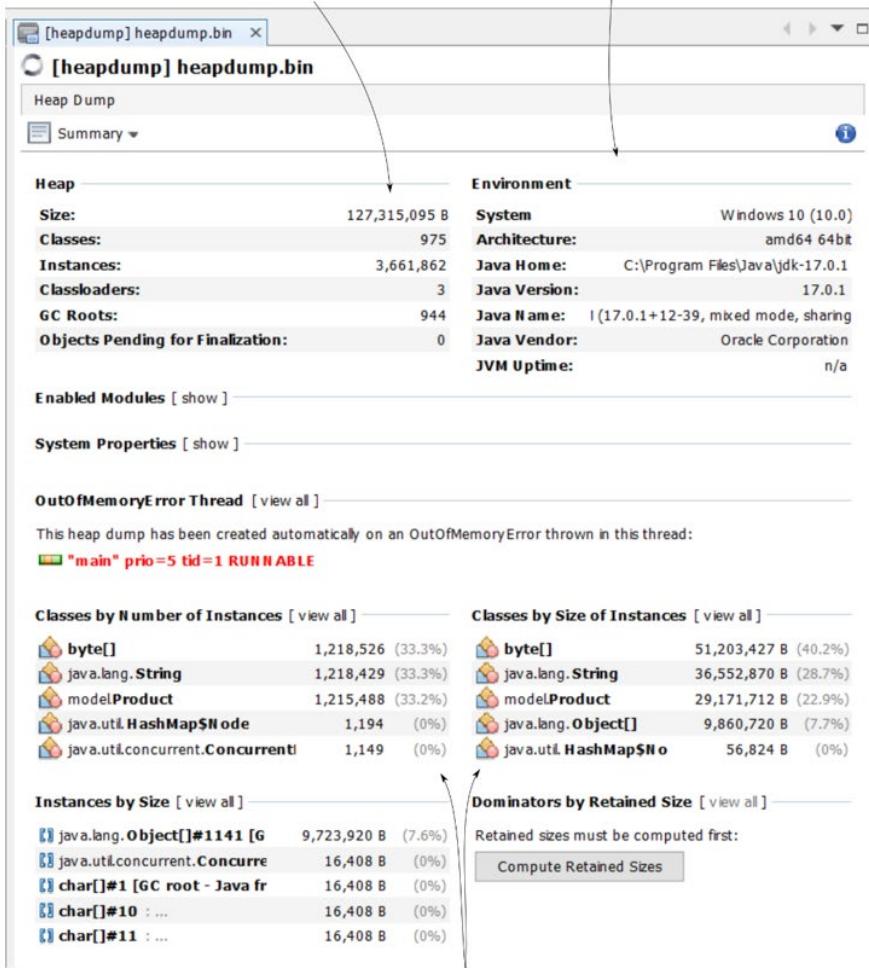


Figure 11.11 In the initial screen after opening a heap dump, VisualVM shows a summary of the heap dump details. You can quickly find details about the dump itself and the system where the app was running. Also, the view shows the types that occupy the largest amount of memory.

To change the view to the objects view, select the **Objects** button from the dropdown in the left upper corner of the heap dump tab (figure 11.12). The objects view will help you investigate the object instances in the heap dump.

To get a view of all the types of objects in the heap dump, change the view to Objects from the left upper side of the tab.

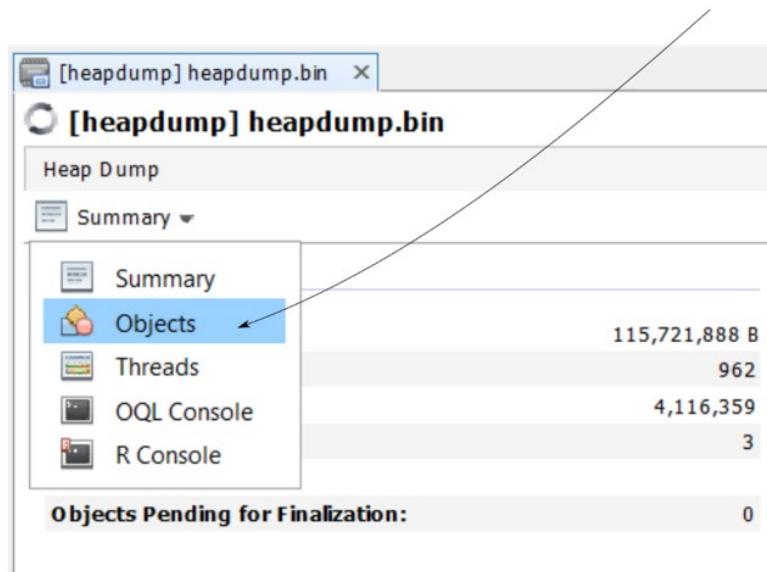


Figure 11.12 You can change the view from the dropdown in the left upper corner of the heap dump tab. To change the view to the Objects view that enables you to investigate easier the instances in the heap dump, select the Objects button.

Same as in the case of memory sampling and profiling, we're searching for the types that cause most of the memory to get occupied. The best approach I use is to sort descending by both the number of instances and occupied memory and look for the first types that are part of the app's codebase. Don't look for types such as primitives, strings, or arrays of primitives and strings. They're generally many as a side effect of the type that's causing the issue and they won't give you too many clues on what is wrong.

In figure 11.13 you can observe that after sorting, the `Product` type seems to be the one involved in the fault. The `Product` type is the first type which is part of the app's codebase that occupies a large part of the memory. We'll need to find out why so many instances have been created, and why couldn't the garbage collector (GC) remove them from the memory.

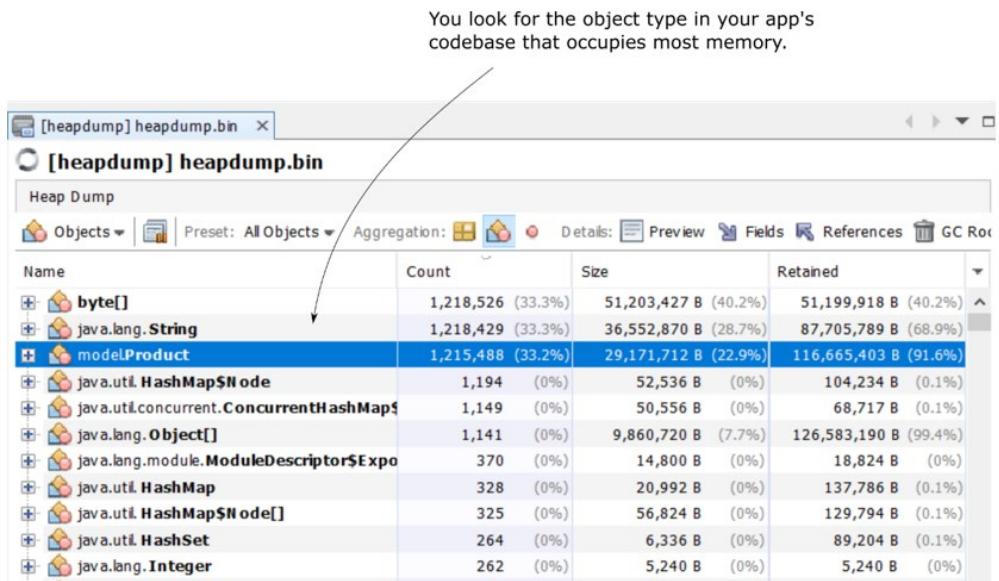


Figure 11.13 Use sorting on columns to identify which type created a large number of instances or occupy a lot of space. You always look for the first object in your app codebase. In this case, both as a number of instances and size, the Product type is the first in the list.

You can select the small (+) button on the left side of the row to get details about all the instances for that type. We already know there're more than 1 million Product instances, but we still need to find

- What part of the code creates those instances
- Why the GC couldn't remove them in time to avoid the app's failure

For each instance, you can find who it is referring to (through fields) and who refers to that instance. Since we know the GC cannot remove an instance from the memory unless it has no referrers, we look for who refers the instance. This way, we'll understand if the instance is still needed in the processing context or if the app forgot to remove its reference.

Figure 11.14 shows the expanded view for the details of one of the `Product` instances. We observe that the instance refers to a `String` (the product name), and its reference is kept in an `Object` array which is part of an `ArrayList` instance. Moreover, the `ArrayList` instance seems to keep a large number of references (over 1 million). Usually, this is not a good sign, either the app implements a non-optimized capability or we found a memory leak.

To understand precisely which is the case, you need to investigate the code using the debugging and logging techniques we discussed in chapters 2 through 5. Fortunately, the profiler shows you exactly where to find the list in the code. In our case, the list is declared as a static variable in the `Main` class.

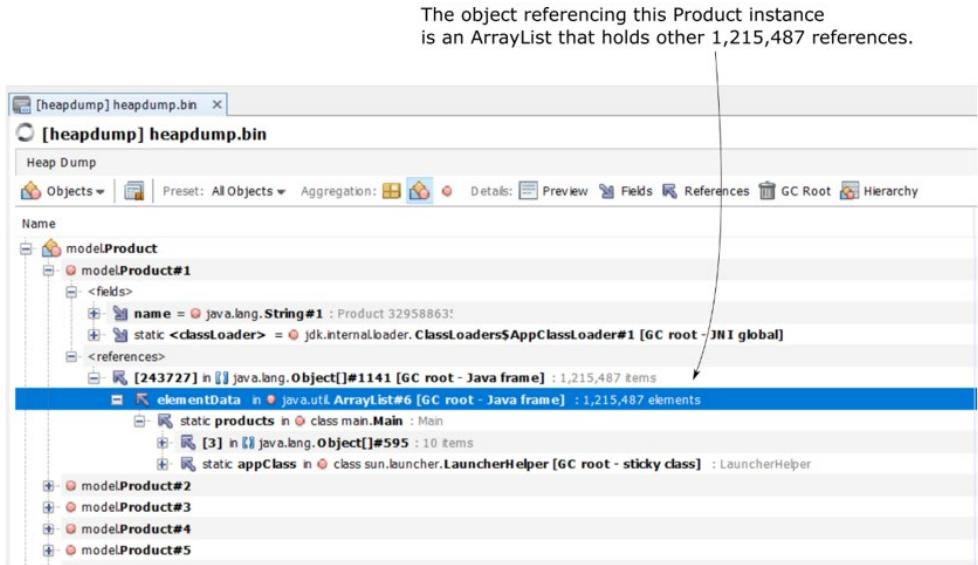


Figure 11.14 References to an instance. Using the heap dump you can find for each instance what other instances were referring it at the time the dump was generated. Also, the profiling tool tells you where in code is a given reference stored. In this case, the ArrayList holding over one million references is a static variable in the Main class.

Using VisualVM we can easily understand the relationships among objects and combined with other investigation techniques you learned throughout the book, you have all the needed tools to address any kind of issue. Depending on the problem's (and app's) complexity it might take you longer (than for our simple example) to identify an issue. But using this approach still helps you spare a lot of time.

11.2.3 Using the OQL console to query a heap dump

In this section, we'll discuss a more advanced way of investigating a heap dump. We'll use a querying language similar to SQL to fetch details from the heap dump. This language is named Object Querying Language (OQL). In most cases, using the simple approaches we discussed in section 11.2.2 was enough for me to identify memory allocation problems' root causes. But the simple approach wasn't enough helpful when I needed to compare the details of two or more heap dumps.

Suppose you want to compare the heap dumps provided for two or more versions of an app to find out if something faulty or non-optimized was implemented in between the version releases. You could take each of them one by one and investigate them manually. But I'll show you a better approach that will save you plenty of time. Instead of manually researching each heap dump, write queries that you can easily run on each of them. That's where OQL is an excellent approach.

Figure 11.15 shows you how to change the view to the OQL console where you can run queries to investigate the heap dump.

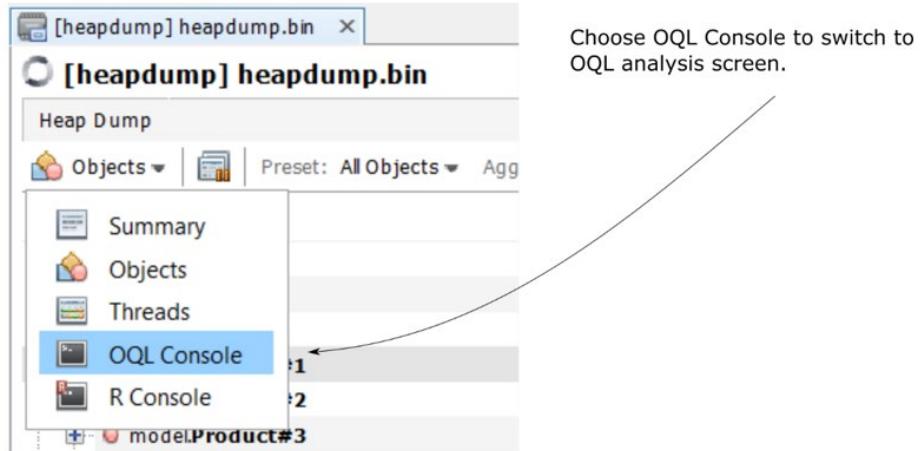


Figure 11.15 To change the view to OQL view in VisualVM, choose OQL Console from the dropdown in the left upper corner of the heap dump tab.

We'll discuss a few examples I found most useful, but remember that OQL is more complex than just the examples I'm going to show you. You can find more details on functions on this page: http://cr.openjdk.java.net/~sundar/8022483/webrev.01/raw_files/new/src/share/classes/com/sun/tools/hat/resources/oqlhelp.html

Let's start with a simple one – selecting all the instances of a given type. Say we want to get all the instances of type `Product` from the heap dump. If we were to write a SQL query to get all the product records from a table in a relational database we would have written something similar to the query in the next snippet.

```
select * from product
```

To query all the `Product` instances in a heap dump using OQL you need to write the query presented in the following snippet.

```
select p from model.Product p
```



NOTE For OQL, keywords such as “select,” “from”, or “where” are always written with a lowercase. The types are always given with their fully qualified name (package + class name).

Figure 11.16 shows the result for executing the simple query that retrieves all the Product instances from the heap dump.

The screenshot shows the VisualVM interface with the title bar "[heapdump] heapdump.bin". Below it, a sub-tab "[heapdump] heapdump.bin" is selected. A "Heap Dump" section is visible. At the top of the main pane, there's an "OQL Console" dropdown, a "Results" button, and a "Results Limit: 100" dropdown. Below these, a list of results is shown, each preceded by a blue link: "modelProduct#1", "modelProduct#2", "modelProduct#3", "modelProduct#4", "modelProduct#5", "modelProduct#6", "modelProduct#7", "modelProduct#8", "modelProduct#9", and "modelProduct#10". At the bottom of the window, a text box contains the OQL query:

```

1 select p from model.Product p
2
3

```

Three numbered callouts point to specific elements:

1. Write the OQL query in the text box.
2. Select the run button.
3. After running an OQL query, the results are displayed above the query box.

Figure 11.16 Running an OQL query with VisualVM. In the OQL console, write the OQL query in the textbox on the bottom of the window and select the run button (the green arrow on the left of the text box) to run the query. The results will appear above the text box.



NOTE When studying OQL, use small heap dumps.
Real-world heap dumps are usually large (4Gb or over).
The OQL queries will run slowly and will take you much time.
If your purpose is only studying, generate and use small-sized
heap dumps same as we do in this chapter.

You can select any of the queried instances to get details about that instance. You can access the details about who keeps a reference to that instance, who that instance refers to, and values. These details will help you investigate certain scenarios you work on.

Selecting any of the rows in the result
(which represents an object instance)
will give you details about that instance.

```

[heappump] heappump.bin
[heappump] heappump.bin
Heap Dump
Product#1 Details: Preview Fields References GC Root Hierarchy
Name
modelProduct#1
<fields>
  static <classLoader> = jdk.internal.loader.ClassLoaders$AppClassLoader#1 [GC root - JN I global]
  name = java.lang.String#1 : Product 32958863!
<references>
  [243727] in java.lang.Object[]#1141 [GC root - Java frame] : 1,215,487 items
  elementData in java.util.ArrayList#6 [GC root - Java frame] : 1,215,487 elements

```

Figure 11.17 You can access the details about a queried instance (referees and referrers) by clicking on the queried instance.

You can as well directly select values or references referred from certain instances. For example, if we wanted to get all the product names instead of the product instances, we could have written the following query (figure 11.18):

```
select p.name from model.Product p
```

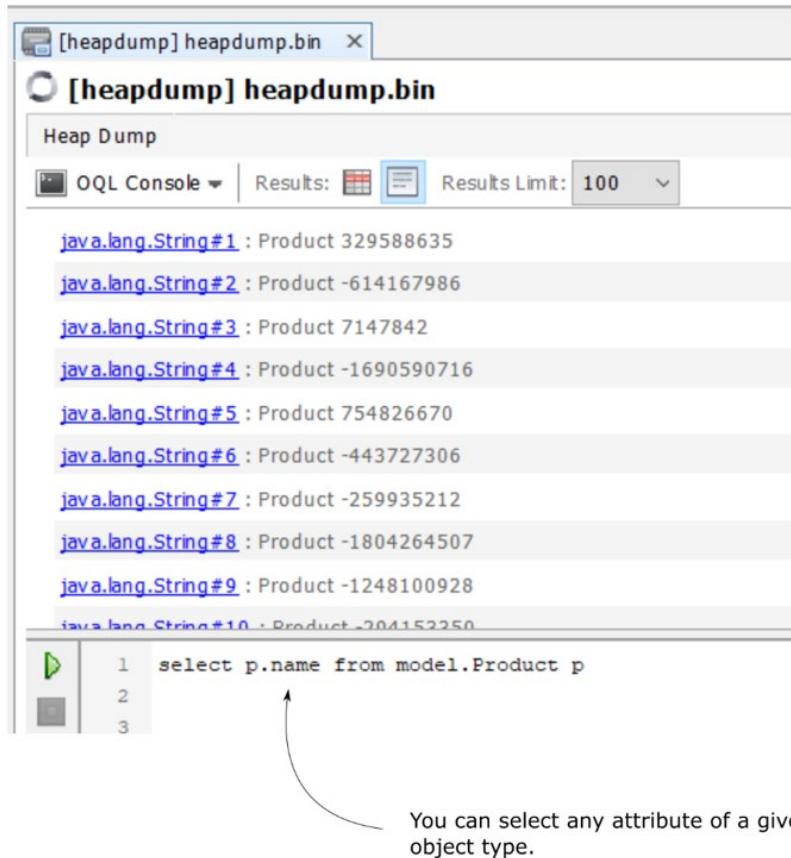


Figure 11.18 Selecting an attribute of a given object type. Just as in Java, you can use the standard dot operator to refer to an attribute of an instance.

With OQL, you can extract multiple values at the same time. To do so, you need to format them as JSON.

```

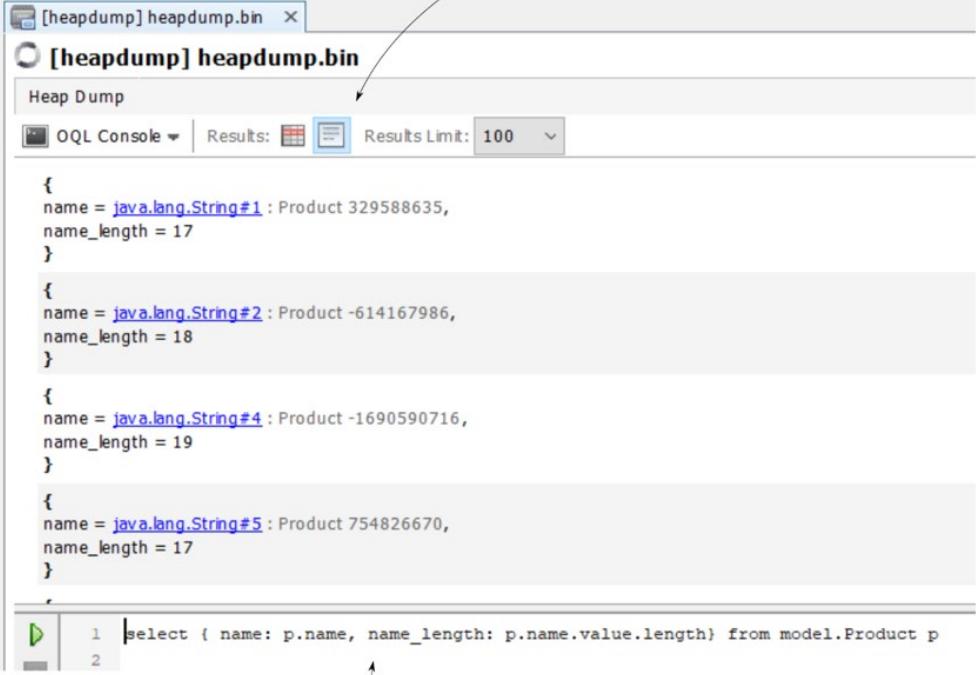
select
{
    #A
    name: p.name,      #B
    name_length: p.name.value.length      #C
}
from model.Product p

#A Curly braces surround the JSON object representation
#B Attribute name takes the value of the product name
#C Attribute name_length takes the value of the product name number of characters

```

Figure 11.19 shows you the result for running this query.

To see the results easier, use the formatter display.



The screenshot shows the heappump interface with the title bar "[heappump] heapdump.bin". Below it, a section titled "Heap Dump" contains a list of selected objects:

```
{
name = java.lang.String#1 : Product 329588635,
name_length = 17
}
{
name = java.lang.String#2 : Product -614167986,
name_length = 18
}
{
name = java.lang.String#4 : Product -1690590716,
name_length = 19
}
{
name = java.lang.String#5 : Product 754826670,
name_length = 17
}
```

Below this, a code editor window shows the OQL query:

```
1 | select { name: p.name, name_length: p.name.value.length} from model.Product p
2 |
```

To select multiple values, use JSON formatting.

Figure 11.19 Selecting multiple values. You can use JSON formatting to obtain multiple values with one query.

You could change this query to, for example, add conditions on one or more of the selected values. If you want to only select the instances that have a name with a length larger than 15 characters, you could write a query as presented in the next snippet.

```
select { name: p.name, name_length: p.name.value.length}
from model.Product p
where p.name.value.length > 15
```

Let's move on to something slightly more advanced. A query I often use when looking into memory issues uses the `referrers()` method to get the objects that refer to instances of a specific type. Yes, using built-in OQL functions such as this one you can do plenty of useful things including:

- **Find or query instance referees** – can tell you if the app has memory leaks.
- **Find or query instance referrals** – can tell if specific instances are the cause of memory leaks.
- **Find duplicates in instances** – can tell you if specific capabilities can be optimized to use less memory.
- **Find subclasses and superclasses of certain instances** – gives you insight into an app's class design before or without needing to see the source code.
- **Identify long lifepaths** – can help you identify memory leaks.

To get all the unique referrals for instances of type `Product` you can use the following query:

```
select unique(referrers(p)) from model.Product p
```

Figure 11.20 shows the result for running this query. In this case, we can see that all the product instances are referred to by only one object – a list. Usually, when a large number of instances have a small number of referrals it's a sign of a memory leak. In our case, a list keeps references to all the `Product` instances, preventing the GC to remove them from memory.

Running the query, you easily observe that all the products have an unique referrer.

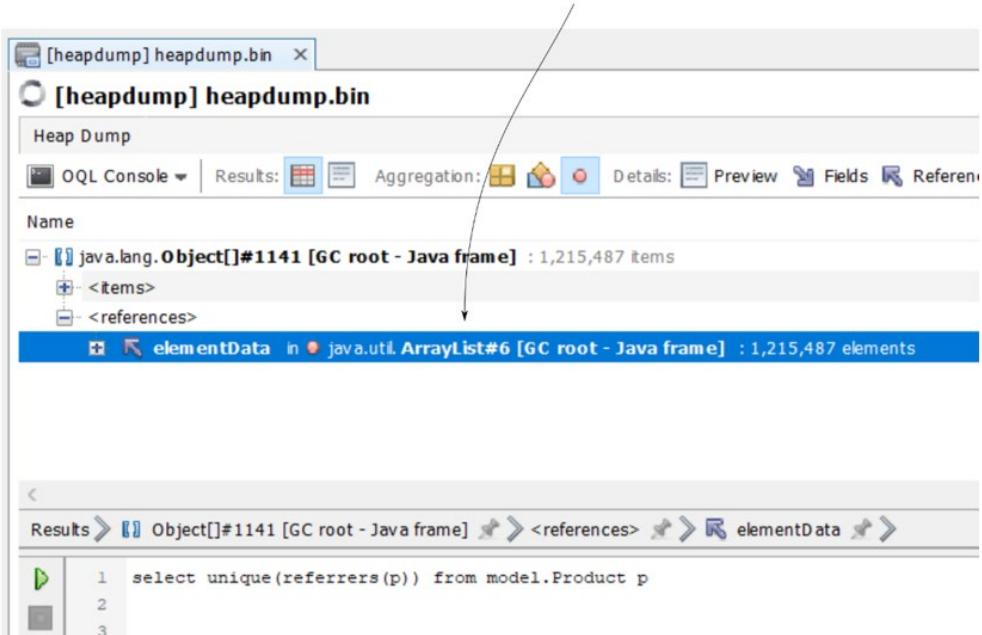


Figure 11.20 Selecting all the unique referrers for instances of a type gives you a clue if there's one object holding that prevents the GC from removing the instances from memory. This can be a quick way to identify a memory leak.

If the result is not unique, you can count the referrals by instance using the next query to find the instances that are potentially involved in a memory leak:

```
select { product: p.name, count: count(referrers(p))} from model.Product p
```

The OQL queries give a lot of opportunities, and, once you wrote a query you can run it as many times as you need and on different heap dumps.

11.3 Summary

- An app with capabilities that are not optimized from the memory allocation point of view can cause performance problems. Optimizing the app to wisely allocate (avoid spending unnecessary memory space) the data in memory is essential to have a performant app.
- A profiling tool can help you sample and profile how the memory gets occupied during an app's execution. This capability of a profiler helps you identify non-optimized parts of your app and gives you details on what could be improved.
- If during execution new object instances are continuously added to the memory, but the app never removes the references to new instances, the garbage collector won't be able to remove the references and free the memory. When the memory gets fully occupied, the app cannot continue its execution and stops. Before stopping, the app throws an `OutOfMemoryError`.
- To investigate an `OutOfMemoryError`, we use heap dumps. A heap dump collects all the data in the app's heap memory and allows you to analyze it to figure out what went wrong.
- You can start your app using a couple of JVM arguments to instruct it to generate a heap dump at a given path if the app fails with an `OutOfMemoryError`.
- You can also get a heap dump using a profiling tool or a command-line tool such as `jmap`.
- To analyze a heap dump, load it in a profiling tool such as VisualVM. VisualVM helps you investigate the instances in a heap dump and their relationships. This way, you can figure out what part of the app is not optimized or has memory leaks.
- VisualVM offers you also more advanced ways to analyze the heap dump, such as OQL queries. OQL is a querying language similar to SQL, but which you use to fetch data from a heap dump.

12

Investigating apps' behavior in large systems

This chapter covers

- Investigating app communication issues
- Using log monitoring tools in your system
- Taking advantage of deployment tools

In this chapter, we go beyond the border of an app and discuss how to investigate situations caused by apps working together in systems. Today, many systems are composed of multiple apps communicating with one another. Large business systems behind an enterprise's gates leverage various apps, often implemented with different technologies and on different platforms. In many cases, the maturity of these apps also varies from brand new services to old and messy scripts.

Debugging, profiling and logs aren't always enough. Sometimes you need to find clues on a larger horizon. An app can independently work well, but not integrate correctly with other apps or the environment it's deployed into.

We start in section 12.1 with ways for investigating communication between services of a system. In section 12.2 we focus our attention on the relevance of implementing monitoring for the apps in a system and how to use the details a monitoring tool provides. We end this chapter's discussion in section 12.3 where we discuss what possibilities you have to take advantage of the deployment tools to help your investigations.

12.1 Investigating communication between services

In this section, we discuss investigating communication between apps. In a system, apps "talk" to one another to fulfill their responsibilities. Up to now in this book, we focused on

investigating how an app works inside, and we also talked about investigating the communication between an app and a database management system. But what about apps talking to one another? And is there a way to monitor the essential events throughout a whole system composed of many apps (figure 12.1)?

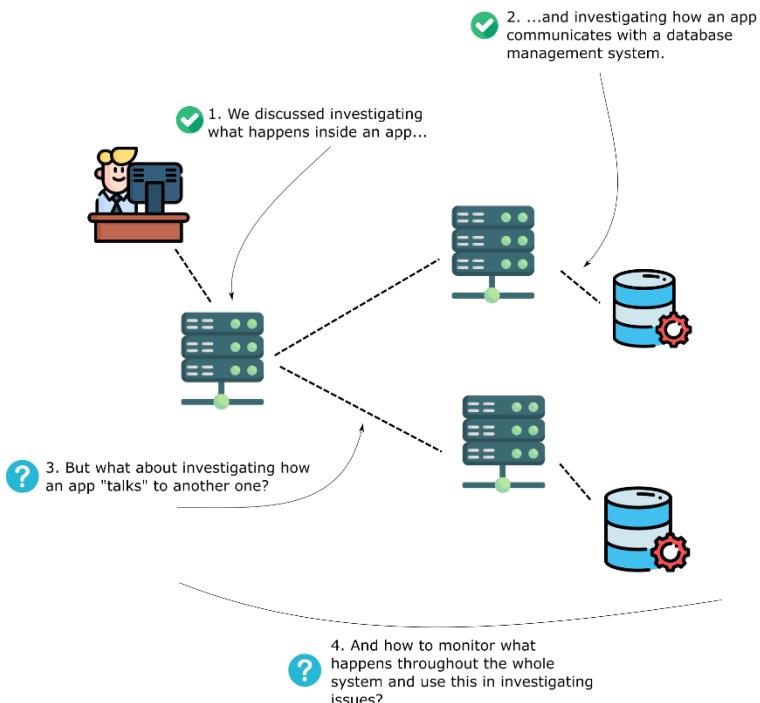


Figure 12.1 In many cases investigation remains inside the boundaries of an app. But it often also happens you need to go beyond what happens inside a given process. Issues or strange behavior can be caused by apps that have problems communicating with one another. Expect to need to focus sometimes on how data flows between apps when investigating issues and make your life easier by implementing a monitoring tool to make you aware of exceptional events happening throughout the system.

Let's discuss how to use a profiling tool for investigating issues with how apps "talk" to each other. We'll use JProfiler to observe the way communication works for a simple app. This app that you find as project da-ch12-ex1 provided with the book, exposes an endpoint you can call (the /demo endpoint). When you send an HTTP request to this endpoint, the app further sends a request to an endpoint exposed by HttpBin.org. HttpBin.org delays the response 5 seconds and then responds back with a 200 OK HTTP status on the response.

As you'll learn in this section, JProfiler offers a set of tools that you can use to observe both the requests an app receives as well as the requests an app sends. Moreover, you can also investigate low-level communication events on sockets. Such approaches will help you identify the root cause of any communication problem you investigate.

In section 12.1.1, we use JProfiler to observe the requests an app receives. In section 12.1.2, we'll investigate details about the requests an app sends, and in section 12.1.3, we focus on investigating low-level communication events on sockets.

Quickly on microservices

Let's talk frankly about microservices. Many systems you'll work on nowadays claim they are microservices. Most often this is not true and the systems you'll work on are just simple service-oriented architectures. Microservices became (for some reason I can't say I fully understand) a brand that sells quite well:

Do you want to employ someone faster? Tell them they're going to work with microservices.

Do you want to impress a customer during a presales meeting? Tell them you do microservices.

Do you want more people to attend your presentation? You guessed, just add microservices in the title.

But microservices are more difficult than what I feel the majority of developers understand. You'll find plenty of literature out there if you consider you want to get better into what microservices are. You can start with "Microservices patterns" by Chris Richardson (Manning, 2018), and then read further "Monolith to Microservices" by Sam Newman (O'Reilly Media, 2018), or "Building Microservices: Designing Fine-Grained Systems, 2nd Edition" (O'Reilly Media, 2021) also by Sam Newman.

Now, whether they are real microservices systems or not, you'll still need to know how to investigate problems and how to quickly understand what the system does in given scenarios. In this chapter, we're going to discuss investigation techniques that apply to microservices **but not only**. I prefer to use the simple "service" term instead of "microservice". Sometimes I'll just use "app" or "application."

12.1.1 Using HTTP server probes to observe HTTP requests the app gets

When discussing communication between two apps, we observe the data flows in two directions. It's either the app sending a request or receiving it. When the app sends requests, we say it acts as a client, when it receives requests, we say it's the server. In this section, we focus on the HTTP requests the app receives (as a server). We'll use a simple app provided with the book as project da-ch12-ex1 to understand how to monitor such events with JProfiler.

Open project da-ch12-ex1 in your IDE and start the application. Using JProfiler, you connect to the app and then start recording the received HTTP requests going to **HTTP Server > Events** and selecting the record events button on the page. Figure 12.2 shows you how to start recording the events. We want to learn what HTTP requests the app receives and what details can we find out about these requests.

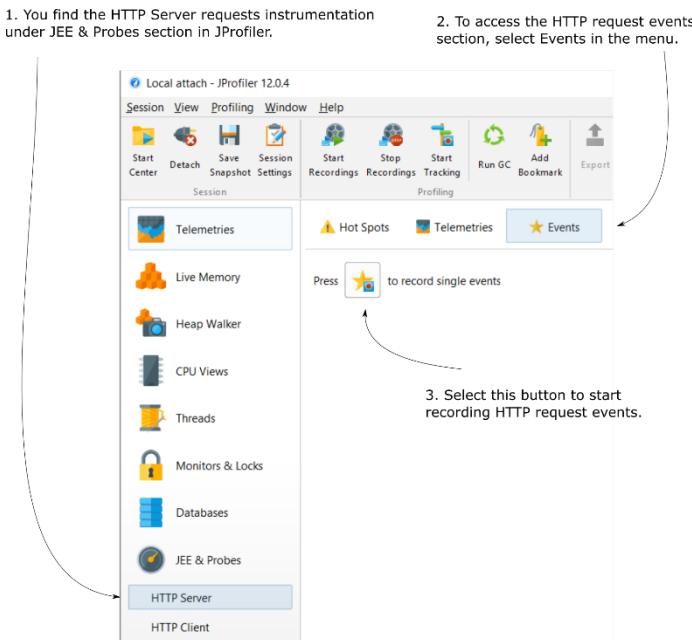


Figure 12.2 To start recording received HTTP requests with JProfiler go to HTTP Server > Events and then select the record single events button on the page. After doing this, whenever the profiled app receives HTTP requests, JProfiler will display their details on the page.

Let's call the only endpoint this demo app exposes:

```
curl http://localhost:8080/demo
```

As presented in figure 12.3, JProfiler shows the request the server received. First of all, you can easily figure out when the event ends. When the event ends, the displayed status will be "completed". If the operation never ends, meaning for some reason either the request hasn't been fully processed or the response haven't been fully sent back to the client, you'll continue to see the status as "in progress". You can determine this way if the request takes too long or it has been delayed to interrupted by something. Usually, in such a case, where the request takes longer than expected, you continue your investigation with what we'll discuss in section 12.1.3 where you'll learn how to observe low-level events on sockets.

The HTTP server events table also displays the event duration. If the event is completed but takes a long time, you will need to figure out what caused the delays. It can be faulty communication, that you observe using socket events discussed in section 12.1.3, or you'll need to sample and profile the execution as discussed in chapters 7 through 9.

Besides the event taking a long time, one essential aspect is to observe how many events the app gets. In some cases, one request can't cause any trouble, but I remember a situation once where an app was affected by one of the clients polling (repeatedly sending requests at a short time) one of the endpoints. If a client sends a high number of requests in

a short time interval and there's nothing preventing the requests to reach the app, the app might get into trouble responding to all and even crash.

When the app gets an HTTP request, the event appears in the Events table and you can observe its details.

The events status shows if the event is still in progress or completed.

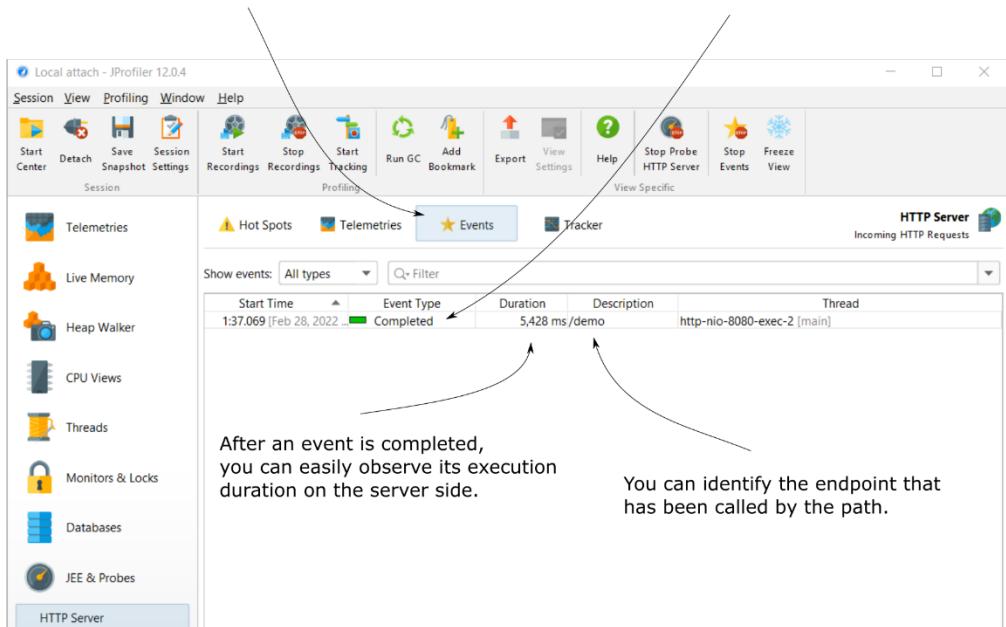


Figure 12.3 After starting to record the HTTP server events (HTTP requests the app receives) the profiling tool shows details of all the received events on the page. You can easily observe if an event ended and the time it took to get completed.

12.1.2 Using HTTP client probes to observe HTTP requests the app sends

Similar to HTTP Server events (which are HTTP requests the app receives), you can profile the HTTP Client events (the HTTP request the app sends). In this section, we discuss profiling HTTP requests the app sends to identify potential issues they cause. To learn using JProfiler to observe such events, we'll continue using the app provided with project da-ch12-ex1, the same one we used in section 12.1.1 as well. This app sends a request to an endpoint of HttpBin.org when someone calls the /demo endpoint it exposes. Let's start the app, call the /demo endpoint and find out if we can observe the HTTP requests this app sends. Then, we discuss what is essential to know about these events when investigating a scenario.

After starting the app, start recording the HTTP Client events in JProfiler (figure 12.4) and call the /demo endpoint:

```
curl http://localhost:8080/demo
```

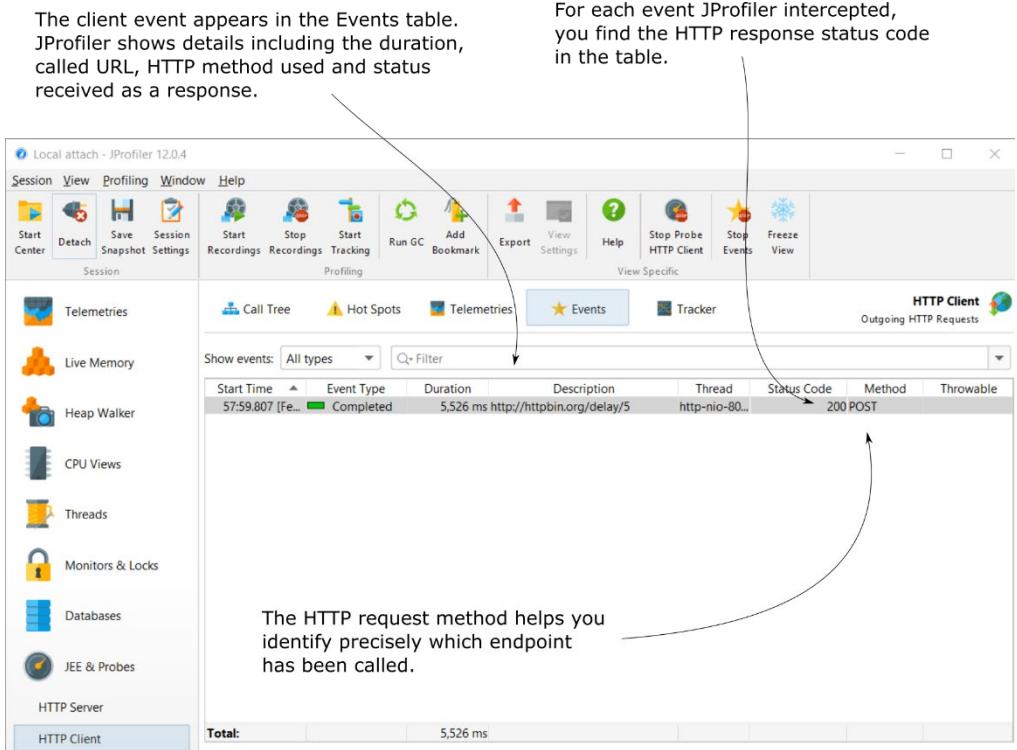


Figure 12.4 Using JProfiler, you can record all the HTTP requests the app sends regardless of the way they're sent (what technology your app uses). The tool displays details such as the duration, the status code, the HTTP method and URI called, and whether an exception has been encountered. All these details are very useful for investigating specific scenarios that involve HTTP requests the app sends.

Take a look at the details you get using this tool (figure 12.4). You're first interested in the description and method columns since they help you identify what endpoint is the app calling. Once you know who's your app calling, the details giving you the most insight are the call duration, the response status code, and whether an exception has been encountered.

If you find out that a call takes a long time to execute (more than you expected), you will want to figure out what exactly takes so much time. First, you would try to figure out if the problem is caused by the data exchange (over the network) or something inside the app (for example deserializing the response or processing it).

As you'll find out in section 12.1.3, investigating the low-level events on sockets can tell you if the problem is the communication itself or whether you should look at something your app does. If you discover the data exchange is not what's causing the problem, you can apply the profiling techniques we discussed in chapters 7 through 9 to discover what affects the app's execution performance.

Same as in the case of HTTP requests an app gets (as discussed in section 12.1.1), an important thing to consider is the event count (how many lines appear in the events table).

Does your app send too many requests causing maybe the other service to respond slower? In one of the apps I've been implementing some time ago, I found out that the app was sending frequent requests because of a faulty retry mechanism. The problem was difficult to spot upfront since the requests were made to retrieve some data and they were not changing anything to result in a wrong output that would have been most likely easier to observe. In this case, the supplementary requests were only affecting the app's performance.

12.1.3 Low-level investigations on sockets

In this section, we discuss investigating low-level communication events on sockets. In sections 12.1.1 and 12.1.2 I mentioned that observing low-level events is the next step in investigating if a communication problem is caused by the communication channel (for example, the network) or by something faulty inside the app. To observe these low-level events you can use JProfiler by going to the **Sockets > Events** section as presented in figure 12.5.

Start the application, start registering the events in JProfiler, and then send a request to the /demo endpoint:

```
curl http://localhost:8080/demo
```

JProfiler intercepts all the events on sockets and displays them in a table as presented in figure 12.5.

JProfiler intercepts all the socket events and displays their details. The most important details are the event type, duration and the throughput.

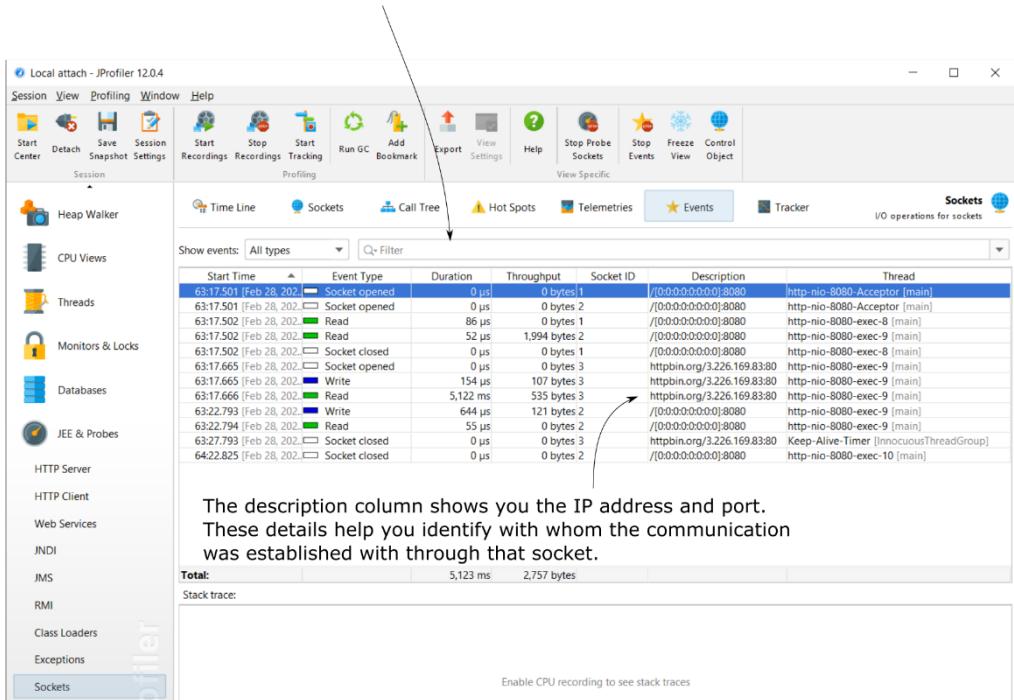


Figure 12.5 Any message an app exchanges through the network level uses sockets behind the scenes. You can use a profiling tool such as JProfiler to observe all the low-level events at the sockets level. To monitor these events use the Sockets > Events section. These events help you understand if the app faces networking issues or if it simply doesn't correctly manage the communication.

A socket is a gateway to another process your app communicates with. When establishing a communication, your app will execute the following socket events (figure 12.6):

1. Open a socket to establish the communication (handshake with the app it needs to talk to).
 2. Read from the socket (receive data) or write through it (send data).
 3. Close the socket.

1. To exchange data with another app (read/write), a service needs first to open a socket.

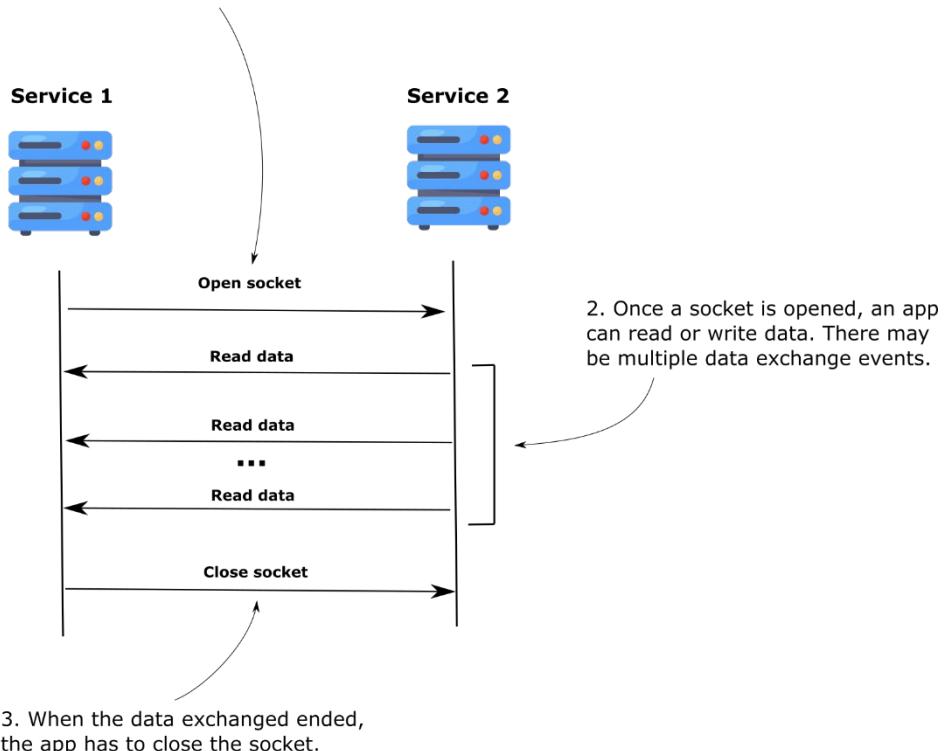


Figure 12.6 When an app starts a data exchange, it first opens a socket. To exchange the data, the app can execute multiple data exchange events (read or write data events). When the data exchange ends, the app has to close the socket.

Let's discuss these three kinds of events in more detail and understand what they tell you about your app's behavior.

OPENING A SOCKET TO ESTABLISH COMMUNICATION

One thing that should draw your attention is a long execution time for the open socket event. Opening a socket shouldn't take a long time. If it does take a long time, this indicates a problem with the communication channel. For example, the system or virtual machine on which the app runs might not be properly configured or the network could have issues. When the open socket event takes a long time, usually it is not caused by a problem with your code.

WRITING DATA THROUGH THE SOCKET OR READING DATA FROM IT

Reading or writing data through the socket is the actual process of communication. The two apps are connected to each other and they are now exchanging data. If this operation is

slow, it can be because either a large amount of data is transferred or the communication channel is slow or faulty.

Using JProfiler, you find exactly what amount of data is being sent through the socket (see column Throughput in figure 12.5). So you can conclude if the slowness is caused by the amount of data or something else. In our example, you can see that the app received a very small amount of data (only 535 bytes) but it had to wait over 5 seconds for this. In such a case we can conclude that the problem is not with the current app, but rather with either the communication channel or the process that our app talks to.

The app we use for our example calls on purpose an endpoint HttpBin.org that causes 5 seconds delay. So our conclusion is indeed right – the other communication endpoint causes the slowness.

CLOSING THE SOCKET

Closing a socket is an event that doesn't cause slowness, but that should always exist for any opened socket. Closing the socket allows the app to free resources allocated to the socket. When the communication ends, the app needs to close the socket.

12.2 The relevance of integrated log monitoring

Many systems today adopt a service-oriented approach and grow in the number of apps with time. These apps communicate with one another, exchange, store, and process data executing business cases the users need. With the increase in the number of applications and the applications' sizes throughout time, the systems become more and more difficult to monitor. Easily noticing where something goes wrong becomes considerably more challenging. To identify easier the system parts that cause issues you can use the capabilities a log monitoring tool offers (figure 12.7).



A **log monitoring tool** is software you can integrate with apps of your system to easily observe exceptions happening throughout the whole system.

The tool observes the executions of all apps and collects data whenever an app throws a runtime exception. It then displays these details in a comfortable way to help you faster identify the problem's cause.

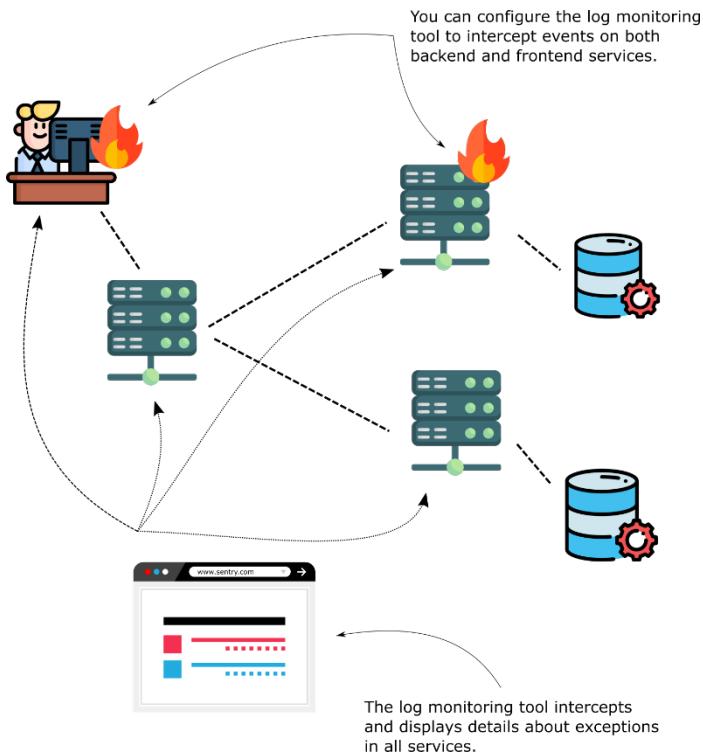


Figure 12.7 A log monitoring tool helps you to easily collect and visualize events throughout the whole system. You can use the details you get to investigate issues and specific app behavior.

We'll use an example of a simple tool you can configure with your system to collect the exception events and present them in an easy-to-read way. **Sentry** (<https://sentry.io>) is a log monitoring tool that I've been using in many of the systems I worked with, has proved to be extremely useful throughout both production and development of the apps. Sentry also has a free plan you can comfortably use for learning purposes as well (for example, for using it with the examples in this chapter).

Let's create an app that throws an exception on purpose and integrate it with Sentry. You find this app in project da-ch12-ex2 provided with the book. The next code snippet shows you the simple implementation of this app. What we want to achieve observe the exceptions caused by this app using Sentry.

```
@RestController
public class DemoController {

    @GetMapping     #A
    public void throwException() {
        throw new RuntimeException("Oh No!");      #B
    }
}
```

```
#A Defining an endpoint you call using HTTP GET.  
#B Throw an exception when you send a request to the endpoint.
```

Integrating an app with Sentry is straightforward. Sentry provides APIs that help you integrate apps developed in a large variety of platforms in only a few lines of code. The official documentation also provides examples and detailed steps on how to integrate your app according to the technologies it uses.

Shortly, the steps you need to follow are simple:

1. Create an account in Sentry
2. Add a new project (representing the app you monitor)
3. Collect the project Data Source Name (DSN) address Sentry provides
4. Configure the DSN address in your project

Once you create an account on [sentry.io](#) (**step 1**) you can add projects (**step 2**). For these two steps, you just follow the instructions on [sentry.io](#) – the process is as simple as creating an account on any website. Each project you add will appear in your dashboard as presented in figure 12.8.

In figure 12.8, “my-demo-project” is the project I created. One or multiple projects can be added to a team. In my case, “my-team” is the team that Sentry created by default when I added the first project. You can rename this team if you like and add others if needed. When you have more apps, you can allocate these apps to teams. Each user can be part of one or multiple teams and can monitor the events of the apps allocated to their teams.

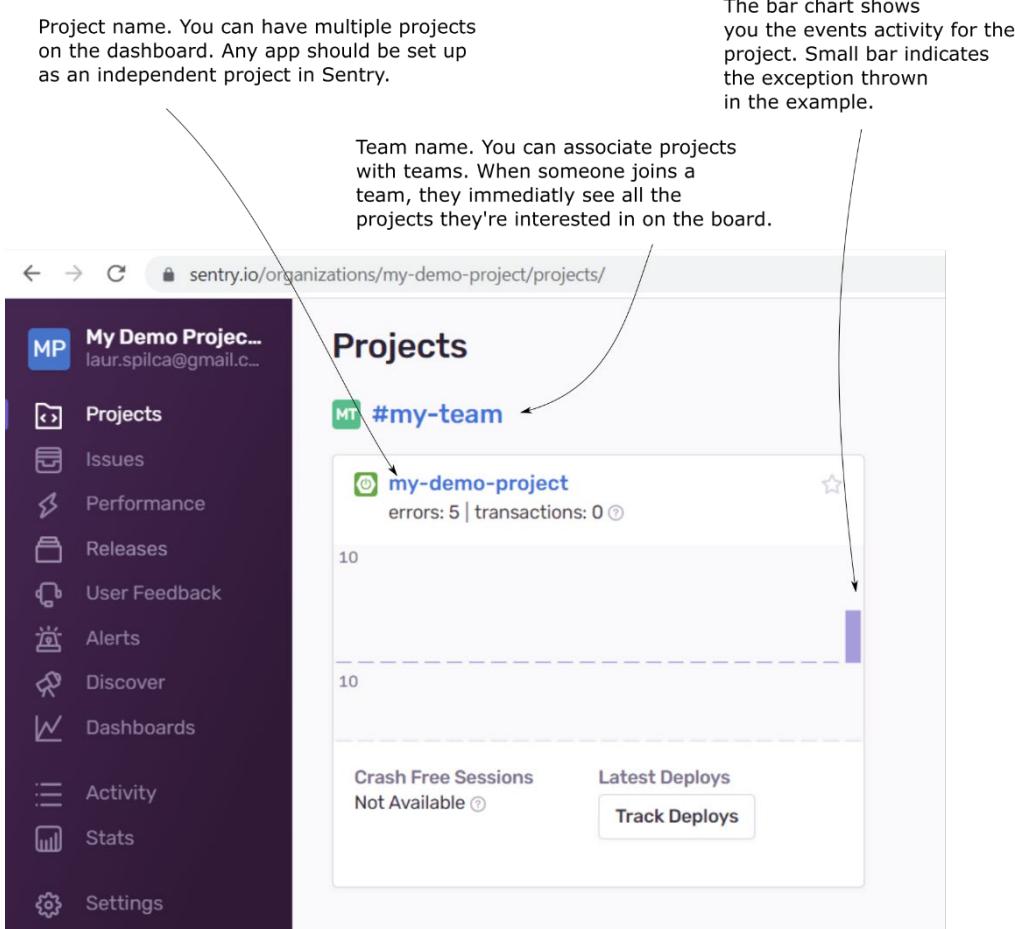


Figure 12.8 Sentry monitors independently the logs of each service in your system. The tool displays a short overview of the events for each service in the initial dashboard. Services (named projects in Sentry) are allocated to teams and Sentry can be configured to send email notifications for events to the team members.

Teams in Sentry are a simple way to organize who takes care of what and make developers accountable for monitoring certain services.

Since your app didn't send any events yet to Sentry, your project won't show a bar on the bar chart yet (as presented in figure 12.8). You first need to tell your app where to send the events. To tell your app how to send the events, you only need to configure the DSN address that Sentry provides as shown in figure 12.9. You find the DSN address in project settings, **Client Keys** section (**step 3**).

Under project settings > Client Keys, you find the DSN address. This is what you need to configure in your app to connect it to Sentry.

The screenshot shows the Sentry interface for a project named "My Demo Project". On the left, there's a sidebar with various navigation options like Projects, Issues, Performance, etc. The main area is titled "Client Keys". It contains a brief description about configuring an SDK with a client key. Below this, there's a table with one row labeled "DEFAULT". The "DSN" field contains the URL `https://ad1facd3a514422bbdaafddacf35bd79@o1136957.ingest.sentry.io/6189082`. There are "Configure", "Disable", and "Delete" buttons next to the table. A callout arrow originates from the text in the first paragraph and points to the "DSN" input field.

Figure 12.9 In the project settings, Client Keys section, you find the Data Source Name (DSN) value. This value takes the shape of an URL. The application uses this URL to send the events to Sentry.

Depending on the app type, Sentry offers different ways in which the configuration is made. You find detailed steps on the official page for each platform (**step 4**): <https://docs.sentry.io/platforms/>

Because the project we're using as an example uses Spring Boot as a platform, we just add the DSN value to the property `sentry.dsn` in the `application.properties` file. You find this configuration in the next snippet. Even if specifying an environment is optional from Sentry's point of view, I always recommend specifying the name of the environment in which the app runs. By specifying the environment, you can filter the events later to get only the ones you'll be interested in. Having the ability to filter the events is very helpful when investigating anything.

```
sentry.dsn=https://ad1facd3a514422bbdaafddacf... #A
sentry.environment=production #B
```

#A The DSN address to connect the app to Sentry.
#B The environment name.

Figure 12.10 shows the way you can access details about exception events that happened in your app. Selecting the **Issues** menu on the left, you access a board where you can browse all the events that Sentry caught from the apps it integrates with. You can filter for which apps you want to see the events, which environment, and the period of time in which you're interested.

This board is the key starting point for an investigation. If you use Sentry and you need to analyze something happening with a service in the system, first make sure to check the events in the issues board. Using Sentry to find exception events is much faster than searching these events in logs as we discussed in chapter 5.

The first thing you find on the board is a list with short details for each audited event. The exception type, its message, and the number of occurrences are the most important details you first need to know.

Another essential detail you find about each event is the last time the event was encountered and the first time it appeared. You can use these details to figure out if the problem is a recurrent one, if it happens frequently or if it's only an isolated case. If the event is an isolated one, you might figure out it's caused by a sporadic problem in the environment, while usually, bugs generated by the app logic are recurrent and more frequent.

As presented in figure 12.10 you find all these details for each event straight in the main issues board.

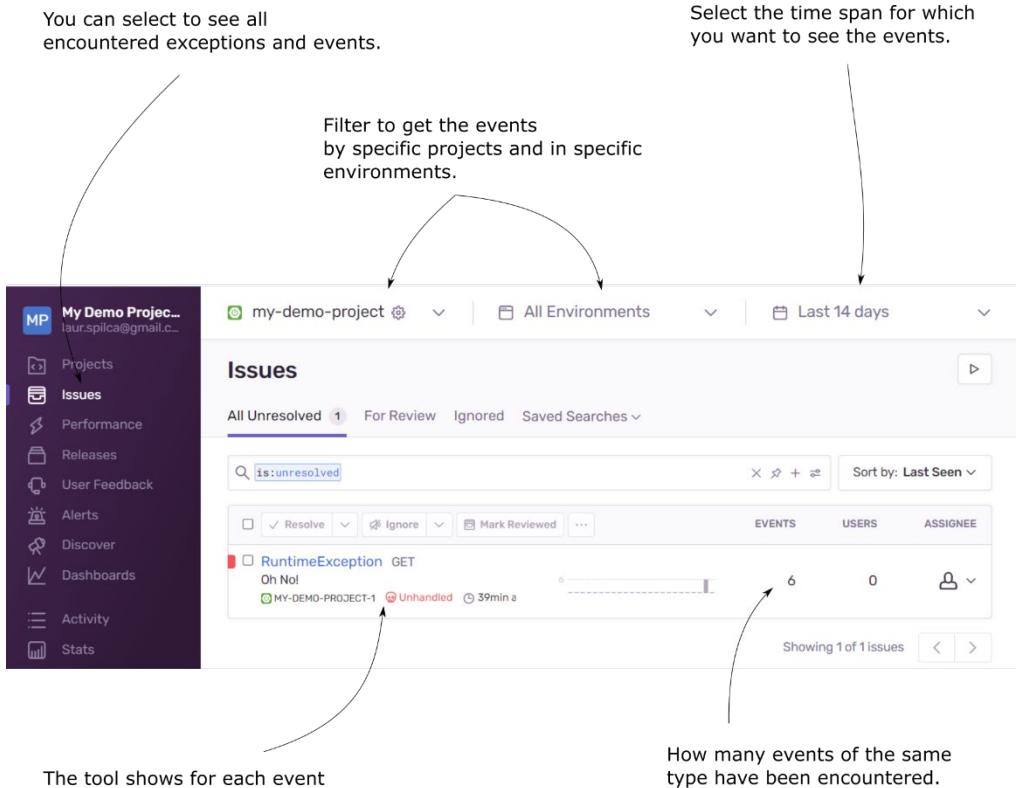


Figure 12.10 Sentry collects all the exceptions caused by any of the monitored services. In the Issues menu, you can browse a list of the issues. Sentry allows you to filter them based on the time when the events happened, the environment in which the event happened, and specific services causing the events. Filtering the events helps you easier identify an issue you're looking for.

If you are interested in more details for a specific event, select the event for which you need to see more details in the main issues board. Sentry collects all the useful details including (figure 12.11):

- The exception stack trace.
- Environment details such as the operating system, runtime JVM and server name.
- Client details in case the exception was caused during an HTTP request.
- Information sent on the request in case the exception happened during an HTTP request.

One of the things I particularly find useful is the fact that Sentry automatically collects the details on the HTTP request in case the exception happened on a thread serving an HTTP request. You can use these details to replicate the problem in a development environment or

try to figure out if any of the data sent through the HTTP request could have potentially caused the exception event. While Sentry doesn't indicate the cause of a problem straightforwardly, you can use the details it provides as pieces of the puzzle you have to solve.



NOTE In many cases today, apps talk to each other over HTTP. For this reason, it's very likely that exception events happen as consequence of a received HTTP request. Sentry takes the details on the HTTP request and associates these details with the event.

This way, Sentry puts together more pieces of the puzzle helping you understand faster what the problem's root cause is.

Exception type and message.

```
RuntimeException
Oh No!
mechanism HandlerExceptionResolver | handled false
com.example.controllers.DemoController in throwException at line 11
Called from: jdk.internal.reflect.NativeMethodAccessorImpl in invoke()
```

Details about the system and the application.

Tags	
browser	Chrome 98.0.4758 100%
browser.name	Chrome 100%
client_os	Windows 10 100%
client_os.name	Windows 100%
environment	production 100%
handled	no 100%
level	fatal 100%
mechanism	HandlerExceptionResolver 100%
runtime	Oracle Corporation 17.0.1 100%
runtime.name	Oracle Corporation 100%
server_name	host.docker.internal 100%
transaction	GET 100%
url	http://localhost:8080/ 100%

BREADCRUMBS

TYPE	CATEGORY	DESCRIPTION	LEVEL	TIME
http	GET /		Info	17:43:14
exception	RuntimetypeException: Oh No!		Error	17:43:14

Details about the event that caused the exception.

```
GET / localhost
Headers
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Show More
```

Figure 12.11 Each of the events that Sentry collects provides details that will help you identify the problem's root cause. You'll find the event stack trace, details about the server and client's environments, and even details about the request (headers, HTTP method, and so on) in case of an HTTP request.

Using Sentry in team management

Even if Sentry is mainly a tool used in auditing, monitoring, and investigating issues with apps, I found an alternative usage for it that I consider quite helpful.

As a development lead, I combine the responsibilities of a team lead with the ones of a technical lead. Before the COVID-19 pandemics, when we used to all work in the office, close to each other, finding out when someone was struggling with something was much easier for me. Same thing from their side: it was much easier to throw a ball of paper towards me to get my attention when they needed me. But things changed with remote, online work.

One of the things that added delays to the team was simply the difficulty added to the communication between team members.

Sentry can be configured to send emails for the events it encounters. What I did is configure it to get emails even for events coming from local environments. This way, I can see what difficulties any team members encounter. And since I know my team well, I know in most cases if someone will be stuck with a specific problem. In some cases, two or more team members experienced the same problem, but since communication is more difficult, they all spent time investigating the same thing.

Using Sentry, I was able to act upfront and help someone before they spent too much time trying to investigate an error and plan team tasks more efficiently. I could also stop them from working when I saw they ran too much over schedule at the same time. Pretty cool, isn't it?

12.3 Using deployment tools in investigations

One of the things I learned with time and working on many projects is that environments hosting apps are different from one another. Besides being different, environments evolve together with technologies and the way we think about architectures and build apps. An important lesson I learned is that understanding properly the environment my apps run in can be tremendously helpful when investigating why an app behaves in a specific way.

Let's discuss one of the latest fashions of deploying service-oriented architectures and how can this way of deploying the apps be helpful when investigating issues your apps might encounter: **service meshes**.

A service mesh is a way to control how different apps in a system communicate with one another. Service meshes are one of the most recent ways we use to deploy service-oriented and microservices systems. And they can be extremely helpful from many points of view, including making your apps easier to monitor and investigate issues they face. The service mesh tool I use and I like the most is **Istio** (<https://istio.io>); I recommend you read more about this tool in the book *Istio in Action* by Christian E. Posta and Rinor Maloku (Manning, 2022) for more details about this subject.

I'll show you a top-of-the-mountain picture of how a service mesh works, and then we'll discuss a couple of ways in which you will find them helpful when investigating scenarios of apps' executions:

1. **Fault injection** – a way in which you can force the app communication to fail to create a specific scenario you need to investigate.
2. **Mirroring** – a way to replicate events from a production application to investigate them easier in a testing environment.

Figure 12.12 visually shows you three services deployed in a service mesh. Each service is accompanied by an app that intercepts the data the app exchanges with other apps.

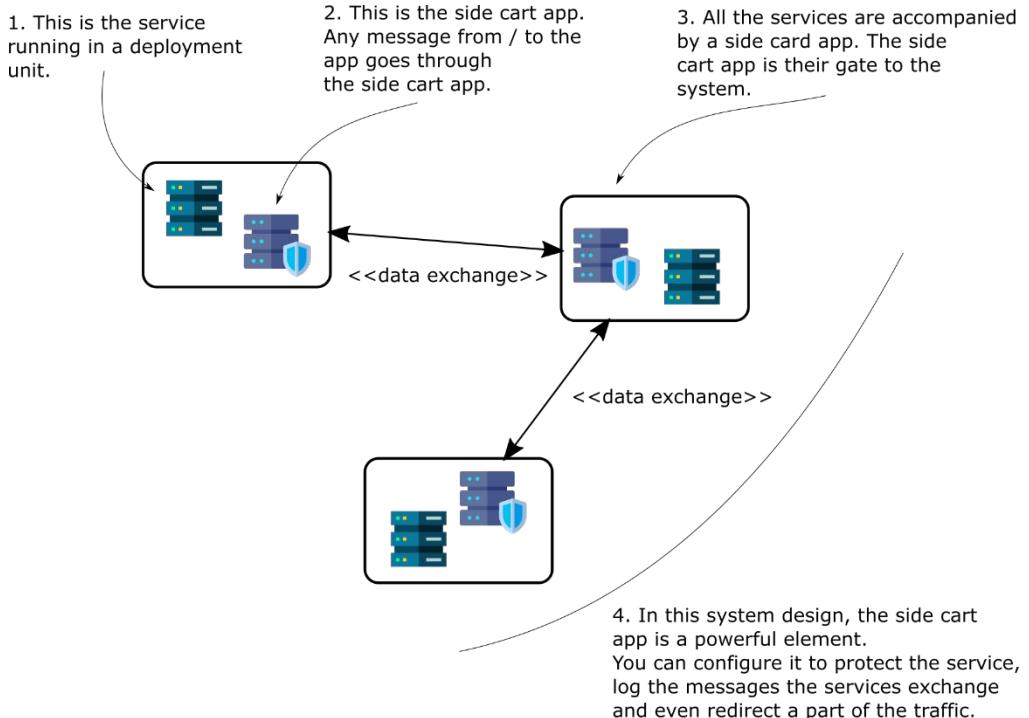


Figure 12.12 In a service mesh deployment, communication from and to each app is intercepted by a side cart app (a separate application). Since the side cart app intercepts the data exchanged, you can configure it to log details you need and even alter the communication to force the system into scenarios you want to investigate.

Because the side cart app intercepts the communication between the service it's linked to and any other app, you can configure the side cart app to manage that data in a way that is completely transparent to the service. Some of the ways you can configure the side cart app are useful for making some investigation scenarios easier, and we'll discuss these ways in sections 12.3.1 and 12.3.2.

12.3.1 Using fault injection to mimic hard to replicate issues

Some of the most challenging scenarios to investigate are the ones that are hard to replicate in your local environment or in an environment where you have more access to debug or profile. And also, in my experience, the environment creates some of the most difficult to replicate scenarios. Events such as the following can give you terrible headaches and make investigating some behavior challenging:

- Some faulty device causes network failures.
- Some additional software running where your app is installed makes the whole environment faulty or unstable.

However, there's something important to remember about such issues: your app should expect they might happen. The network is never 100% reliable and you can't trust 100% completely the environment. If your app fails because of a network spike, then your app isn't reliable enough: don't try to sweep the problem under someone else's rug – solve it!

You need to design the apps to be robust and to expect and know how to act upon an external event that doesn't allow them to execute a normal flow. But designing a system in such a way is no easy job. So one thing you should expect as a developer is that even if you made great efforts to cover all the cases, there're still problems that can occur. You need to be ready to investigate where these problems come from and implement solutions to solve them.

I know I mentioned multiple times throughout the book that the best way to investigate a problem is to find a way to replicate it, but I repeat this detail again since it's something I wish you to remember. Issues caused by the environment are sometimes difficult to replicate, but some scenarios can easily be recreated when you use a service mesh with your deployment.



NOTE The best way to investigate a scenario is by replicating the app's or system's behavior first in a test environment.

One of the easiest but very useful things to do is simulate a faulty communication scenario. In a service-oriented or microservices system, the whole system behavior relies on the way the apps communicate one with another. For this reason, it is extremely important to be able to test what happens when a certain service in the system can't be accessed. You need to simulate faulty behavior either for testing or for investigation purposes.

Since with a service mesh the communication from and towards an app is managed by the side cart app, you can configure the side cart app to act abnormally on purpose to simulate faulty communication between the two apps (figure 12.13). This way, you can investigate how the system behaves in such a case.

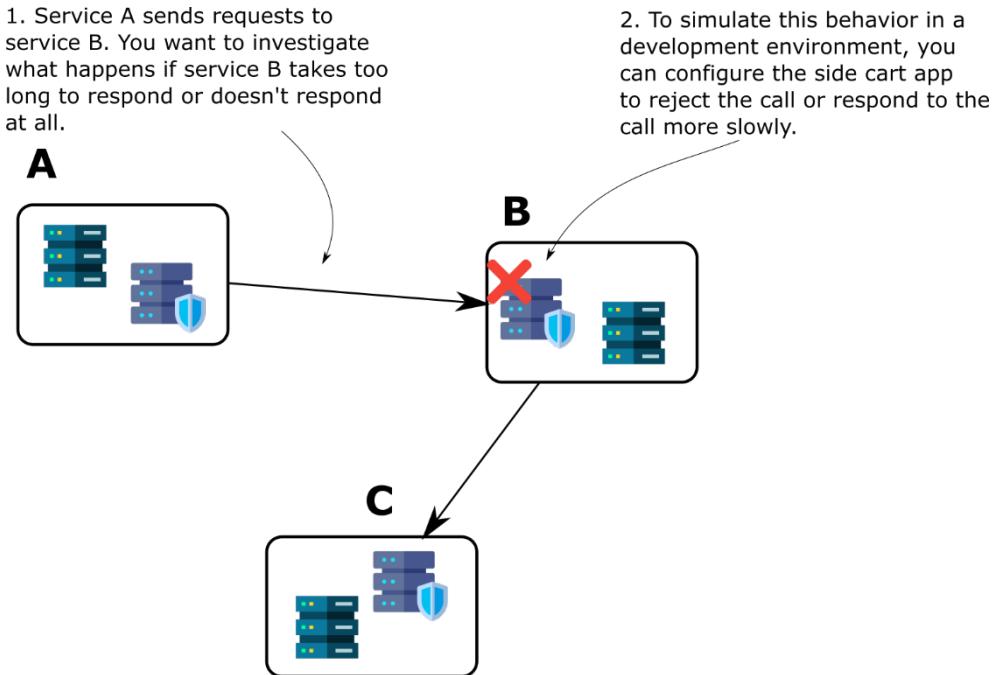


Figure 12.13 You can use the side cart container of a service mesh to force the system into scenarios you want to investigate. Say you want to replicate in a development environment a case happening in production where the communication is often disrupted between two services. You can easily configure a service mesh side cart container to force the execution into such a scenario to allow you to investigate what happens in such a case.



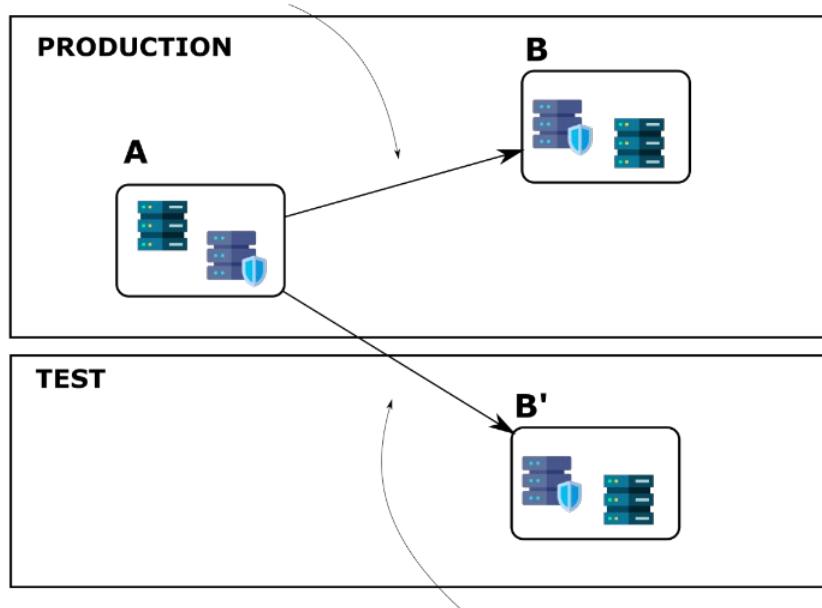
Fault injection means breaking your system on purpose in a test environment to replicate specific behavior that is difficult to replicate otherwise.

12.3.2 Using mirroring to facilitate testing and error detection

When using service meshes, one technique you can use to easily replicate a problem in a different environment is mirroring. Mirroring means configuring the side cart app to send a copy of the same requests the service sends to a replica of the app it communicates with. This replica may run in a different environment than you use for testing (figure 12.14). Now,

you can use the app running in the test environment to debug or profile the communication between services.

1. You investigate a problem in production and suspect that the root cause is somewhere in the communication between services A and B.



2. Using mirroring, you deliver a copy of each request to a replica of the B service in a test environment. Now, you have full access and can debug or log without the risk of affecting the production environment.

Figure 12.14 You can configure the side cart container to mirror the events from a production app to a service deployed in development. This way, you can investigate a problem from production which is hard to replicate in development without interfering with the production environment.

Mirroring is a really useful investigation tool that you can use. But remember that even if your system uses a service mesh for deployment, it's still possible you won't be allowed to use mirroring. In many systems, the data used in the production environment is private and can't be simply copied to a test environment. If your system doesn't allow copying data from production to test, then mirroring will be prohibited as well.

12.4 Summary

- Systems today are in many cases composed of many services communicating with one another. Faulty communication between services can cause issues such as bad performance or even wrong output. It's essential to know how to investigate the communication between services using tools such as a profiling tool or taking benefit of deployment tools such as service meshes.
- You can use JProfiler to intercept the HTTP requests a server app receives and the event duration. You can use these details to observe if a given endpoint is called too many times or takes too long to execute causing stress on the app instance.
- You can use JProfiler to observe also the behavior of an app as an HTTP client. You can intercept all the requests an app sends and details about the requests such as duration, HTTP response status code, and encountered exceptions. These details give you the needed information to figure out if something is wrong with how the app integrates with other services.
- JProfiler gives you excellent tools to observe low-level communication established by an app by investigating directly the socket events. Investigating socket events you isolate the problem and make sure the issues are related either to the communication channel or some part of your application.
- With large, service-oriented systems, using a log monitoring tool is an excellent way to help you easier observe issues and put puzzle pieces together faster to find the problems' root causes. A log monitoring tool is software that collects exceptional events happening with each app in the system and displays the details you need to understand what the problem is and where it comes from. Sentry is an excellent tool you can use for system log monitoring.
- In some cases, you can take advantage of tools used to deploy the apps. For example, if your service deployments rely on a service mesh, you can use the service mesh capabilities to easily reproduce scenarios you want to investigate. You can configure:
 - Fault injection to simulate a service doesn't work properly and investigate how other services are affected in this case.
 - Mirroring to get a copy of all the requests an app sends to a replica of the receiver service. This replica is installed in a test environment where you can investigate a scenario using debugging and profiling techniques without affecting the production system.

A

Tools

In this appendix, you find all the recommended tools you need to install if you want to apply all the examples discussed throughout the book.

A.1 IDEs

To open and execute the projects provided with the book, you need to install an IDE. For the examples in the book, I used IntelliJ IDEA. You can download IntelliJ IDEA Community from here: <https://www.jetbrains.com/idea/download/>

Alternatively, you could use Eclipse IDE, which you can get here: <https://www.eclipse.org/downloads/>

Or Apache Netbeans, which you can get from here: <https://netbeans.apache.org/download/index.html>

A.2 JDK

To run the Java projects provided with the book, you need to install JDK 17 or a higher version. I recommend using the OpenJDK distribution, which you can download here: <https://jdk.java.net/17/>

A.3 Profilers

To discuss profiling techniques and reading heap dumps and thread dumps, we'll use VisualVM. You can download VisualVM here: <https://visualvm.github.io/download.html>.

A.4 Other tools

Throughout the book, we'll use Postman to call endpoints to demonstrate investigation techniques. You can download Postman at the following link <https://www.postman.com/downloads/>

B

Opening a project

In this appendix, you find the steps for opening and running an existing project. The projects provided with the book are Java apps using Java 17 version. We use these projects to demonstrate the use of several techniques and tools.

First, you need to have installed an IDE such as IntelliJ IDEA, Eclipse, or Apache Netbeans. For the examples, I used IntelliJ IDEA. If you want to use IntelliJ IDEA also, you can easily download and install IntelliJ IDEA Community from the official web page: <https://www.jetbrains.com/idea/download/>

To run the projects provided with the book, you need to install JDK 17 or a higher version. You can use any Java distribution. I use the OpenJDK distribution, which you can download from this page: <https://jdk.java.net/17/>.

Figure B.1 shows you how to open an existing project in IntelliJ IDEA. To select the project you want to open, choose **File > Open**.

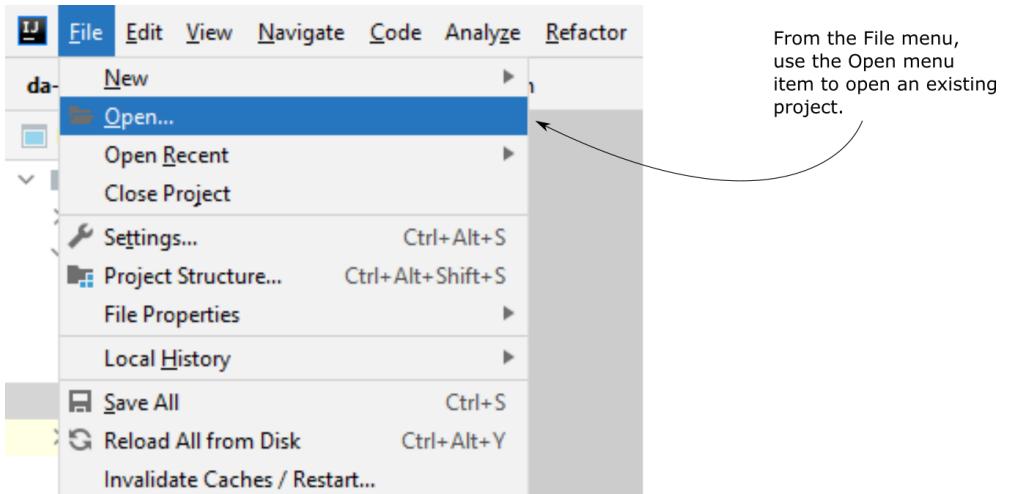


Figure B.1 To open an existing project in IntelliJ IDEA, use the Open menu item in the File menu.

Click **File > Open**, and a popup window appears. Select the project you want to open. Figure B.2 shows this popup window where you have to select the project from the file system.

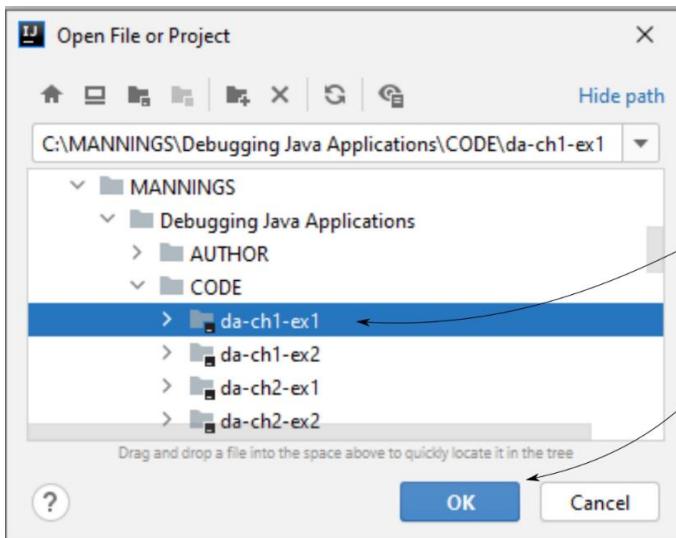
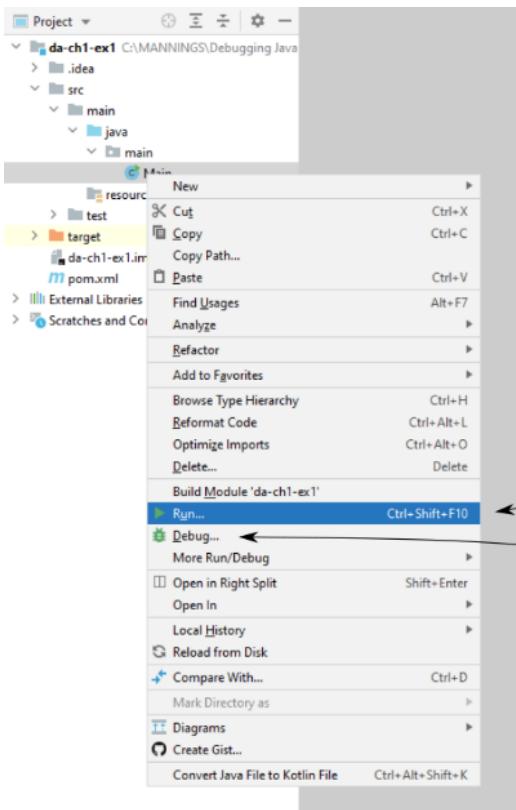


Figure B.2 After clicking on the Open menu item in the File menu, a popup window appears. In this window, select the project you want to open from the file system and click on the OK button.

To run the application, right-click on the class containing the `main()` method. For the projects provided with the book, the `main()` method is defined in a class named `Main`. Right-click on this class as presented in figure B.3 and select Run.

If you want to run the app with a debugger: **Right-click on the Main class > Debug.**



To run an app once you opened the project, right click on the Main class, and then click Run in the context menu.

To run an app with the debugger, click on the Debug menu item in the context menu.

Figure B.3 Once you opened an app, you can run it. To run the app, right click on the Main class and the select the Run menu item. If you want to run the app with a debugger, click on the Debug menu item.

C

Recommended further study

This appendix recommends some books related to this book's subject that you could find useful and interesting to read.

1. ***The Programmer's Brain by Felienne Hermans (Manning, 2021)*** explores how a developer's brain works when they investigate code. Reading the code is part of understanding software, and it's something we do before applying investigation techniques. Understanding these aspects better will also help you become better at investigating code.
2. ***Monolith to Microservices by Sam Newman (O'Reilly Media, 2019)*** is one of the recommendations I made in chapter 12 about continuing to study microservices as an architectural style. This book focuses especially on the difference between a monolithic approach vs. microservices and where and how to use each of the two architectural styles.
3. ***Building Microservices: Designing Fine-Grained Systems, 2nd Edition by Sam Newman (O'Reilly Media, 2021)*** is another book by Sam Newman focusing on designing systems composed of fine-grained services. The author analyses the pros and cons of the presented techniques with clear and detailed examples.
4. ***Microservices patterns by Chris Richardson (Manning, 2018)*** is one of the books I consider a must for anyone working with microservices architectures. The author details with clear examples the most essential techniques used in large-scale microservices and service-oriented systems.

5. **Five Lines of Code by Christian Clausen (Manning, 2021)** teaches you clean coding practices. Many apps today are unclean and challenging to understand. I designed many of the code listings you find throughout the examples of the book to be realistic, so they don't always follow clean coding principles. But what you shall always do after you understand how a piece of unclean code works is refactor to make it easier to understand. Developers call this principle the "Boy scout rule". In many cases, debugging is followed by refactoring to make code easier to understand in the future.
6. **Good Code, Bad Code by Tom Long (Manning, 2021)** is also an excellent book that teaches writing high-quality code principles. Along with Five Lines of Code of Christian Clausen, I also recommend you read Good Code, Bad Code to upskill in refactoring and writing easier-to-understand apps.
7. **Software Mistakes and Tradeoffs (Manning, 2022)** discusses with excellent examples how to take difficult decisions, make compromises, and optimize decisions in software development.
8. **Refactoring: Improving the Design of Existing Code by Martin Fowler with Kent Beck (Addison-Wesley Professional, 2018)** is another must-read for any software developer willing to improve their skills in designing and building clean and maintainable applications.

D

Threads

In this appendix, we'll discuss the basics of threads in a Java app. A thread is an independent sequential set of instructions your app runs. Operations on a given thread run concurrently with those on other threads. Any Java app today relies on having multiple threads. For this reason, it's almost impossible not to get into investigation scenarios where you'll have to understand more deeply why specific threads don't do what they have to do or don't "collaborate friendly" with other threads. That's why you'll find threads in several discussions throughout this book (especially chapters 7 to 9, but also here and there in the first half of the book where we discuss debugging). To properly understand these discussions, you need to know some basics about threads. This appendix teaches you those elements that are essential for understanding other discussions we have throughout the book.

We'll start with section D.1, where I'll remind you of the threads' big picture and why we use them in apps. We'll continue in section D.2 with more details on how a thread executes by discussing a thread's life cycle. Knowing the states of a thread life cycle and the possible transitions gives you an understanding of "what happens," and is mandatory for investigating any thread-related issue. In section D.3, we discuss threads synchronization, which is a way to control the executing threads. Faulty synchronization implementations introduce the majority of problems you'll need to investigate and solve. In section D.4, we discuss the most common thread-related issues.

Threads are a complex subject, so in this appendix, I'll only focus on the topics you need to know to understand the techniques presented in this book correctly. I can't promise to make you an expert in the subject in only a few pages. For this reason, at the end of the appendix, you'll find a few resources I recommend to you for getting this subject in depth.

D.1 What a thread is

In this section, we discuss what threads are and how using multiple threads helps an app. A thread is an independent timeline of operations in a running process. Any process can have

multiple threads that run concurrently, enabling your app to solve multiple tasks, potentially, in parallel. Threads are an essential component of how a language handles concurrency.

I like to visualize a multithreaded app as a group of timelines, as presented in figure D.1. Observe the app starts with one thread (the main thread). This thread launches other threads which can start others, and so on. Remember that each thread is independent of the others. For example, the main thread (the app started with) can end its execution long before the app itself. The process stops when all its threads stop.

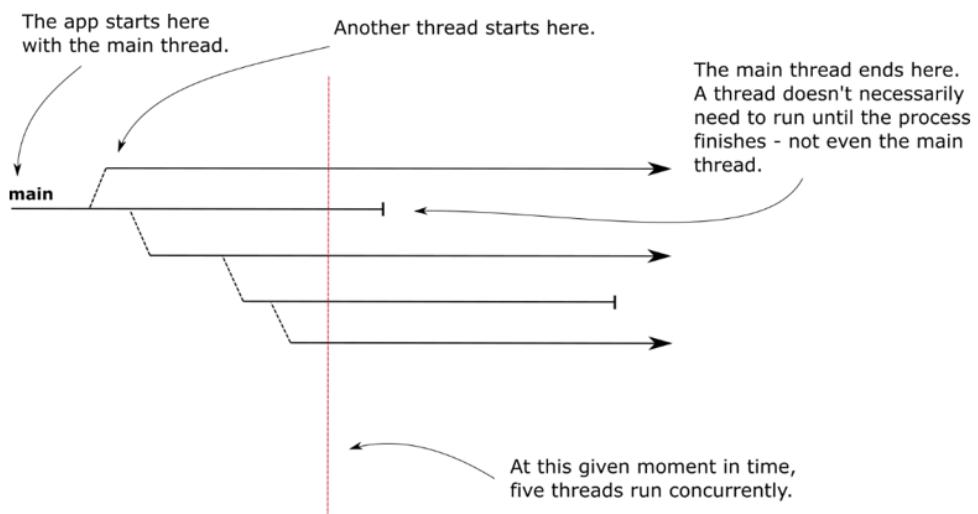
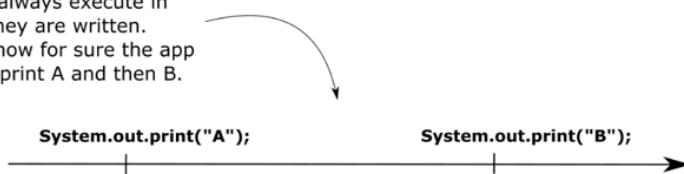


Figure D.1 A multithreaded app visualized as a group of timelines. Each arrow in the figure represents the timeline of a thread. An app starts with the main thread, which can further launch other threads. Some threads run until the process ends, while others stop earlier. At a given time, an app can have one or more threads running in parallel.

Instructions on a given thread are always in a defined order. You always know that A will happen before B if instruction A is before instruction B on the same thread. But since two threads are independent of one another, you can't say the same about two instructions A and B, each one on a separate thread. In such a case, either A can execute before B or the other way around (figure D.2). Sometimes, we can say that one case is more probable than another. But we can't conclude how one flow will consistently execute.

Two instructions on the same thread will always execute in the order they are written.
Here, we know for sure the app will always print A and then B.



Since every thread is independent of others, we can't say in which order two instructions on two different threads will execute.
In this case, the app could print both AB or BA.

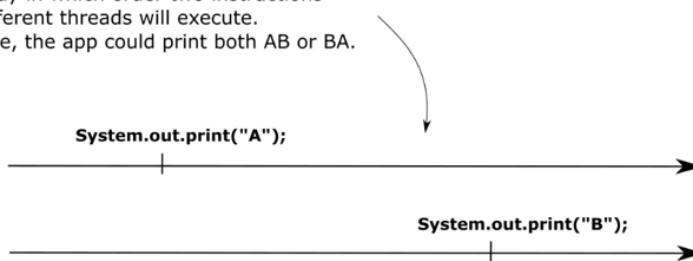


Figure D.2. With two instructions on one thread, we can always say which is the exact order of execution. But because two threads are independent, if one instruction is on one thread and one on another, we can't precisely say which order they will execute. At most, we can sometimes say that a scenario is more likely than the other.

In many cases, you'll see threads visually represented by tools as timelines. Figure D.3 shows the way VisualVM (a profiler tool we use throughout the book) presents the threads as timelines.

Threads seen as timelines
in VisualVM.

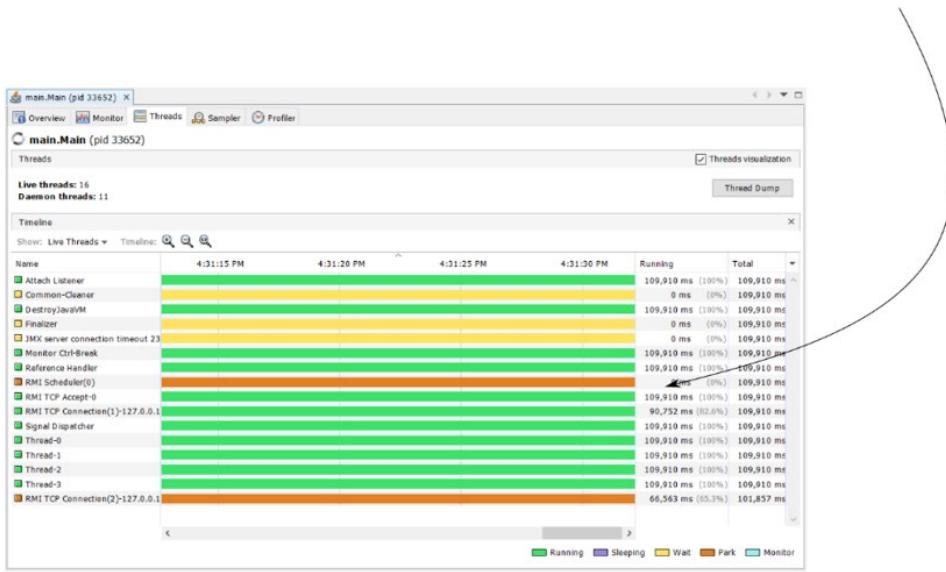


Figure D.3. VisualVM shows threads as timelines. This visual representation makes the app's execution easier to understand and helps you investigate possible problems.

D.2 A thread's life cycle

Once you visualize the threads, another essential aspect in understanding their execution is knowing the thread life cycle. Throughout its execution, a thread goes through multiple states (figure D.4). When using a profiler (as discussed in chapters 6 through 9) or a thread dump (as discussed in chapter 10), we'll often refer to the thread's state, which is essential for figuring out the execution. Knowing how a thread can transition from one state to another and how the thread behaves in each state is essential to following and investigating the app's behavior.

Figure D.4 visually presents the thread states and how a thread can transition from a state to another. We can identify the following main states for a Java thread:

- **New** – the thread is in this state right after its instantiation (before being started). While in this state, the thread is a simple Java object. The app can't yet execute the instructions it defines.
- **Runnable** – the thread is in this state after its `start()` method has been called. In this state, the JVM can execute the instructions the thread defines. While in this state, the JVM will progressively move the thread between two substates:

- Ready – the thread doesn't execute, but the JVM can anytime put it in execution.
- Running – the thread is in execution. A CPU currently executes instructions it defines.
- **Blocked** – The thread had been started but it was temporarily taken out of the runnable state, so the JVM can't execute its instructions. This state helps us control the thread execution, allowing us to temporarily "hide" the thread from the JVM so it can't execute it. While blocked, a thread can be in the following sub-states:
 - Monitored – The thread is paused by a monitor of a synchronized block (object controlling the access to a synchronized block) and waits to be released to execute that block.
 - Waiting – During the execution, a monitor's `wait()` method was called, which caused the current thread to be paused. The thread remains blocked until the `notify()` or `notifyAll()` methods are called to allow the JVM to release the thread in execution.
 - Sleeping – The `sleep()` method in the `Thread` class was called, which paused the current thread for a defined time. The time is given as a parameter to the `sleep()` method. The thread becomes runnable after this time passes.
 - Parked – almost the same as waiting, a thread will show parked after someone called the `park()` method, which blocks the current thread until the `unpark()` method is called.
- **Dead** – A thread is dead or terminated after it either finished its set of instructions, an `Error` or `Exception` halted it, or it was interrupted by another thread. Once dead, a thread cannot be started again.

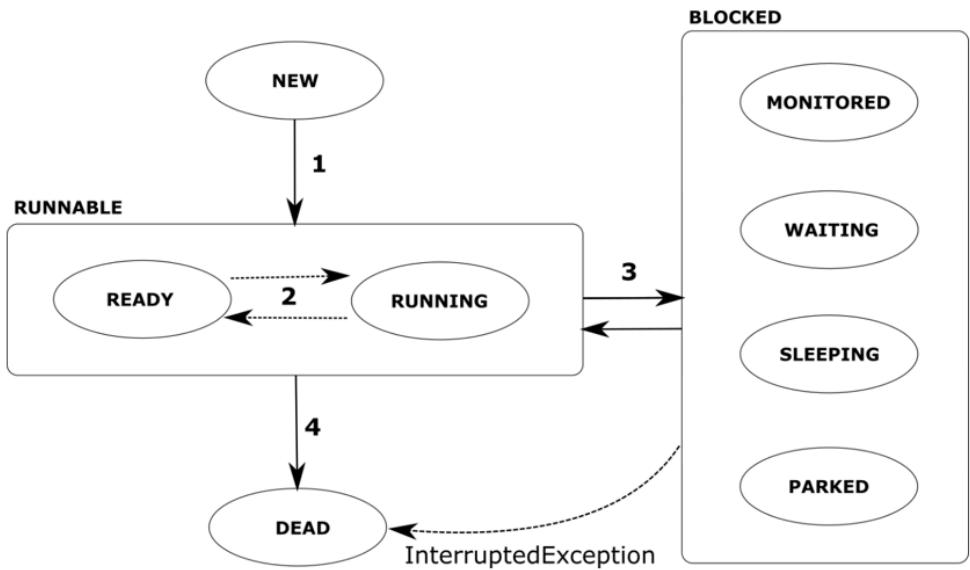


Figure D.4. A thread life cycle. During its life, a thread goes through multiple states. First, the thread is new, and the JVM cannot run the instructions it defines. After starting the thread, it becomes runnable and starts to be managed by the JVM. The thread can be temporarily blocked during its life, and at the end of its life, it goes to a dead state from which it can't be restarted.

Figure D.4 also shows the possible transitions between thread states:

1. The thread goes from new to runnable once someone calls its `start()` method.
2. Once in the runnable state, the thread oscillates between ready and running. The JVM decides which thread is executed and when.
3. Sometimes (but not mandatory), the thread might get blocked. The thread goes into the blocked state in several ways. Some examples are:
 - o The `sleep()` method in Thread class has been called putting the current thread into a temporary blocked state.
 - o Someone called the `join()` method causing the current thread to wait after another one.
 - o Someone called the `wait()` method of a monitor, pausing the execution of the current thread until the `notify()` or `notifyAll()` methods are called.
 - o A monitor of a synchronized block pauses the execution of a thread until another active thread finishes the execution of the synchronized block.

4. The thread can go into the dead (terminated) state either when it finishes its execution or when another thread interrupts it. Transitioning from the blocked state to the dead state is considered illegal by the JVM. If a blocked thread is interrupted by another, the transition is signaled with an `InterruptedException`.

D.3 Synchronizing threads

In this section, we discuss approaches to synchronize threads that developers use in multithreaded apps. Developers use synchronization techniques to control the threads in a multithreaded architecture. But in most cases, the wrong use of synchronization is also the root cause of many problems you'll have to investigate and solve. We'll go through an overview of the most common ways used to synchronize threads and, this way, give you the big picture you need to understand the discussion when we refer to one or another from the book's chapters.

D.3.1 Synchronized blocks

The simplest way to synchronize threads, and usually the first concept any Java developer learns, is using a synchronized block of code. The purpose is to allow only one thread at a time through the synchronized code - to prohibit concurrent execution for a given piece of code. The language directly offers this option to synchronize the threads, which you find in two flavors:

- Block synchronization – applying the `synchronized` modifier on a given block of code.
- Method synchronization – applying the `synchronized` modifier on a method.

The next code snippet shows an example of a synchronized block.

```
synchronized (a) {    #A
    // do something    #B
}
```

#A The object "a" between the parenthesis is the monitor of the synchronized block.

#B The synchronized block of instructions is defined between the curly braces.

The next code snippet shows you a method synchronization.

```
synchronized void m() {    #A
    // do something    #B
}
```

#A Synchronized modifier applied to the method.

#B The whole block of code of the method defined between the curly braces is synchronized.

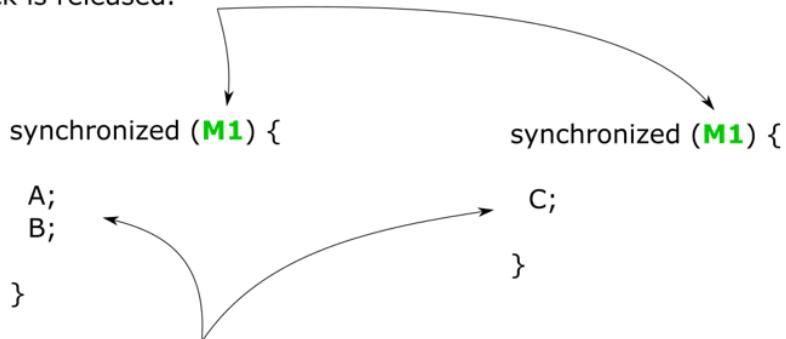
Both ways of using the `synchronized` keyword work the same way, even if they look a bit different. You'll find two important components of each synchronized block:

1. The monitor – an object managing the execution of the synchronized instructions.
2. The block of instructions – The actual instructions which are synchronized.

The method synchronization looks to be missing the monitor, but for this syntax, the monitor is actually implied. For a non-static method, instance "this" will be used as a monitor, while for a static method, the synchronized block will use the class's type instance.

The monitor (which cannot be null) is the object that gives sense to a synchronized block. This object decides if a thread can enter or not and execute the synchronized instructions. Technically, the rule is easy: once a thread enters the synchronized block we say it acquires a lock on the monitor. No other thread will be accepted in the synchronized block until the one who has the lock releases it. For the moment, to simplify things, let's assume that a thread only releases the lock when it exits the synchronized block. Figure D.5 shows a visual example. Imagine the two synchronized blocks are in different parts of the app. But because they both use the same monitor, M1 (the same object instance), a thread can execute in either one of the blocks at a time. Neither one of the instructions A, B, or C will be called concurrently (at least not from the presented synchronized blocks).

Both synchronized blocks use the same monitor. For this reason, when one thread acquires a lock on M1, no other thread can enter any of the two synchronized blocks until the lock is released.



A, B, and C are common code instructions.
Neither one of A, B, and C can be executed at the same time because only one thread will be active throughout the two synchronized blocks.

Figure D.5 An example of using synchronized blocks. Multiple synchronized blocks of the app can use the same object instance as a monitor. When this happens, all threads are correlated such that only one active thread executes in all. In this image, if one thread enters the synchronized block defining instructions A and B, no other thread can enter either in the same block or in the one defining instruction C.

However, the app may define multiple synchronized blocks. The monitor is the one that links multiple synchronized blocks. But when two synchronized blocks use two different

monitors (figure D.6), these blocks are not synchronized one with the other. In figure D.6, the first and the second synchronized blocks are also synchronized with each other since they use the same monitor. But these two blocks aren't also synchronized with the third. The result is that instruction D, defined in the third synchronized block, can execute concurrently with any of the instructions of the first two synchronized blocks.

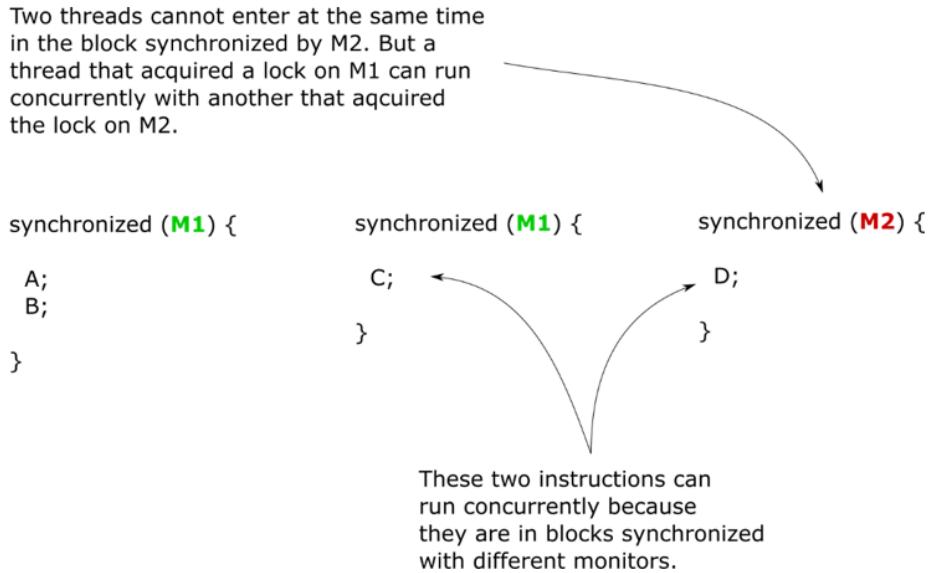


Figure D.6 When two synchronized blocks don't use the same object instance as monitor, they are not synchronized one with another. In this case, the second and the third synchronized blocks use different monitors. That means that instructions from these two synchronized blocks can execute simultaneously.

When investigating issues using tools such as a profiler or a thread dump, you'll need to recognize the way a thread has been blocked. This aspect might tell what happens, why, or what causes a given thread not to execute. Figure D.7 shows how VisualVM (a profiler we use throughout chapters 7 to 9) indicates the monitor of a synchronized block has blocked a thread.

In this example, you observe how VisualVM shows certain threads which are blocked by a monitor of a synchronized block of code. When investigating an app's behavior, knowing what this state means helps you understand the big picture of what executes and may reveal certain problems.

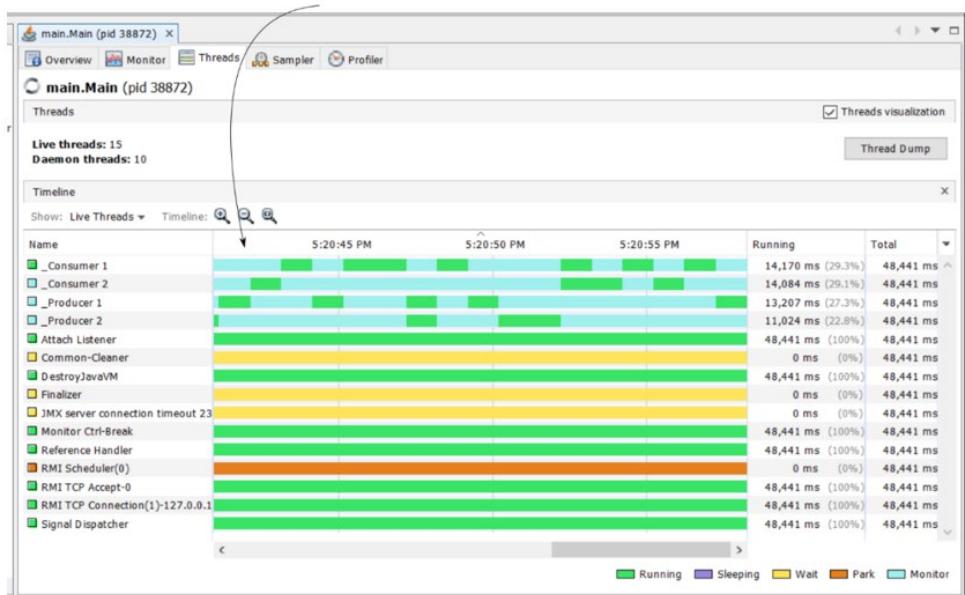


Figure D.7 VisualVM indicates the state of a thread. When using the threads tab of the profiler, you get a complete picture of what each thread does and, if a thread is blocked, what blocked that thread.

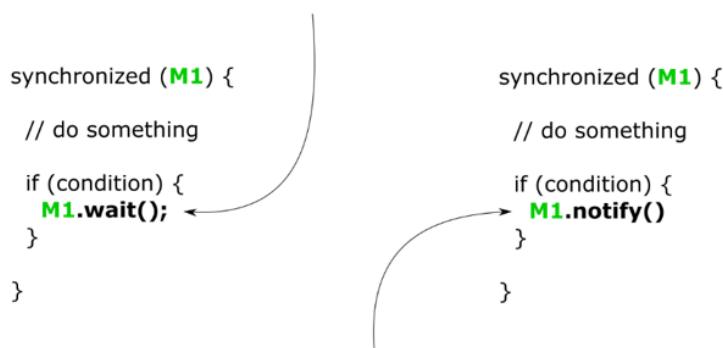
D.3.2 Using wait(), notify(), and notifyAll()

Another way a thread might be blocked is by asking it to wait for an undefined time. Using the `wait()` method of a monitor of a synchronized block, you can instruct a thread to wait an undefined time. Some other thread can then "tell" the one that's waiting to continue its work by using the `notify()` or `notifyAll()` methods of the monitor. Usually, these methods are used in several cases to improve the app's performance by preventing a thread from executing if it doesn't make sense to execute. At the same time, the wrong usage of these methods might lead to deadlocks or situations where threads wait indefinitely without being released to execution anymore.

Remember that `wait()`, `notify()` and `notifyAll()` make sense only when they are used in a synchronized block. These methods are behaviors of the synchronized block's monitor, so you can't use them without having a monitor. Using the `wait()` method, the monitor can block a thread for an undefined time. When blocking the thread, it also releases the lock it acquired so other threads can enter blocks synchronized by that monitor. When

the `notify()` method is called, the thread can again be executed. Figure D.8 summarizes the usage of `wait()` and `notify()`.

If for a given condition a thread should pause executing, you use monitor's `wait()` method to instruct the thread to wait. While waiting, the thread releases the lock on the monitor to allow other threads to enter the synchronized blocks.

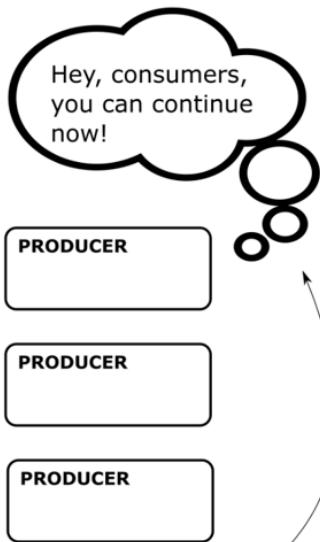
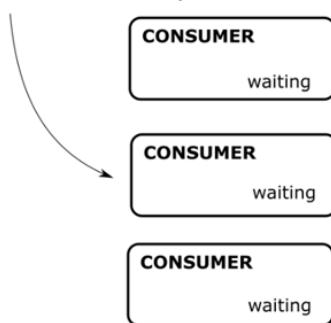


To allow the waiting thread continue its execution, you call the monitor's `notify()` or `notifyAll()` methods.

Figure D.8 In some cases, a thread should pause from executing and wait for something to happen. To make a thread wait, the monitor of a synchronized block can call its `wait()` behavior. When the thread becomes executable again, the monitor can call the `notify()` or `notifyAll()` methods.

Figure D.9 shows a more particular scenario. In chapter 7, we used as an example an app implementing a producer-consumer approach. In a producer-consumer approach, multiple threads share a resource. The producer threads add values to the shared resource, and the consumer threads consume values from the shared resource. But what happens if the shared resource no longer has values? The consumers would not benefit from executing at this time. Technically, they can execute, but they have no value to consume. So allowing the JVM to execute them would just cause unneeded resource consumption on the system. A better approach would be to "tell" the consumers to wait when the shared resource has no values and tell them to continue their execution only after a producer added a new value to the shared resource.

The three consumers are waiting because the list is empty so they don't have a value to consume. For them, it doesn't make sense to execute when the list is empty, so they better wait to save the system's resources.



When a producer adds an element to the list, it notifies the consumers using the `notify()` or `notifyAll()` methods. The consumers enter again the runnable state and the JVM can execute them.

Figure D.9 A use case for `wait()` and `notify()`. When a thread brings no value when executing because of the current conditions, then we can make it wait until further notice. In this case, a consumer should not execute where it has no value to consume. We can make the consumers wait, and a producer can tell them to continue only after new values are added to the shared resource.

D.3.3 Joining threads

A quite common thread synchronization approach is joining threads. This approach actually means to make a thread wait until another finished its execution. Different from the `wait/notify` pattern discussed below is that the thread doesn't wait to be notified. The thread simply waits in this case for the other one to finish its execution. Figure D.10 shows an example of a scenario that could benefit from such a synchronization technique.

Suppose you have to implement some data processing based on data retrieved from two different independent sources. Usually, fetching the data from the first data source takes about 5 seconds, and getting the data from the second data source takes about 8 seconds. If you execute the operations sequentially, the time needed to get all the data for processing is $5 + 8 = 13$ seconds. But you know a better approach. Since the data sources are two independent databases, you can get the data from both at the same time if you use two threads. But then, you need to make sure the thread that processes the data waits for both

that retrieve data to finish before it can start. To achieve this, you make the processing thread join the threads that retrieve the data (figure D.10).

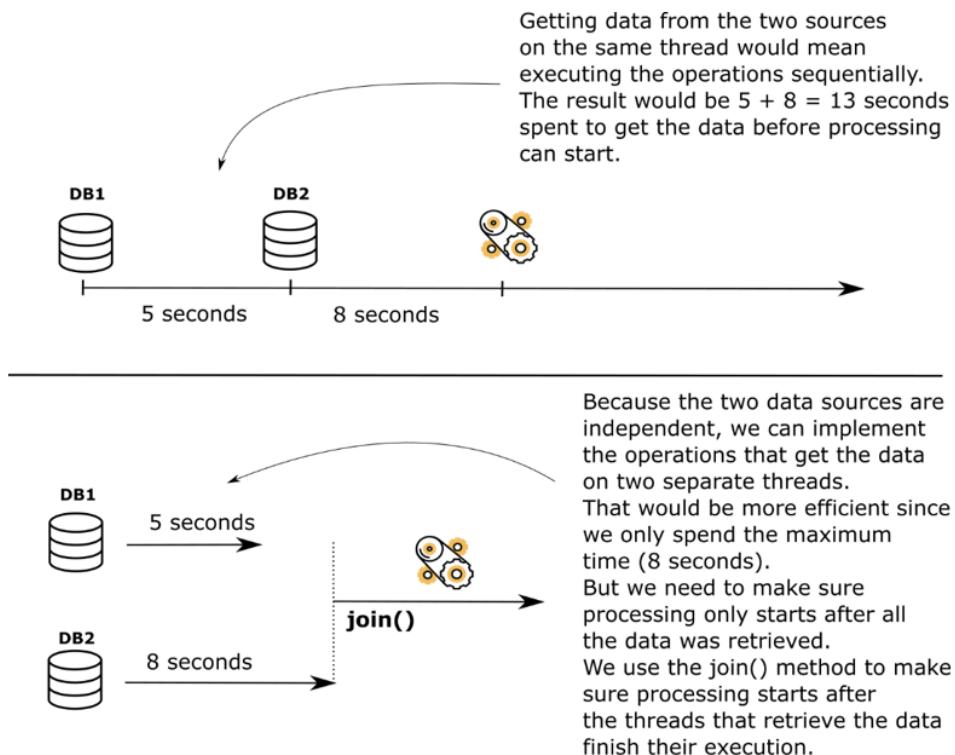


Figure D.10 In some cases, you can improve the app's performance using multiple threads. But you need to make some threads wait after others since they depend on the execution result of those threads. You can make a thread wait after another using a join operation.

Joining threads is, in many cases, a needed synchronization technique. But when not used well, it can also cause problems that you might need to investigate and solve. For example, if the thread is waiting for another stuck or, on the contrary, that never ends, the one joining it will never execute.

D.3.4 Blocking threads for a defined time

Sometimes a thread needs to wait only for a given amount of time. We say that the thread is in a “timed waiting” state when using such an operation. Also, profilers and thread dumps will show the thread in such a state either as “timed waiting” or “sleeping.”

The following operations are the most common to cause a thread to be timed waiting:

- `sleep()` – you can always use the static `sleep()` method in class `Thread` to make the

thread currently executing the code to wait for a fixed amount of time.

- `wait(long timeout)` – the `wait` method with a timeout parameter can be used the same as the `wait()` method without any parameter as discussed in section D.3.2. However, if you provide a parameter, the thread will wait for a maximum of the given time if not notified earlier.
- `join(long timeout)` – works the same as the `join()` method we discussed in section D.3.3, but waits for maximum the timeout given as a parameter.

A common antipattern I often find in apps is using `sleep()` to make a thread wait instead of the `wait()` method we discussed in section 4.3.2. Take as an example the producer-consumer architecture we discussed. You could use `sleep()` instead of `wait()`, but how long should a consumer sleep to ensure the producer has time to run and add values to the shared resource? We don't have an answer to this question. So, for example, making the thread sleep for 100 milliseconds (as shown in figure D.11) can be too long or too short. So in most cases, if you follow this approach, you end up not having the best performance we could ensure.

Sometimes timed waiting is wrongly used instead of waiting. While functionally this approach might work sometimes, it usually is a less performant implementation.

```
synchronized (M1) {
    // do something
    if (condition) {
        Thread.sleep(100);
    }
}
```

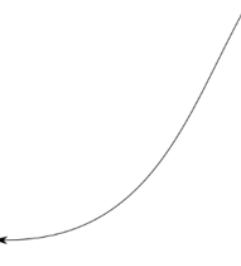


Figure D.11 Using a timed waiting approach instead of `wait()` and `notify()` is usually not the best approach. Whenever your code can determine when the thread can continue its execution, use `wait()` and `notify()` instead of `sleep()`.

D.3.5 Synchronizing threads with blocking objects

The JDK offers an impressive suite of tools for synchronizing threads. Out of these, a few of the most known classes used in multithreaded architectures are

- `Semaphore` – an object you can use to limit the number of threads that can execute a given block of code.
- `CyclicBarrier` – an object you can use to make sure at least a given number of threads are active to execute a given block of code.

- **Lock** – An object that provides more extensive synchronization options.
- **Latch** – An object you can use to make some threads wait until certain logic in other threads is performed.

These objects are higher-level implementations, each implementing a defined mechanism to simplify the implementation in certain scenarios. In most cases, I found these objects caused trouble because of the improper way they've been used – in many cases, developers having overengineered the code with them.

My advice is, of course, to use the simplest solution you can find to solve any problem and, before using any of these objects, make sure you properly understand how they work.

D.4 Common issues in multithreaded architectures

Investigating multithreaded architectures, you'll identify common problems which are root causes of various unexpected behavior scenarios (be it an unexpected output or a performance problem). Understanding these problems up-front will help you identify faster where a problem comes from and fix it. These issues are:

- **Race conditions** – two or more threads compete for changing a shared resource.
- **Deadlocks** – two or more threads stick, waiting after one another.
- **Livelocks** – two or more threads fail to meet the conditions to stop and continuously run without executing any useful work.
- **Starvation** – a thread is continuously blocked while the JVM executes other threads. The thread never gets to execute the instructions it defines.

D.4.1 Race conditions

Race conditions happen when multiple threads try to change the same resource concurrently. When something like this happens, we can either encounter unexpected results or exceptions. Generally, we use synchronization techniques to avoid such situations. Figure D.12 visually shows such a case. Threads T1 and T2 simultaneously try changing the value of variable x . Thread T1 tries to increment the value, while thread T2 tries to decrement it. This scenario may result in different outputs for repeated executions of the app. The following scenarios are possible:

- After the operations execute, x might be 5 – if T1 changed the value first, and T2 read the already changed value of the variable or the other way around, the variable will continue having value 5.
- After the operations execute, x might be 4 – if both threads read the value of x at the same time, but T2 wrote the value last, x will become 4 (the value T2 read, 5, minus 1)
- After the operations execute, x might be 6 – if both threads read the value of x at the same time, but T1 wrote the value last, x will become 6 (the value T1 read, 5, plus 1)

Such situations usually lead to unexpected output. With a multithreaded architecture where multiple execution flows are possible, such scenarios might be challenging to reproduce. Sometimes, they happen only in specific environments, which makes investigations difficult.

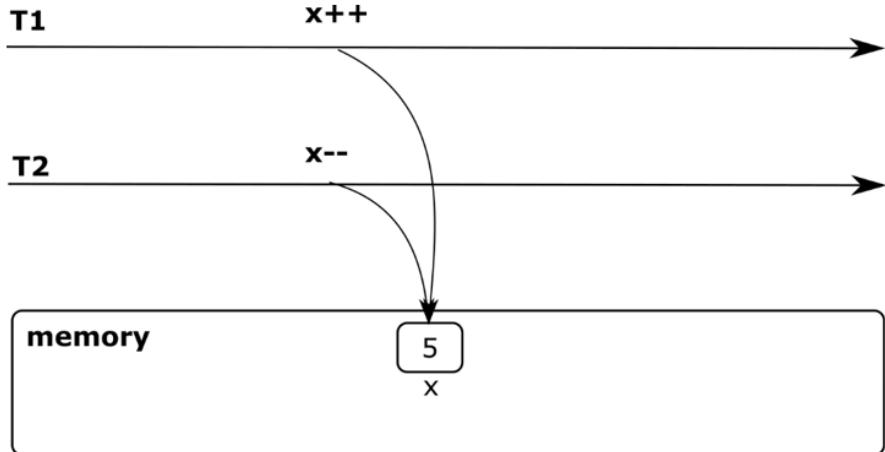


Figure D.12 A race condition. Multiple threads concurrently try to change a shared resource. In this example, threads T1 and T2 simultaneously try changing the value of variable x. This scenario can result in different outputs.

D.4.2 Deadlocks

Deadlocks are situations where two or more threads pause and then wait for something from one another to continue their execution (figure D.13). Deadlocks cause freezes for the whole app or at least a part, preventing certain capabilities from running.

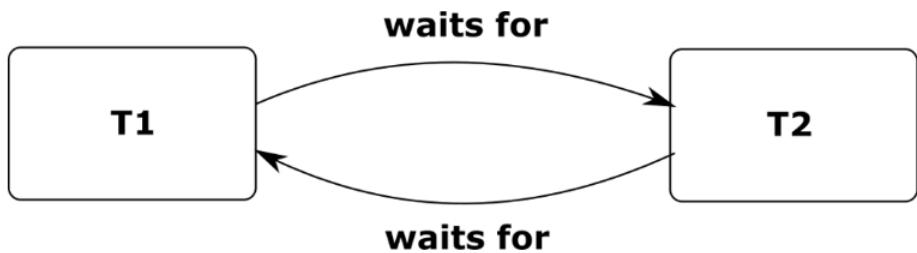


Figure D.13 Example of a deadlock. In a case where thread T1 waits for T2 to continue the execution and thread T2 waits for T1, we say the threads are in a deadlock. Neither can continue because they wait for the other.

Figure D.14 demonstrates the way a deadlock can occur with code. In this example, one thread acquired a lock on resource A, and another on resource B. But each thread also needs the resource acquired by the other thread to continue their execution. Thread T1 waits for thread T2 to release resource A, but at the same time, thread T2 waits for thread T2 to

release resource B. Neither one of the threads can continue since both waits for one another to release the resources they need. The threads are now in a deadlock.

Suppose a thread T1 entered the synchronized block by acquiring the resource B and now executes some instructions here.

```
synchronized (B) {
    // do something
    synchronized(A) {
        // do something
    }
}
```

At the same time, a thread T2 entered the synchronized block by acquiring the resource A and now executes some instructions here.

```
synchronized (A) {
    // do something
    synchronized(B) {
        // do something
    }
}
```

When reaching the nested synchronized block, none of the two threads can continue. T1 waits for T2 to release resource A. But to release resource A, T2 needs first to acquire resource B to enter the nested synchronized block. So T2 waits for T1 to release resource B. T1 waits for T2 and T2 waits for T1 to continue their executions. The threads are in a deadlock.

Figure D.14 A deadlock. Thread T1 can't enter the nested synchronized block because thread T2 has a lock on resource A. Thread T1 waits for T2 to release resource A so it can continue its execution. But thread T2 is in a similar situation. It cannot continue its execution because T1 acquired a lock on resource B. Thread T2 waits for thread T1 to release resource B to continue its execution. Since both threads wait for one another and neither one can continue its execution, we say the threads are in a deadlock.

The example presented in figure D.7 is simple, but it's just a didactic one. A real-world scenario is usually much more difficult to investigate and understand and can involve more than two threads. Beware that synchronized blocks are not the only way in which threads can remain stuck in a deadlock. The best way to understand such scenarios is using the investigation techniques you learn in chapters 7 through 9.

D.4.3 Livelocks

Livelocks are more or less the opposite of deadlocks. When threads are in a livelock, they should stop on a given condition, but the condition always changes in such a way that the threads continue their execution. The threads can't stop, and they continuously run, usually consuming reasonably the system's resources. Livelocks can cause performance issues in an app's execution, and it's one of the issues you'll potentially need to investigate and solve.

Figure D.15 describes a livelock with a sequence diagram. Two threads T1 and T2, run in a loop. To stop its execution, thread T1 makes a condition true before its last iteration. Next time T1 comes back to the condition, it expects it to be true and stop. However, this doesn't happen since another thread, T2, changed it back to false. Thread T2 finds itself in the same situation while running. Each thread changes the condition so they can stop, but at the same time, changing the condition make the other thread continue running.

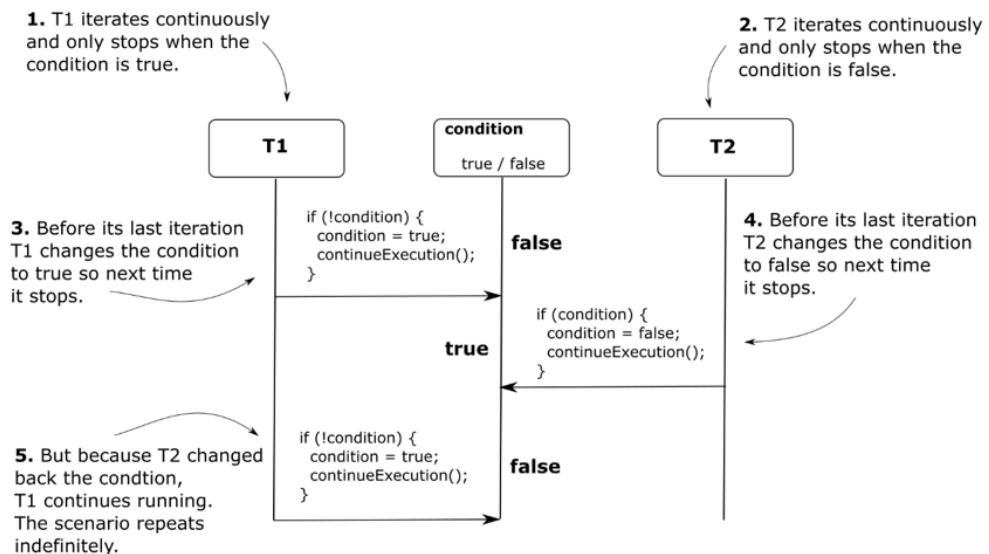


Figure D.15 An example of a livelock. Two threads rely on a condition to stop their execution. But when changing the value of the condition so they can stop, each thread makes the other one continue running. The threads cannot stop and spend reasonless the system's resources.

Same as in the case of the deadlock example in section 4.4.2, remember this is just a simplified scenario to help you understand straightforward what a livelock is. Livelocks can be caused by more complex scenarios in the real world, and more than two threads might be involved. Throughout chapters 7 to 9, you'll learn several ways in which you can approach the investigation of such scenarios.

D.4.4 Starvation

Another common problem but less likely to occur in today's apps is starvation. Even if it's less likely to find it, I wanted to mention it and make you aware of it if you find it in different other sources. Starvation is caused by a situation in a multithreaded architecture where a certain thread is constantly excluded from the execution even if it is runnable. The thread wants to execute its instructions, but the JVM continuously allows other threads to access the

system's resources. We say that the thread that cannot access the system's resources and execute its defined set of instructions is starving.

In the early JVM versions, such situations could occur when the developer sets a much lower priority to a given thread. Today, the JVM implementations are much smarter in treating these cases, so (at least in my experience) starvation scenarios are unlikely to happen anymore.

D.5 Further reading

As mentioned at the beginning of this appendix, threads are a complex and long topic. In this appendix, we discussed the essential topics that will help you properly understand the techniques discussed throughout this book. But, for any Java developer, understanding in detail how threads work is a valuable skill. Here is a list of resources I recommend you to grasp threads in depth.

1. OCP Oracle Certified Professional Java SE 11 Developer Complete Study Guide by Jeanne Boyarsky, Scott Selikoff (Sybex, 2020), chapter 18 describes threads and concurrency starting from zero and covering all the thread fundamentals OCP certification also requires. I recommend you starting with this book to learn threads.
2. The second edition of the Well-grounded Java developer by Benjamin Evans, Jason Clark, and Martijn Verburg (Manning, 2022) teaches in part 2 concurrency, starting from the fundamentals to performance tuning.
3. Java Concurrency in Practice by Brian Goetz et al. (Addison-Wesley, 2006) is an older book but didn't lose its value in many of the subjects it describes. I still recommend today this book as a must to read for any Java developer willing to upskill their threads and concurrency knowledge.

E

Memory of a Java app

In this appendix, we discuss how the Java Virtual Machine (JVM) manages the memory of a Java app. Some of the most challenging problems you'll have to investigate in Java apps are related to the way the app manages the memory. Fortunately, we can use several good techniques today to analyze such problems and find their root causes with a minimum time invested. But to benefit from those techniques, you first need to know at least some basics about how a Java app manages its memory.

An app memory is a limited resource. Even if today the systems can offer a large amount of memory for an app to use during its execution, we still need to be careful with how the app spends this resource. No system can offer a magical solution for unlimited memory (figure E.1). Memory issues lead to performance problems (the app becomes slow, it's more costly to deploy, starts more slowly, and so on) and sometimes can even lead to a complete stop of the process (such as in the case of an `OutOfMemoryError`).

We'll cover the essential aspects of memory management. In section E.1, we discuss how the JVM organizes the memory for an executing process. You'll learn about three ways of allocating the app's memory: the stack, the heap, and the metaspace. In section E.2, we discuss the stack, a memory space a thread uses to store locally declared variables and their data. Appendix D discusses threads and is a good read before this one if you don't know the basics related to concurrency in Java apps. Section E.3 discusses the heap and the way an app stores object instances in memory. We end our discussion in section E.4 with the metaspace, a memory location where the app stores the object types metadata.

However, be aware that the memory management of a Java app is a complex subject, and it could cover a full book of its own. In this appendix, I'll only present the details you immediately need to understand the discussions you'll find throughout the book concerning an app's memory management. But I can't promise to make you an expert in this area only in a few pages.

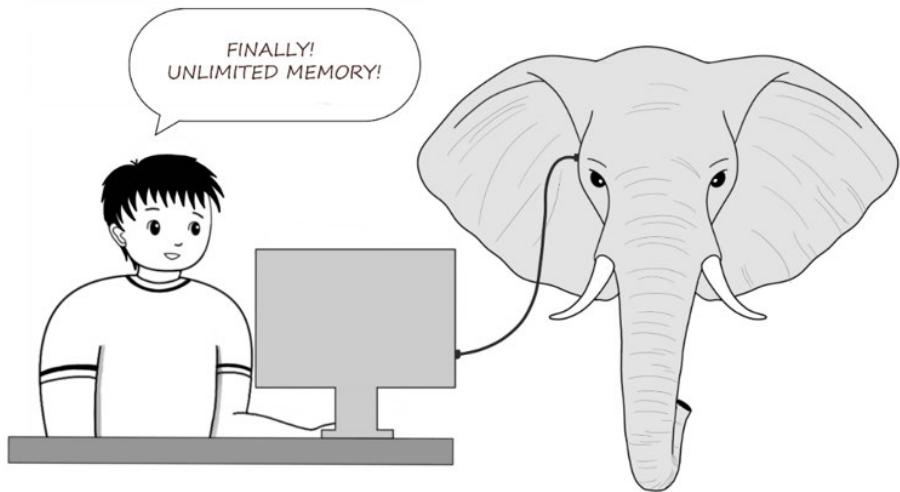


Figure E.1. An app's memory is a limited resource. There's no magical solution that allows us to allocate infinite memory to an app. When building apps, we need to treat memory consumption with consideration and avoid spending it for no reason. Apps may sometimes have memory issues. Suppose a certain capability uses too much memory. In that case, it may cause performance problems or even a complete failure, and you need to be ready to investigate and find the causes of such issues to be able to solve them properly.

E.1 How the JVM organizes an app's memory

In this section, we discuss how JVM organizes data in different memory locations, each of which it manages differently. Understanding how the JVM manages memory is essential for investigating issues that are related to memory. We'll use some visuals to discuss the main aspects related to memory management, and you'll learn which data goes where in a Java app's memory. Once we finish this big picture, we'll independently detail the memory management in each memory location in the following sections.

For the moment (to simplify the discussion), let's assume that a Java app has two ways to manage the data it stores during its execution: the stack and the heap. Depending on how the data is defined, the app will manage it either in the stack or the heap. But before discussing which data goes where, remember one essential detail: an app has more than one thread allowing it to process data concurrently. The heap is a singular memory location, and all the app's threads use it. However, each thread has its own memory location we call a stack. This is one of the first things I observed that can create confusion among developers when they first learn details about memory management. Take a look at figure E.2, which presents these details visually.

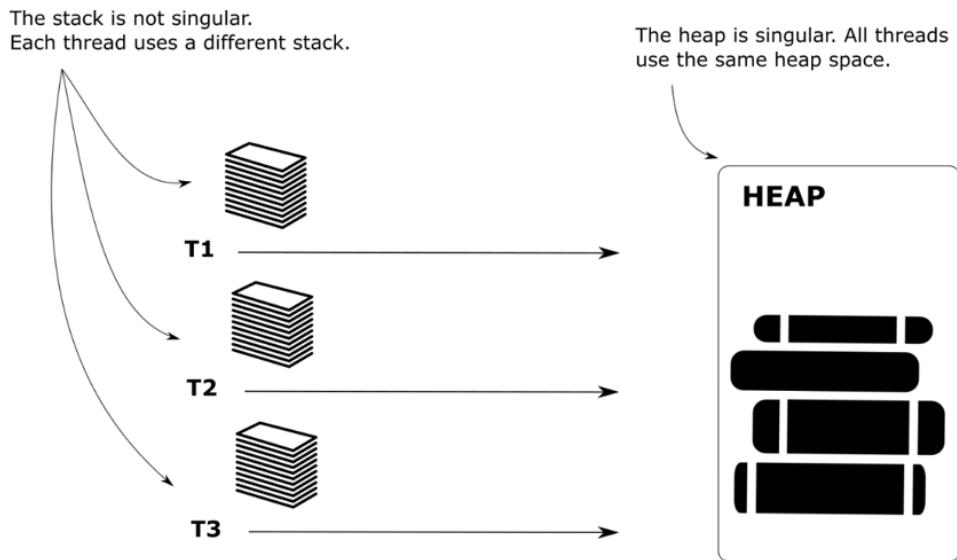


Figure E.2. T1, T2, and T3 are all threads of a Java app. All these threads use the same heap. The heap is a memory location where the app stores object instances data. However, each thread called its own memory location called a stack to store data locally declared.

The stack is a memory location owned by a thread. Each thread owns a particular stack, not shared with other threads. The thread stores any data declared locally in a block of code executed by that thread in this memory location. Say you have a method like the one presented in the next code snippet. The parameters `x` and `y` and the variable `sum` declared inside the method's code block are local variables. These values will be stored in the thread's stack when the method executes.

```
public int sum(int x, int y) {    #A
    int sum = x + y;    #A
    return sum;
}
```

#A Variables `x`, `y`, and `sum` will be stored in the stack.

The heap is a memory location where the app stores object instances data. Suppose your app declares a class `Cat` such as the one shown in the next code snippet. Any time you create an instance using the class's constructor – `new Cat()` – the instance goes to the heap.

```
public class Cat {
}
```

If the class declares instance attributes, the JVM stores these values in the heap, too. For example, if the `Cat` class would look like the one in the next code snippet, the JVM will store the name and age of each instance in the heap.

```
public class Cat {  
  
    private String name;    #A  
    private int age;       #A  
  
}
```

#A Object's attributes are stored in the heap.

Figure E.3 visually presents an example of data allocation. Observe that the locally declared variables and their values (`x`, and `c`) are stored in the thread's stack, while the `Cat` instance and its data go in the app's heap. A reference to the `Cat` instance will be stored in the thread's stack in variable `c`. Even the method's parameter that stores a reference to a `String` array will be part of the stack.

In this example, we consider the main thread. This thread starts its execution with the `main()` method. The variables declared locally in the `main()` method are stored in the main thread's stack. The values in the stack are variable `x` holding value 10 and variable `c` holding the reference of a `Cat` object.

The object instance and the values its attributes hold (if any) are stored in the heap.

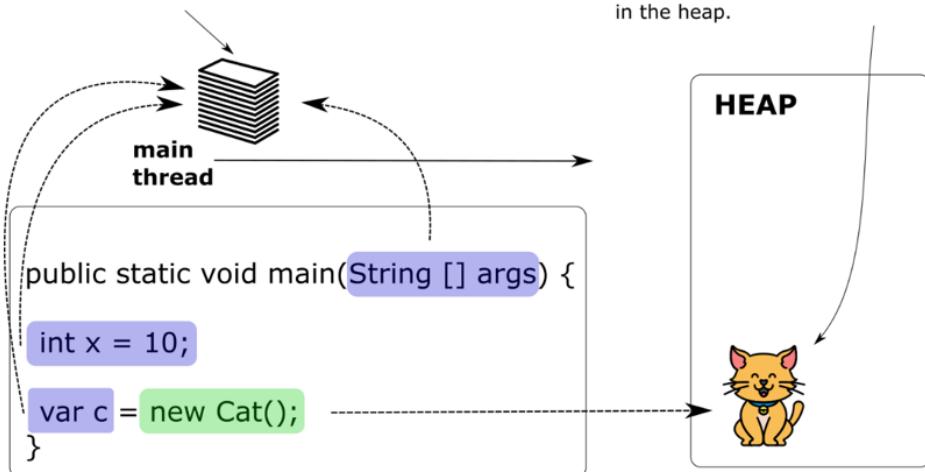


Figure E.3. The app reserves the locally declared variables in the threads's stack and the data defining an object instance in the heap. A variable in one thread's stack may refer to an object in the heap. In this example, variable `x` holding value 10 and variable `c` holding the reference to the `Cat` instance are part of the thread's stack.

E.2 The stack used by threads to store local data

In this section, we analyze in more depth the mechanics behind the stack. In section E.1, you learned that local values are stored in a stack and that each thread has its own stack location. Let's find out now how these values are stored and when the app removes them from memory. We'll use visuals to describe this process step by step with a short code example. Once we clarify the mechanics behind the stack memory management, we'll discuss what could go wrong and cause problems related to it.

First of all, why is this memory location called "a stack"? A thread's stack uses the principles of a stack data structure. A stack is an ordered collection where you can always remove only the latest added element. We usually visualize such a collection as a stack of layers, where each layer is stored one above another. You can only add a new layer on top of all the existing ones, and you can only remove the top layer. This method of adding and removing elements on which a stack relies is also named last-in-first-out (LIFO). Figure E.4. demonstrates how a stack works with a series of add and remove steps. To make the example simpler, I used numbers are the values in the stack.

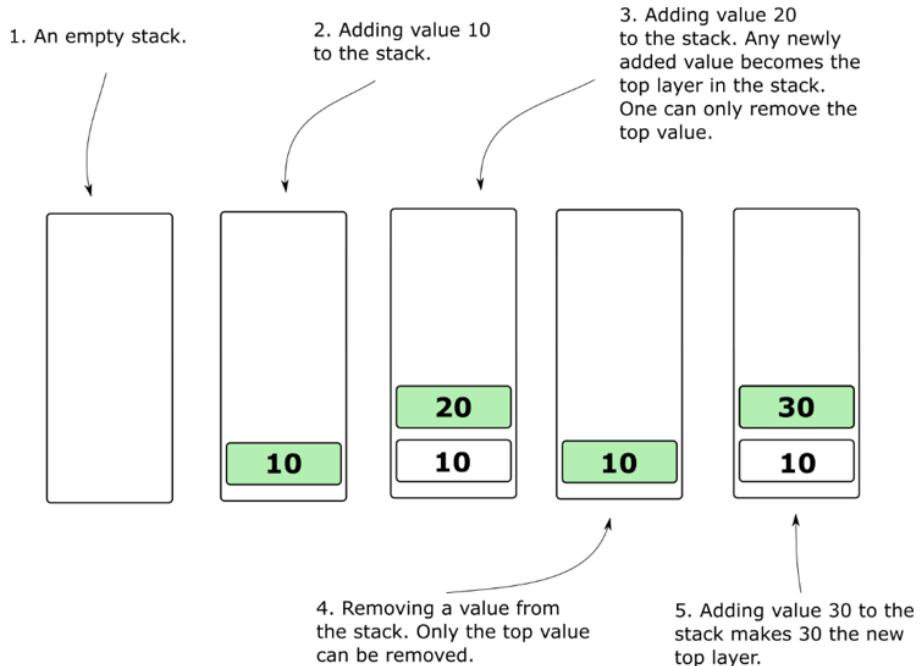


Figure E.4. Adding and removing values from a stack. The stack is an ordered collection working on the last-in-first-out principle. When you add a value to the stack it becomes the top layer in the stack. The top layer is the only one you can remove.

You will recognize the same behavior in how the app manages the data in a thread's stack. Whenever the execution reaches a start of a code block, it creates a new layer in the thread stack. Following a common stack principle, any new layer becomes the top layer and is the first to be removed. In figures E.5, E.6, E.7, and E.8, we follow the execution of a simple code snippet step by step and observe how the thread's stack changes:

```
public static void main(String [] args) {  
    int x = 10;  
    a();  
    b();  
}  
  
public static void a() {  
    int y = 20;  
}  
  
public static void b() {  
    int y = 30;  
}
```

The execution starts with the `main()` method (figure E.5). When the execution reaches the start of the `main()` method, the first layer is added to the thread's stack. This layer is a memory location where every local value declared in the code block will be stored. In this case, the code block declares a variable `x` and initializes the variable with value 10. This variable will be stored in this newly created layer of the thread's stack. This layer will be removed from the stack when the method ends its execution.

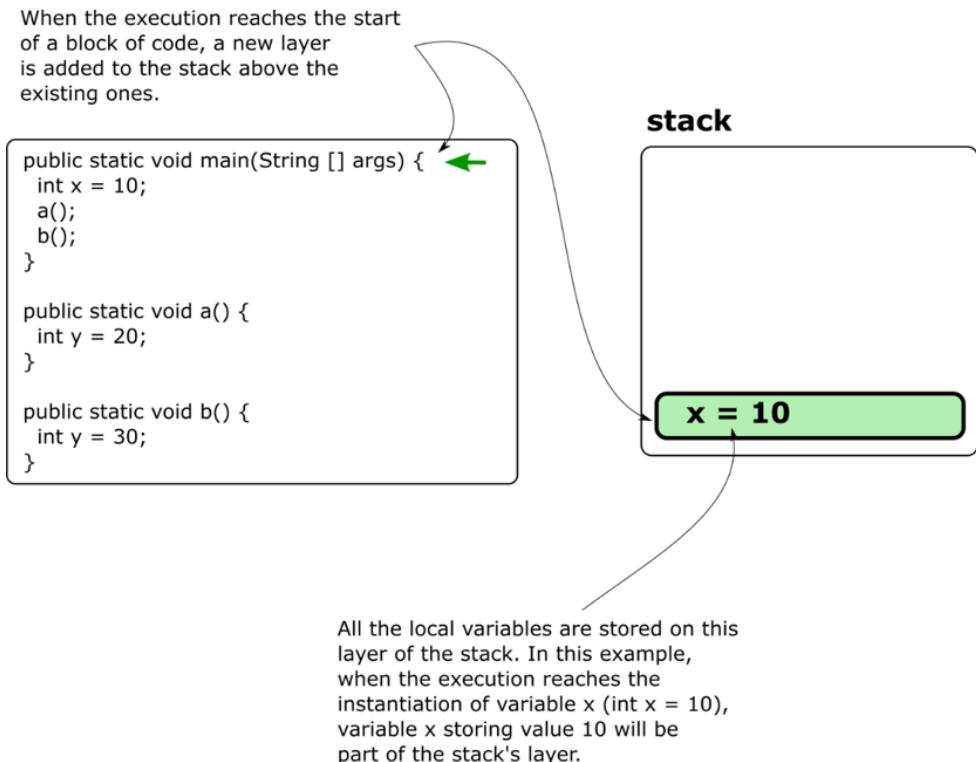


Figure E.5. When the execution reaches a start of a block of code, a new layer is created in the thread's stack. All the variables the block of code defines will be stored in this new layer. The layer is removed when the block of code ends. This way, we know that the values in this part of memory are released when they're not needed anymore.

A code block can call other code blocks. For example, in this case, method `main()` calls methods `a()` and `b()`. Methods `a()` and `b()` will work similarly. When the execution reaches the start of their block of code, a new layer is added to the stack. That new layer is the memory location where all the data declared local in that code block will be stored. Figure E.6 shows what happens when the execution reaches method `a()`.

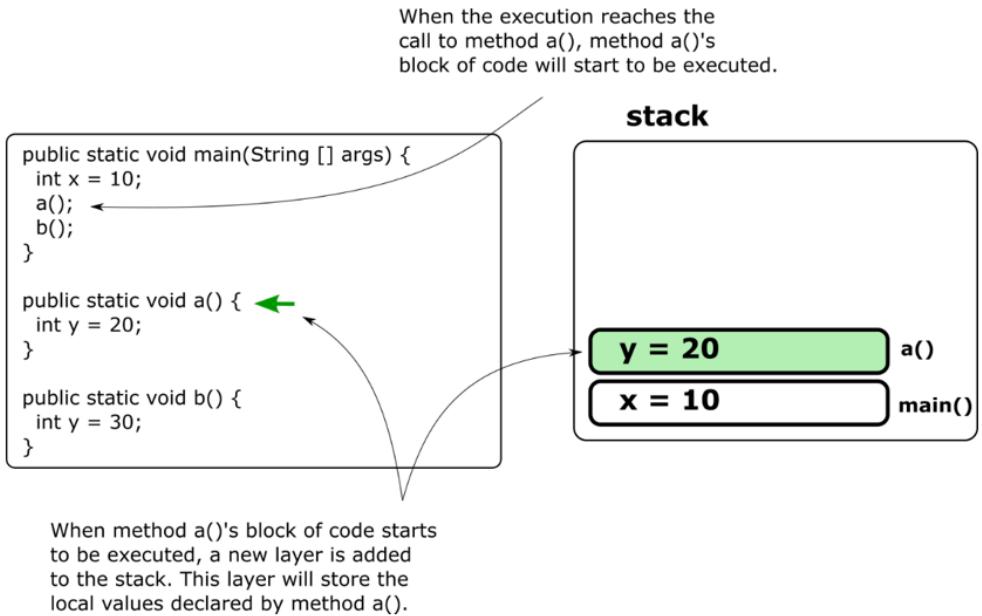


Figure E.6. Another block of code can be called from one in execution. In this case, method `main()` calls method `a()`. Since `main()` didn't finish yet, its layer is still part of the stack. Method `a()` creates its own layer where the local values it defines will be stored.

When method `a()` ends its execution and returns to `main()`, the layer reserved in the thread's stack is also removed (figure E.7) – meaning the data it stored is no longer in the memory. This way, the undeeded memory is deallocated to allow space for new data to be stored. A code block ends either when the execution reaches its last instruction, a `return` instruction, or throwing an exception. Observe that when a code block ends, its layer is always the top one in the stack fulfilling the LIFO principle.

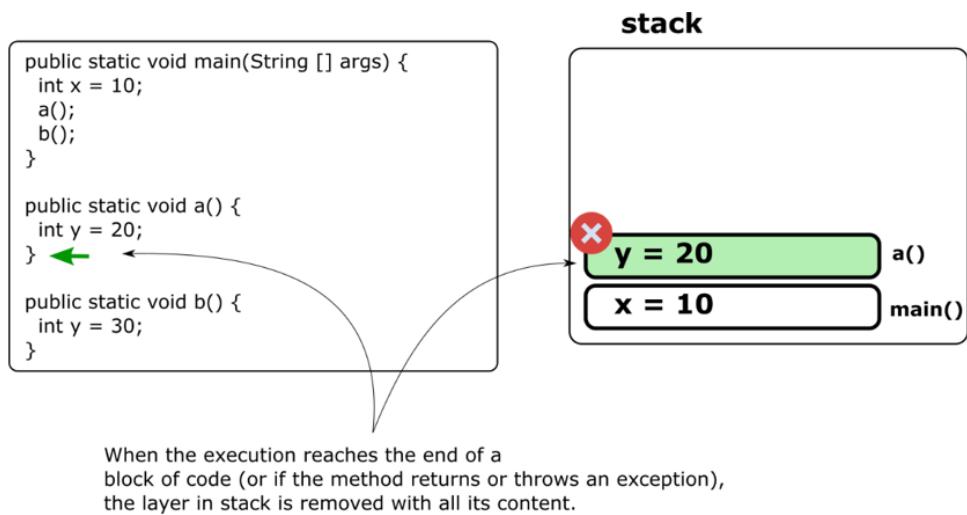
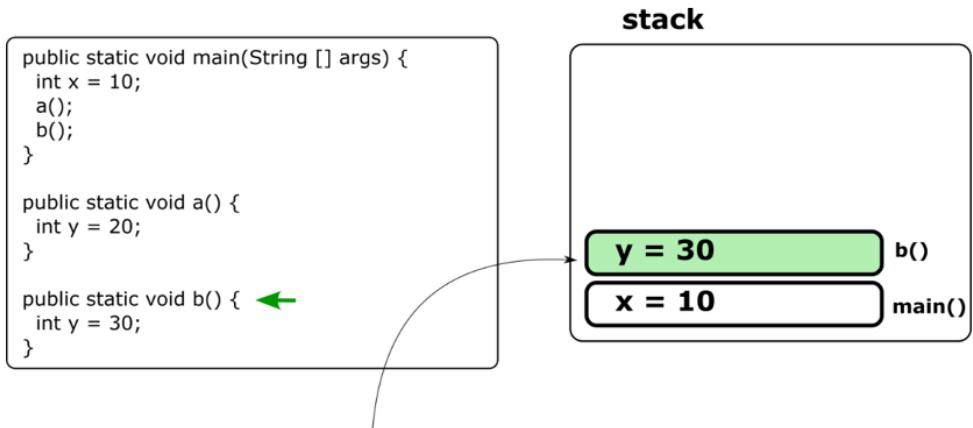


Figure E.7. When the execution reaches the end of a block of code, the stack layer opened for that block of code is removed together with all the data it contains. In this case, when method `a()` returns, its stack layer will be removed. This way, we make sure the unneeded data is removed from the memory.

Method `main()` continues its execution calling method `b()`. Same as method `a()` did, method `b()` reserves a new layer in the stack to store the local data it declares (figure E.8).



When the execution reaches method `b()`, method `a()`'s stack layer doesn't exist anymore. Method `b()` will create its own layer in the stack and store the local values it declares in it. When method `b()` ends its execution, its layer in the stack will also be removed. The same will happen for `main()`. In the end, when the thread ends its execution, the stack will be empty.

Figure E.8. Same as it happened for method `a()`, when method `b()` is called, and the execution reaches the start of its block of code, a new layer is added to the stack. The method can use this layer to store local data until the method returns, and the layer is removed.

When method `main()` finally reaches its end, the thread ends its execution, and the stack remains empty and is completely removed. At the same time, the thread goes into the dead state of its life cycle, as described in appendix D.

The stack has a default allocated. You can find the precise values depending on the JVM you use here: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html. This limit can also be adjusted, but you wouldn't be able to make it infinite anyway. A common issue with the stack is the `StackOverflowError`, which means a stack is filled completely, and no more layers can be added. When something like this happens, the code throws a `StackOverflowError`, and the thread whose stack got full stops completely. A recursion with a wrong stop condition usually causes such a problem. A recursion (or recursive implementation) is a method that calls itself until a given condition is fulfilled. If this condition is missing or allows the method to call itself too many times, the stack might get filled with the layers the method creates every time it begins its execution. Figure E.9 visually presents the stack created by an infinite recursion caused by two methods that call one another.

Because any beginning of a new block of code execution creates a new layer in the stack, any uncontrolled recursion can cause a stack overflow. A stack overflow is a situation where the stack fills and the app cannot allocate more layers to store the local values. In this example, method a() calls method b(), and method b() calls method a() without any condition for this cycle to stop at some point.

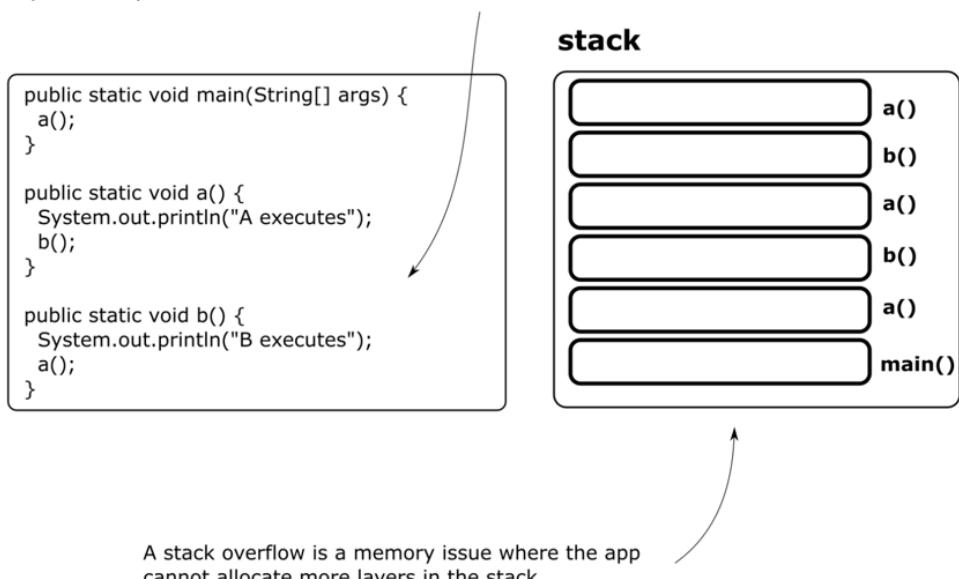


Figure E.9. Every new execution of a method creates a new layer in the stack. In case of a recursion, a method might fill the stack if it's called too many times. When the stack gets full, the app throws a `StackOverflowError`, and the current thread stops.

Since each thread has its own stack, a `StackOverflowError` only affects the thread whose stack gets full. The process can continue its execution, and other threads will not be affected. Also, a `StackOverflowError` produces a stack trace which you can use to identify precisely the code that caused the problem. Figure E.10 shows an example of how such a stack trace might look. You can also use project da-app-e-ex1 provided with the book to replicate this stack trace.

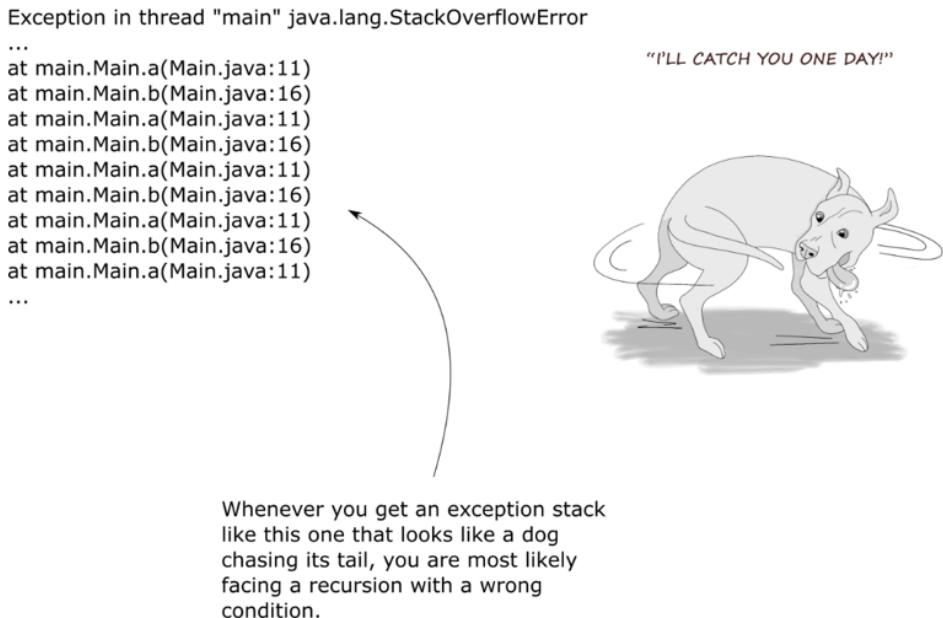


Figure E.10. The stack trace caused by a StackOverflowError. Usually, a StackOverflowError is easy to identify at a glance. The stack trace shows a method calling itself repeatedly or a group of methods that call each other, as in this example. You can directly go to these methods and find out how they got calling one another indefinitely.

E.3 The heap the app uses to store object instances

In this section, we discuss the heap. The heap is a memory location shared by all threads of a Java app. The heap stores object instance data. As you'll find out in this section, the heap causes problems more often than the stack. Also, the root causes of heap-related issues are more challenging to find. We'll analyze how objects are stored in the heap and who can keep references to them, which is relevant to understanding when they can be removed from the memory. Further, we'll discuss the main causes of issues related to the heap. You need to know all these details to understand properly the investigation techniques we discuss in chapters 7 through 9.

NOTE The heap has a complex structure. We won't discuss all the heap details since you won't immediately need them for this book's purpose. We won't discuss details such as the String pool or the heap generations to make the learning straightforward for the book's subject.

The first thing you need to remember about the heap is that it's a memory location shared by all the threads (figure E.11). Not only does this characteristic allow for thread-related issues such as race conditions to happen (discussed in appendix D), but it also makes

memory issues more challenging to investigate. Since all the threads add the object instances they create in the same memory location, one thread may impact the execution of others. If one thread suffers from a memory leak (which means it adds instances in the memory but never makes sure they can be removed), it impacts the whole process since other threads will also suffer from the lack of memory.

In most cases, as shown in figure E.11, when an `OutOfMemoryError` occurs, the situation is signaled by a different thread than the one affected by the root cause of the problem (the memory leak). The `OutOfMemoryError` will be signaled by the last thread that tries to add something in the memory and observes there's no more free space.

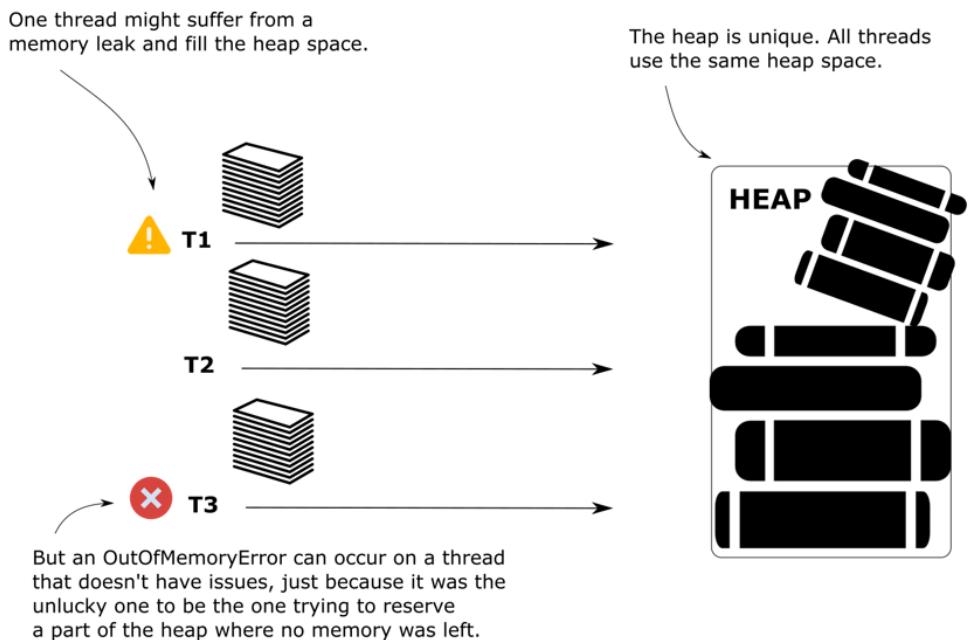


Figure E.11. All threads use the same heap location. If one of the threads has an issue causing the heap to get full (memory leak), another thread might signal the problem. This scenario happens quite often because the problem will be reported by the thread that was the last to try to store data in the heap, but couldn't. Because any thread can signal the problem and it's not necessarily the one causing the problem, heap-related issues are more challenging to find.

The garbage collector (GC) is the mechanism that frees the heap by removing unneeded data. The GC knows that no one needs an object instance anymore when no one references it. Thus, if an object isn't needed, but the app fails to remove all the references, the GC won't remove that object. When an app has a particular problem where it continues to fail to remove references to newly created objects such that at some point they fill the memory (causing an `OutOfMemoryError`), we say that the app has a memory leak.

An object instance might be referred from another object in heap (figure E.12). A common example of a memory leak is a collection where we continuously add object references. If these references aren't removed, as long as the collection is in the memory, the GC won't remove them – they are now a memory leak. You should especially pay attention to static objects (object instances referred to from static variables). These variables don't disappear once they are created, so unless you explicitly remove the reference, you can assume that object referred from a static variable will stay for the whole life of the process. If that object is a collection referring to other objects that are never removed, it may potentially become a memory leak.

The objects in heap may refer to one another. In this case, the cat instance can't be removed by the garbage collector until the reference made by the person instance is removed or the person is removed.

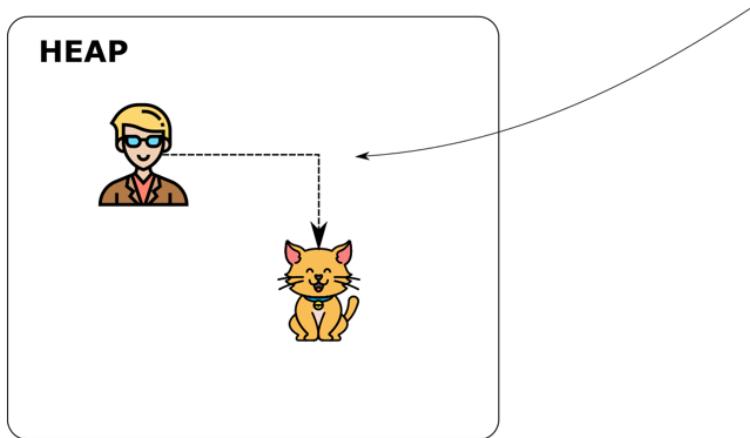


Figure E.12. Any object in the heap can keep references to other objects in the heap. The garbage collector can remove an object only when no reference to it exists.

An object instance can also be referred to from the stack (figure E.13). Usually, references from the stack don't cause memory leaks since (as discussed in section E.2) a stack layer disappears automatically when the execution reaches the end of the code block for which the app created the layer. But in specific cases, when combined with other issues, references from the stack can also cause trouble. Imagine a deadlock that keeps the execution from running through a whole block of code. The layer in the stack won't be removed in this case, and if it keeps references to objects, this may also become a memory leak.

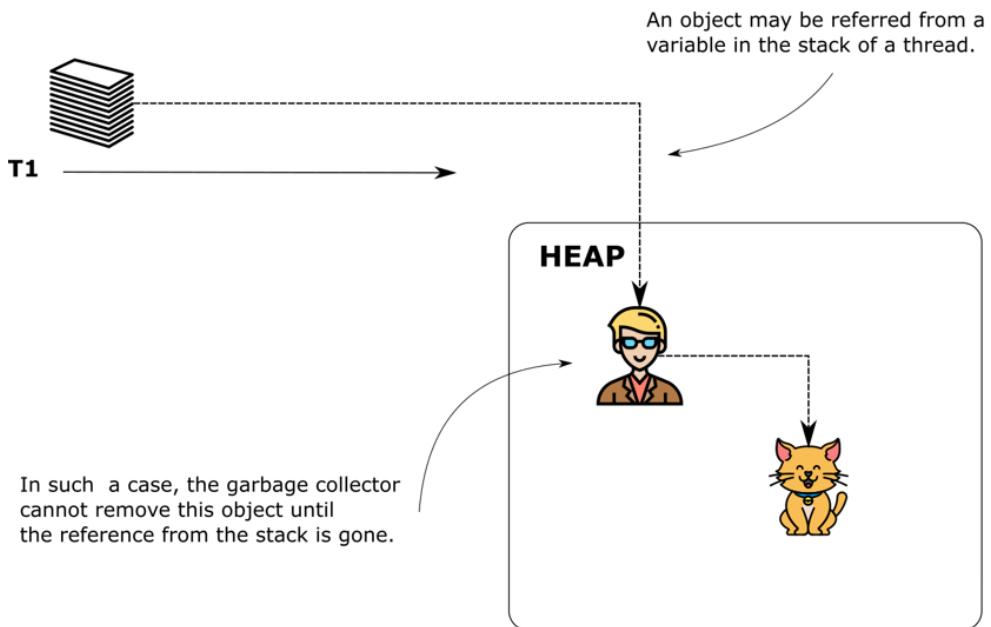
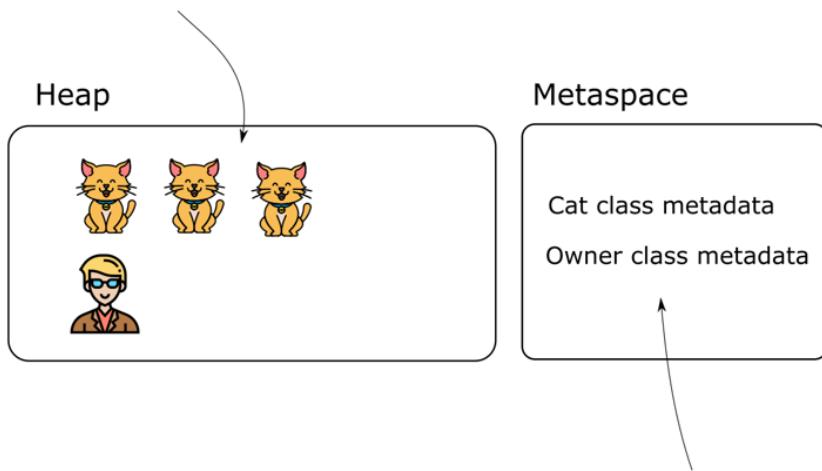


Figure E.13. A variable in the stack can also refer to an instance in the heap. The instance cannot be removed until all its references are gone (including the ones in the stack).

E.4 The metaspace memory location for storing the data types

The metaspace is a memory location the JVM uses to store the data types used to create instances stored in the HEAP (figure E.14). The app needs this information to handle the object instances in heap. Sometimes, in specific conditions, an `OutOfMemoryError` can also affect the metaspace. If the metaspace becomes full and there's no more space for the app to store new data types, the app throws an `OutOfMemoryError`, complaining that the metaspace is full. In my experience, these errors are rare, but I would like you to be aware they may show, and you might have to investigate such issues too.

The heap may contain several instances of several types of objects.



The metaspace contains the information that tells the JVM how to understand and create instances that are found in the heap.

Figure E.14. The metaspace is a memory location where the app stores the data types descriptors. We can say that the metaspace stores the blueprints used to define the instances stored in the heap. The app uses metaspace to store all details it needs to know how to create and understand the object instances afterward.