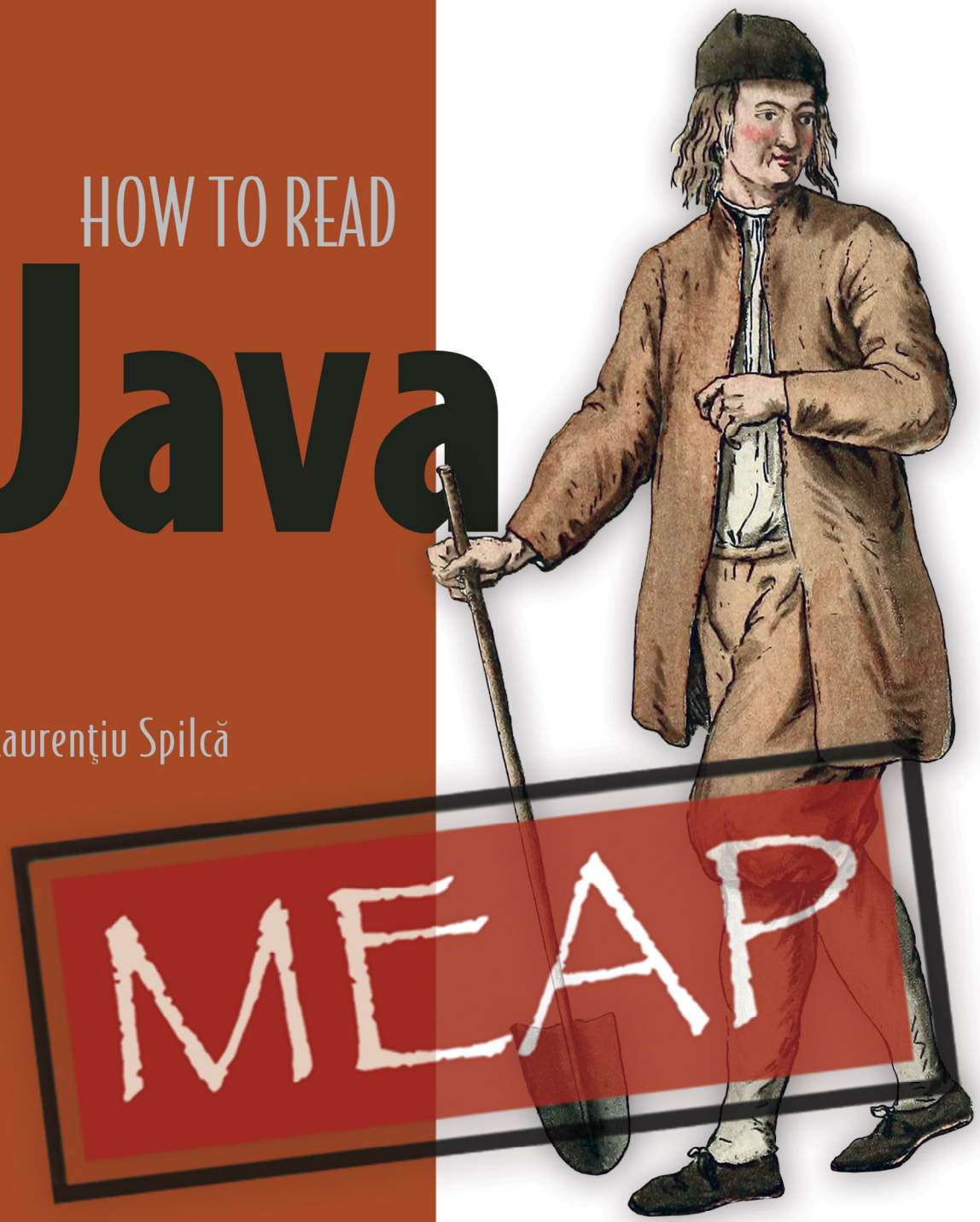
Understanding, debugging, and optimizing JVM applications

HOW TO READ Java



Laurenţiu Spilcă

MANNING

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

Version 3

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.

—Laurenţiu Spilcă

brief contents

1

Starting to know your apps

This chapter covers

- What is a code investigation technique
- What code investigation techniques we use to understand Java apps
- What will you learn in this book

As software developers, we find out soon enough after starting our careers that a big part of what we're doing is understanding software behavior. We, software developers, are craftsmen, but crafting software is different than crafting anything else. In many cases, we build new capabilities and solve problems over something that already exists, and, because of this, we need first to understand the behavior of what's already there.

Whether we implement some new capability or solve a problem in existing code, we spend more time understanding how software behaves than changing the software. And for this reason, learning investigation techniques well is gold. Knowing to use these techniques will save you valuable time and make you more efficient.

1.1 How to easier understand your app

In this section, we look at several frequent 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? Why do we analyze code and apps' executions?

We mainly investigate the app's behavior to

- To find a particular root cause of an issue (bug)
- To understand how a particular software capability works so we can enhance it
- Test a capability we implemented
- To learn a specific technology (for example, a framework 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):

- **Reading code** – follow the code logic line by line to understand what it does
- **Debugging** – using a debugger tool to pause the execution on defined line and manually navigate through the executing code
- **Profiling** – observing what code executes and analyzing execution details such as the time spent by specific instructions or how many times these instructions run.
- **Log analysis** – Using log messages to understand software behavior and identify potential bugs.
- **Thread state analysis** – using thread dumps to identify thread-related problems such as locks, CPU or memory consumption.
- **Memory state analysis** – using memory dumps to identify memory consumption problems such as memory leaks or issues related to the way data is stored which can cause security vulnerabilities.
- **Mocking and stubbing** – Using tools to simulate apps that communicate with a given process to identify integration problems.

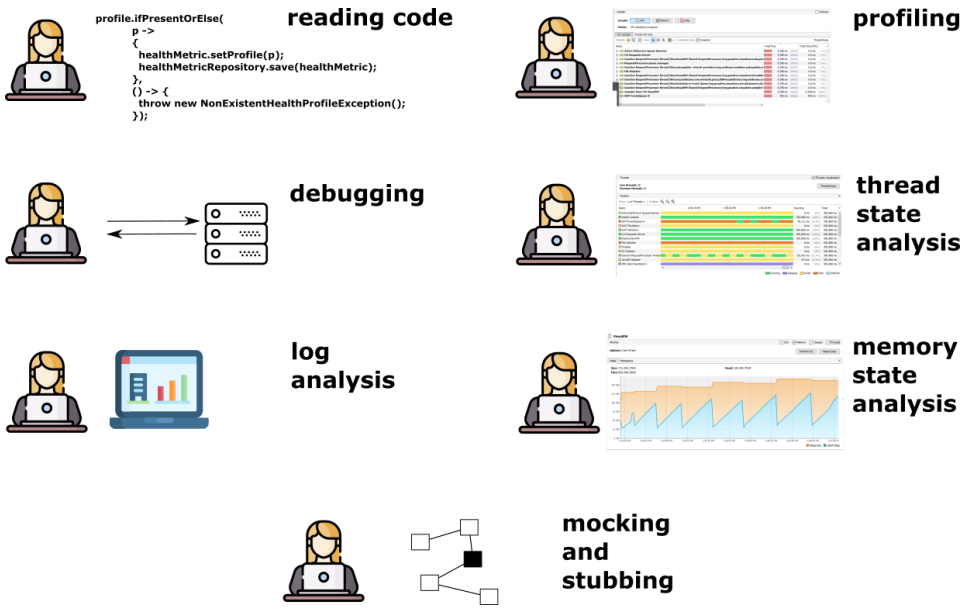


Figure 1.3 Code investigation techniques. Depending on the case, a developer would choose one or more of these techniques to understand how a certain capability works.

NOTE 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 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 takes 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 shouldn't 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 practices 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. One other common reason for messy code is stress and time pressure caused by strict deadlines (yes, sometimes poor management can cause this effect).

In most cases, you can't avoid all these situations, but you need to be able to investigate such poor 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. You would like to re-write this code to make its purpose easier to understand. When we re-write code to make it easier to read we say we are "**refactoring**". This code in listing 1.1 definitely needs refactoring. But before you can refactor it, you need to understand what's doing now. 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. With a bit of experience and some tricks we'll discuss in chapters 2 through 4, 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 through 4, we'll take some of these examples where using a debugger tool helps you understand the code faster and discuss this technique more.

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.

You find the code presented in listing 1.1 as part of the project da-ch1-ex1 provided with the book. Appendix A discusses the recommended tools to use with the examples of the book, and appendix B helps you with opening and starting the projects provided with this 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 read 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's 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 (as you'll learn in chapters 6 through 8) to identify what code executes before deciding where to start the investigation.

A **profiler** is a tool we use to analyze details of an app's execution, such as what methods execute, how long it takes them to execute and how many times they are called.

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 after the unfortuntate event happened. 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 instruments that we'll discuss in chapters 9 through 11.

1.2 Typical scenarios for using investigation techniques

Let's briefly discuss in this section some common scenarios for using code investigation approaches that we'll detail in the rest of the book. 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.
- Test a new capability you implemented in an app.
- Learn how certain dependencies work (libraries or frameworks).
- 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 and how to combine them depending on the case to faster get to a solution.

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.

1.2.1 Demystifying the unexpected output

The most frequently encountered 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 (figure 1.4). 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 is 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.

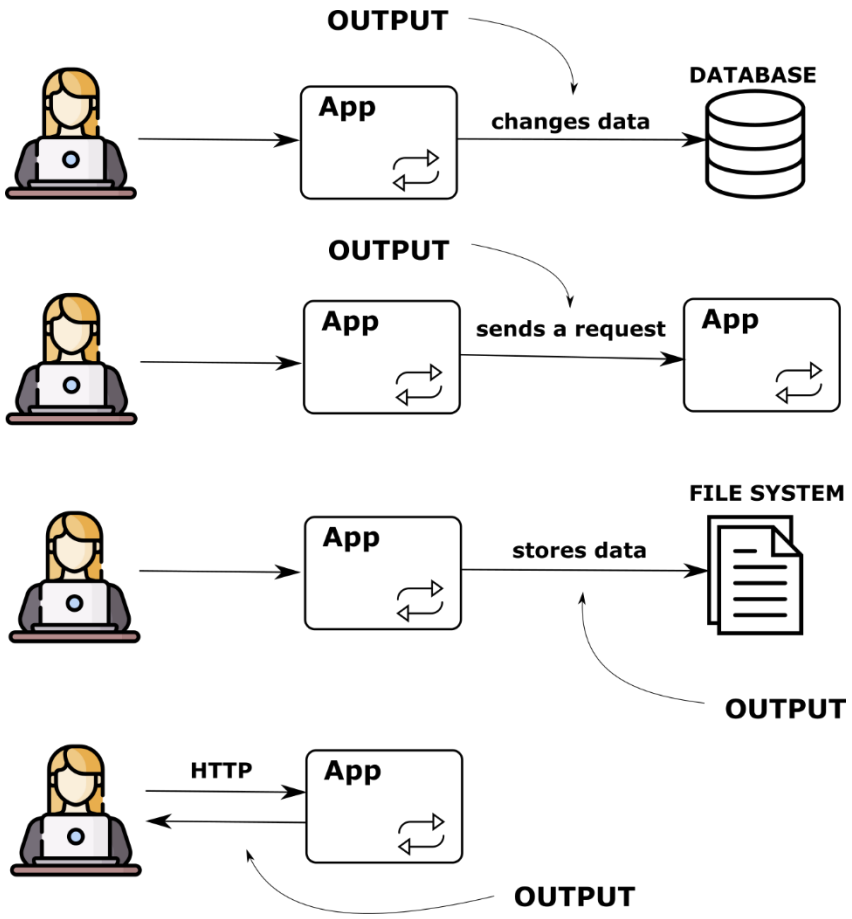


Figure 1.4 Different definitions of output. Any data change, display, or executed actions resulting from running a certain part of the app is the output of that specific capability. The app adding records in a database, storing information in a file, sending an HTTP request, or sending data in an HTTP response are examples of outputs. In many cases, we use various approaches to investigate why such an output differs from what we expect.

How we investigate a case in which a specific part of the app doesn't have an expected execution result? You can use a debugger to pause the app's execution and easily follow its logic. Or you can use log messages to observe what did the app do. Sometimes, you need even more advanced techniques like using a profiling tool (profiler) to identify what code executes and understand where to start from.

But choosing the right techniques to use really depends a lot on the case. For example, situations become even more complicated when we have to deal with logic implemented through multiple threads – a multi-threaded architecture. In most such cases, using a debugger might 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 1.5).

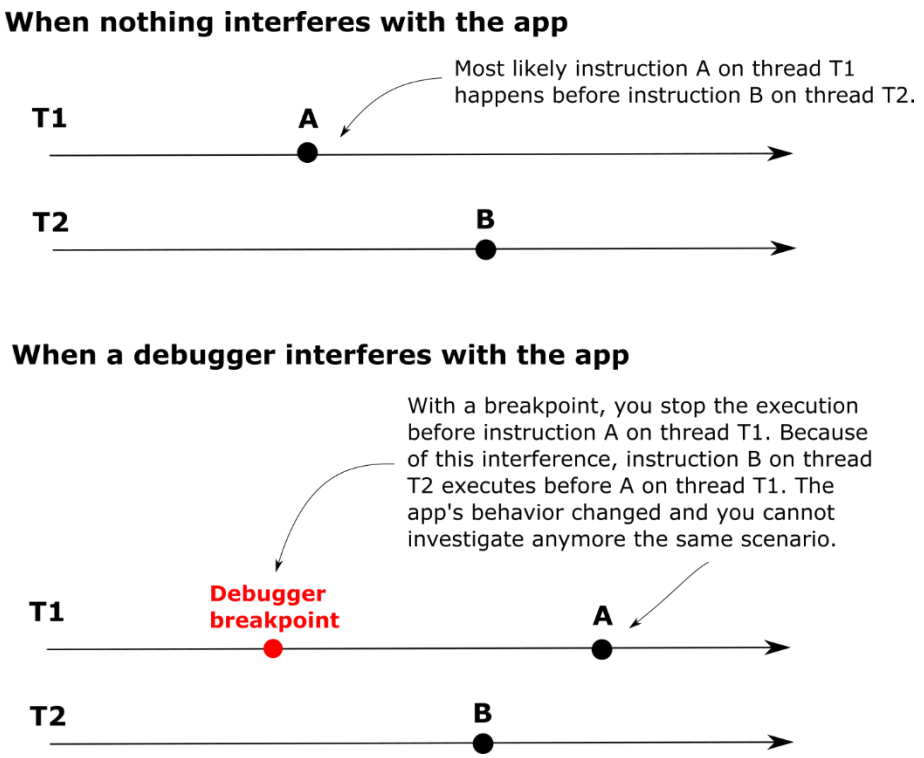
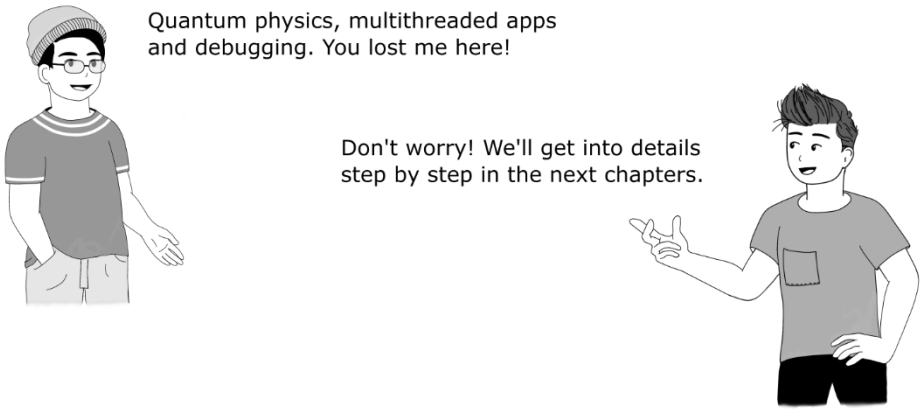


Figure 1.5 A Heisenberg execution. In a multi-threaded app, when a debugger interferes with the app's execution, it might change how the app behaves. This change doesn't allow you to correctly investigate the initial app behavior that you wanted to research.

For the area of a 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.



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 1.6 visually presents this case.

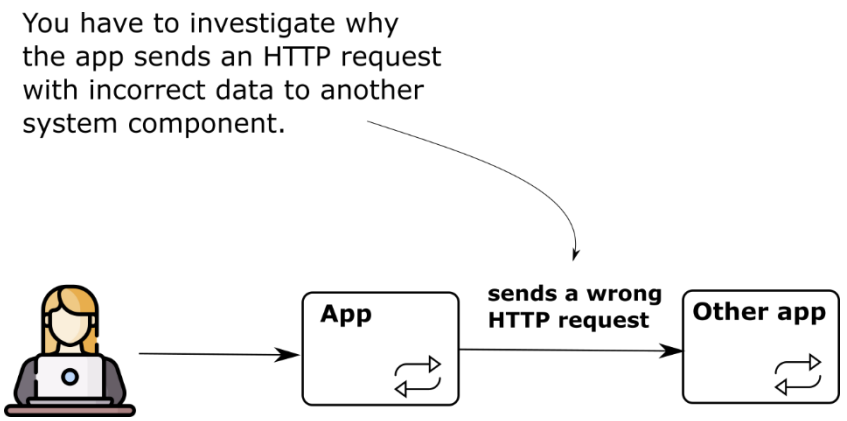


Figure 1.6 You can replace the system component your app calls with a stub. You control the stub to allow to determine where your app sends the request from more quickly. Also, you can use the stub to test your solution after you correct the issue.

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 from. I replace the other app (the one my app wrongly sends requests to) with a stub (figure 1.7). **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. Using stubs when investigation app behavior is something we'll discuss in detail in chapter 10.

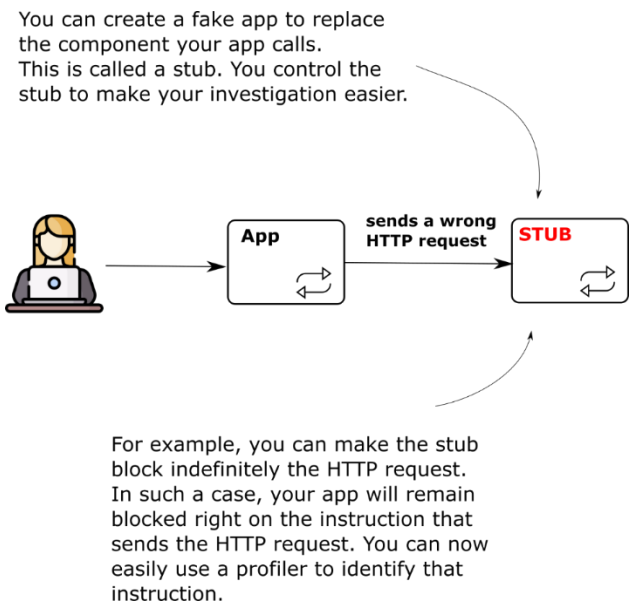


Figure 1.7 You can replace the system component your app calls with a stub. You control the stub to allow to determine where your app sends the request from more quickly. Also, you can use the stub to test your solution after you correct the issue.

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 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.

I'll start with my favorite, Spring Security (https://www.manning.com/books/spring-security-in-action). 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 along with other investigation techniques 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 use the investigation techniques you learn in this book with all examples provided with books you read to learn various technologies.

The second example of technology I mostly learned through investigation code 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. Chapters 7 through 9 describe some excellent profiling approaches I use every time when I implement apps that use a persistence framework such as Hibernate.

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.

1.2.3 Clarifying slowness

For some problems you need to use specific tools to identify the root causes. 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 investigation 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 and 7.

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 8. 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 1.8 shows how a profiler displays the execution time.

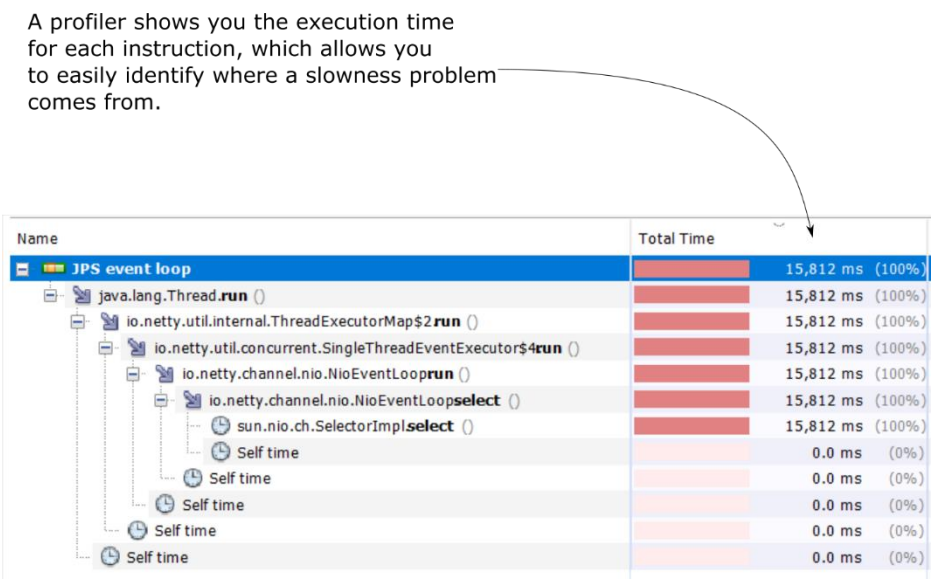


Figure 1.8 Investigating slowness problems with a profiler. The profiler shows you the time spent by each instruction during code execution. This profiler feature is excellent for identifying the root causes of performance problems.

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 because the app needs to wait for the data to be transferred from or to a specific point. 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

In this section, we discuss investigating issues that caused an app not to respond anymore. These kinds of problems are usually also 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 (or replicating) 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 to reproduce 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 9. 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 1.9 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 have 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 9. In this section, I just try to draw you the idea of what and why you'll learn further in the book.

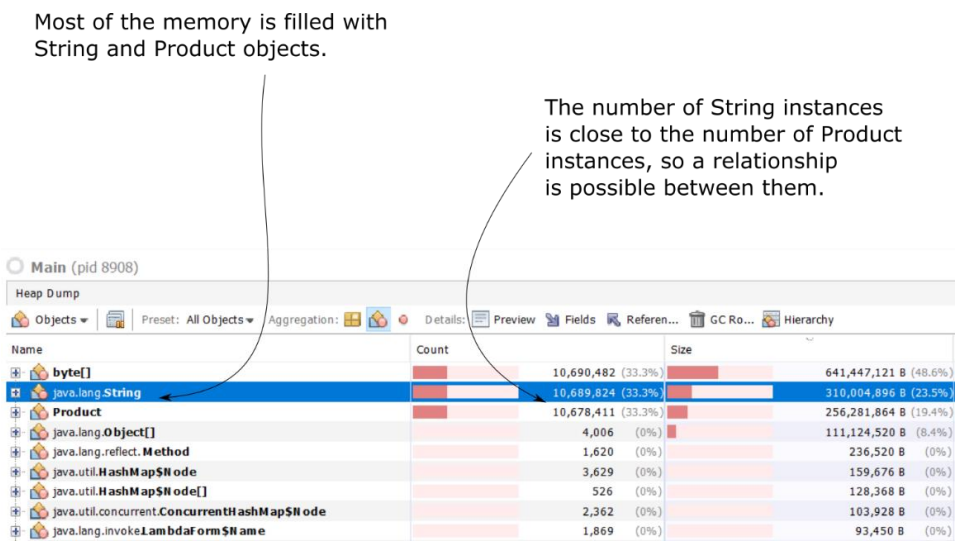


Figure 1.9 A heap dump is like a heap memory map. If you learn how to read it, it gives you invaluable clues of how the app internally processes data. A heap dump helps you investigate memory problems or performance issues. In this example, you see how easy you find out which object fills most of the 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 1.10 shows you an example of thread dump and some of the details you can easily observe with this tool.

Figure 1.10 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 9, 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 beginner to expert. In this book, you'll learn various code investigation techniques, scenarios in which to apply them, and how to apply them to bring you to benefit from saving 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 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 to choose using 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.

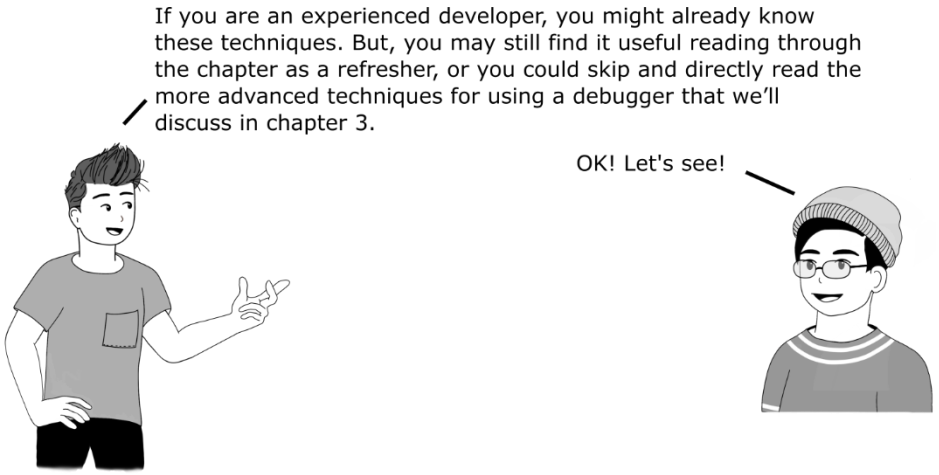
In this chapter, we discuss one of the most important tools used in understanding code: **the debugger**.

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 a compass 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 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.



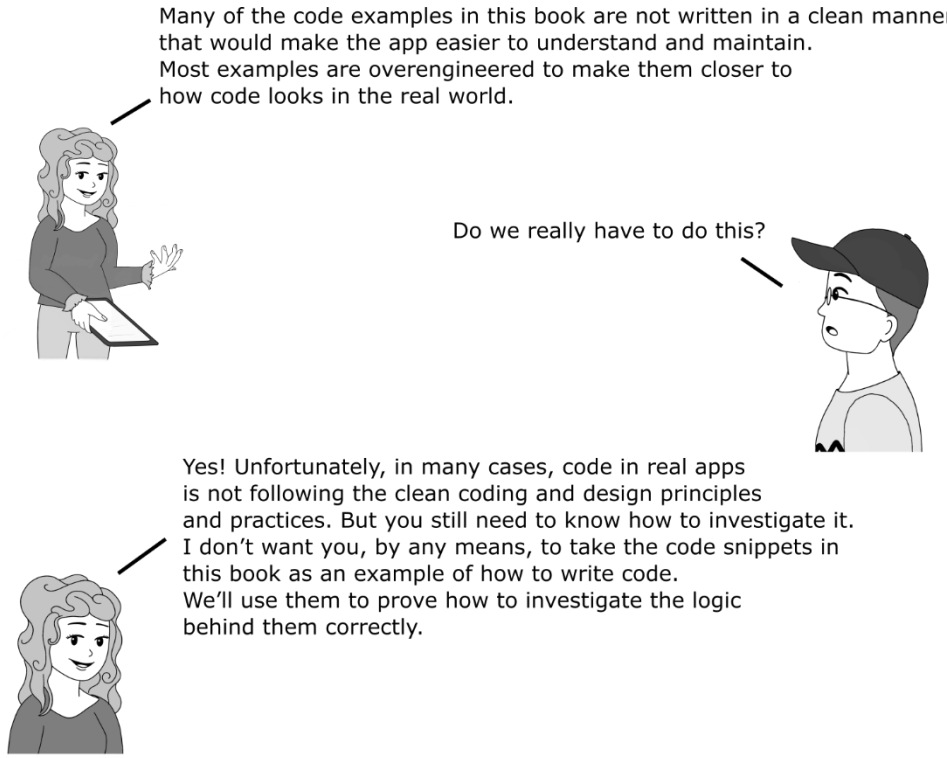
2.1 When analyzing code is not enough...

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?

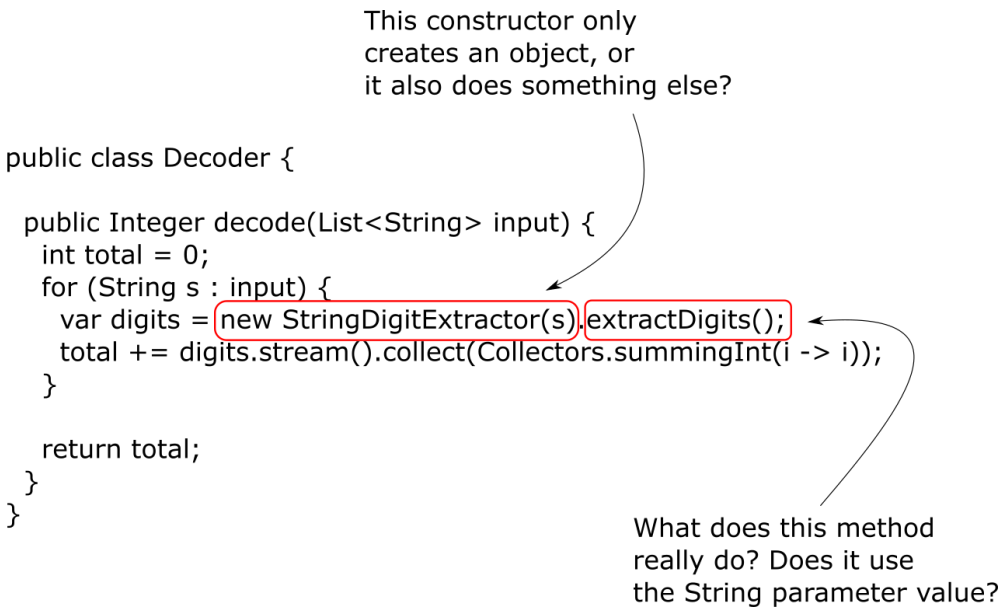


Figure 2.1 When reading a piece of code, you often need to figure out what happens behind the scenes in some of the instructions composing that logic. The names given to methods are not always suggestive enough, and you can't totally rely on them. Instead, you need to go deeper into what these methods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

(Lesson about the cube, N. Stănescu)

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

Figure 2.2 Reading poetry is linear - you just read up to down the text line by line.

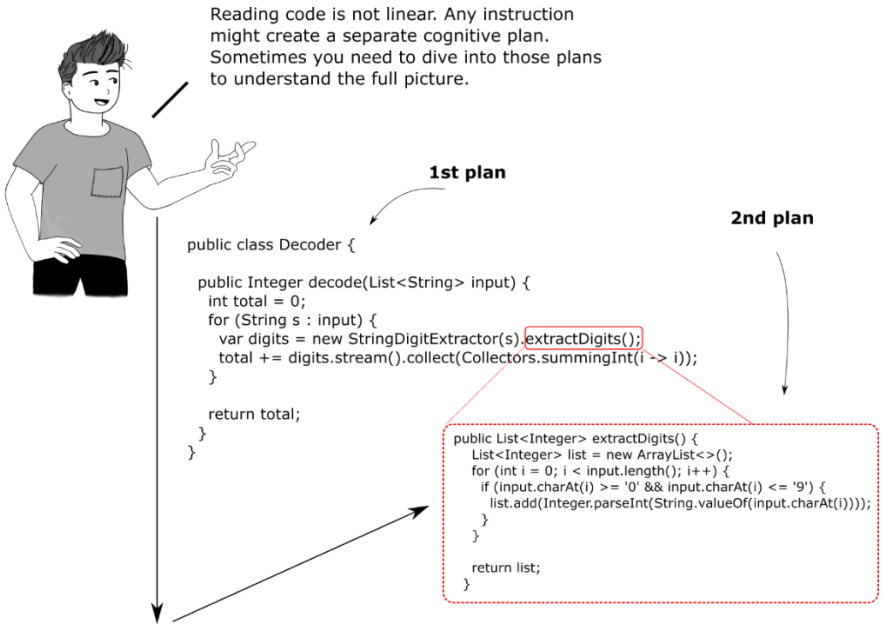


Figure 2.3 Reading code is different than reading poetry and adds a lot of complexity. You can imagine reading code as reading in two dimensions. One dimension is reading a current piece of code up to down. The second dimension is going into a specific instruction to understand it in detail. Remembering 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, 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 is the instruction where 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? 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 straightforward (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.

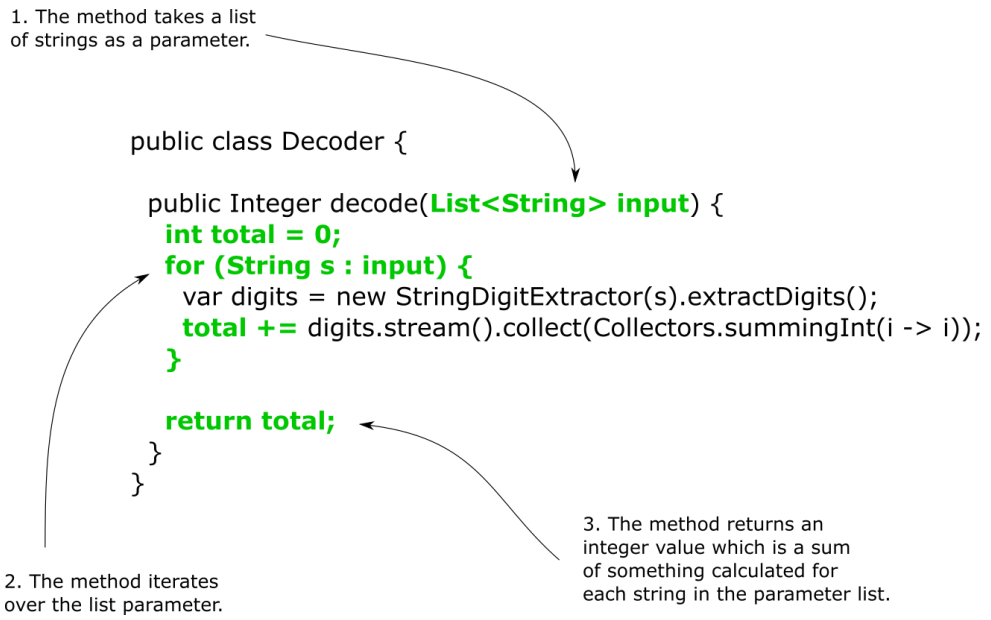


Figure 2.4 Assuming you know the language fundamentals, you can easily observe that this code takes a collection as input and parses the collection to calculate an integer.

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.

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.

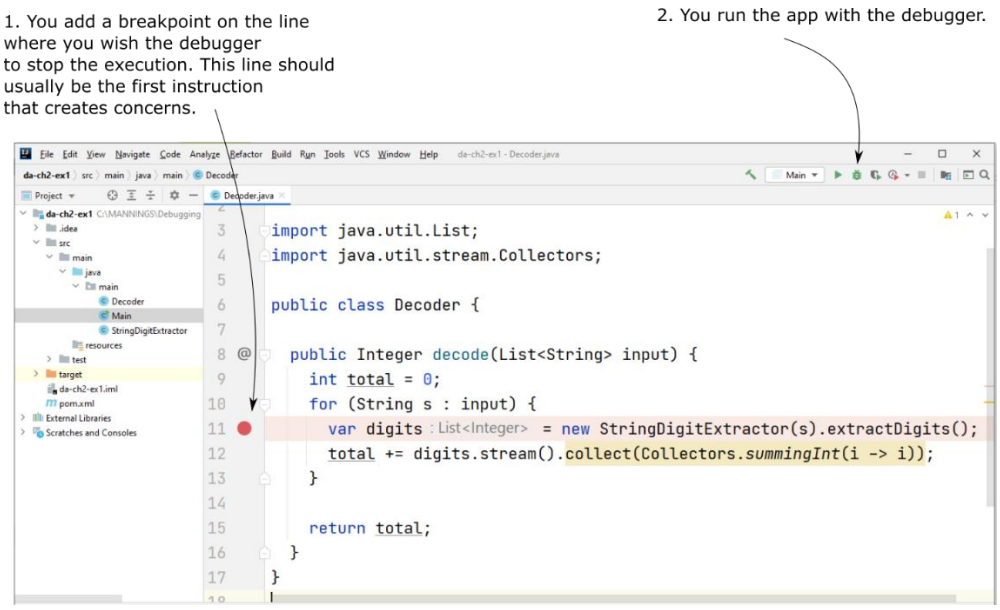
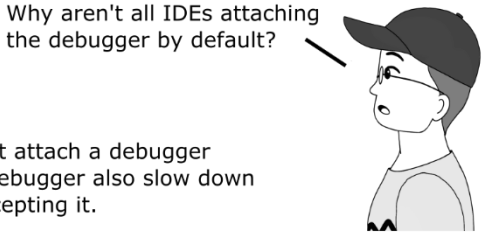


Figure 2.6 Click near the line number to add a breakpoint on a specific line. Then, run the app with the debugger. The execution pauses on the line you marked with a breakpoint and allows your control further.

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 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.
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 you only use a limited number at a time and focus only on those lines of code you marked with the breakpoints. I usually don't exceed three breakpoints I use at the same time. I often see developers adding too many breakpoints, forgetting them, and getting lost on the investigated code.

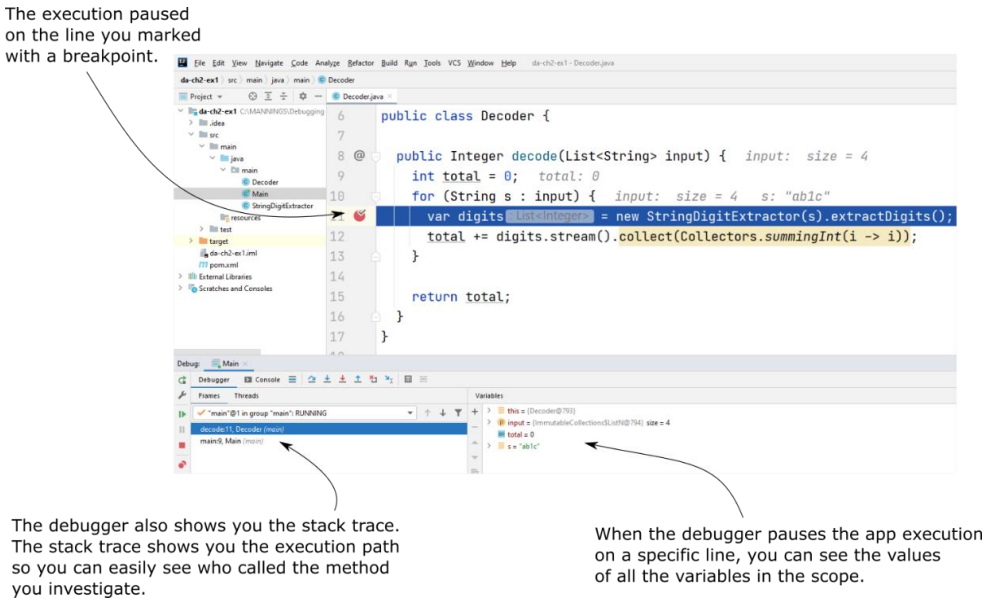


Figure 2.7 When the execution is paused on a given line of code, you can see all the variables in scope and their values. You can also use the execution stack trace to remember where you are navigating through the lines of code.

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?

In this section, we discuss the execution stack trace. 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 call 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.

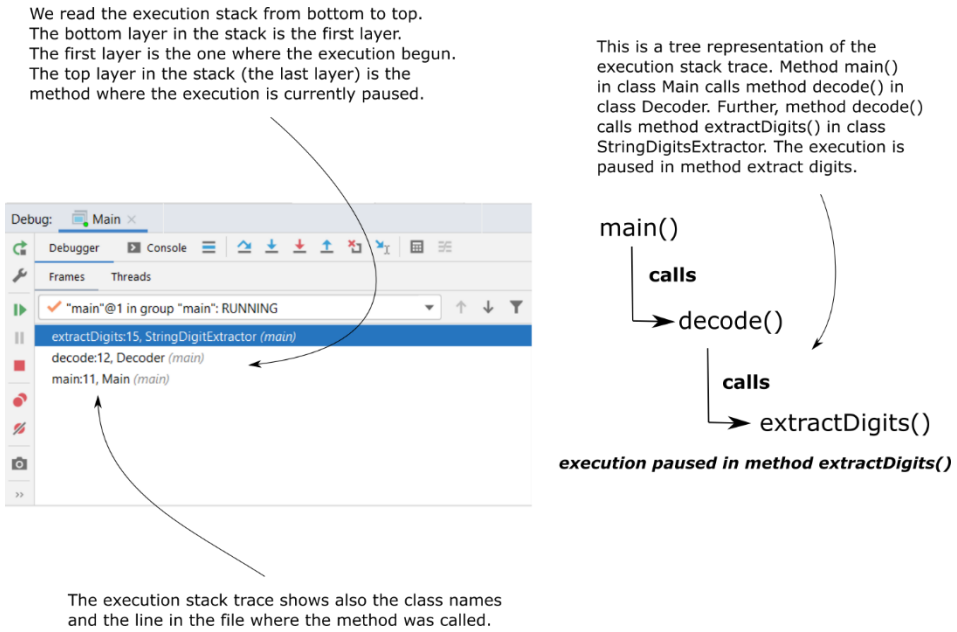


Figure 2.8 The top layer of the execution stack trace is the method where the debugger paused the execution. Each other layer in the execution stack trace is where the method represented by the above layer has been called. The bottom layer of the stack trace (also named the „first“ layer) is where the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

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, Quarkus, Hibernate, and so on) sometimes alter the execution chain of the method.

For example, Spring apps often use code 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.

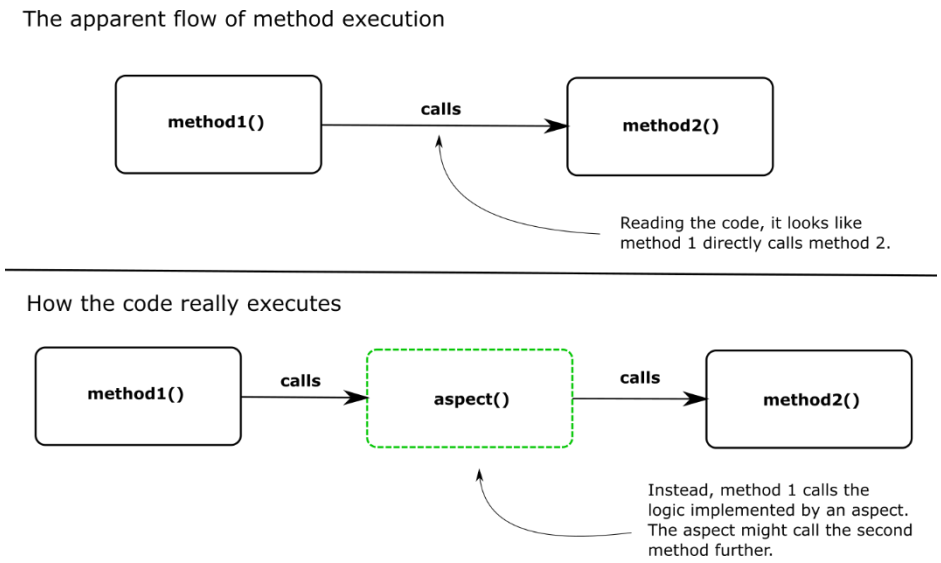


Figure 2.9 An aspect logic is completely decoupled from the code. For this reason, when reading the code is difficult to realize there's more logic that will execute than you actually see. Such cases where you have hidden logic executing can be confusing when investigating a certain capability.

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` 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.

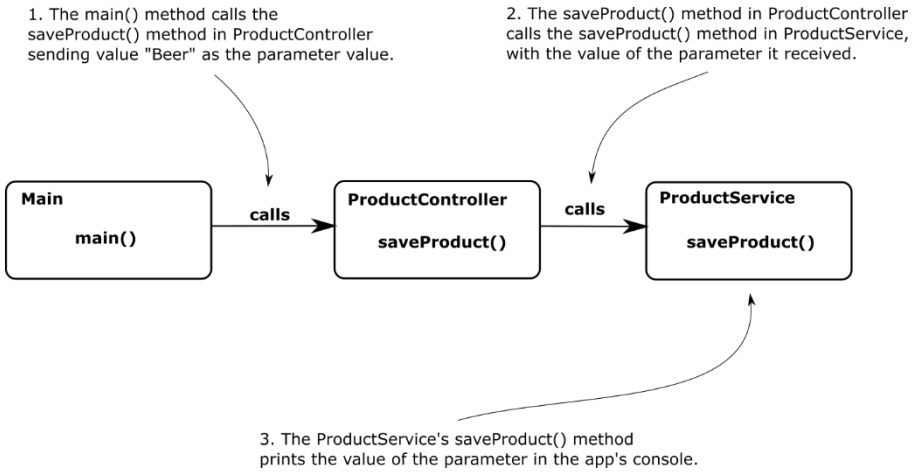


Figure 2.10 Method main() calls saveProduct() of bean ProductController sending the value "Beer" as the parameter value. The ProductController's saveProduct() method calls the ProductService bean sending the same parameter value as the one it receives. The ProductService bean prints the parameter value 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).

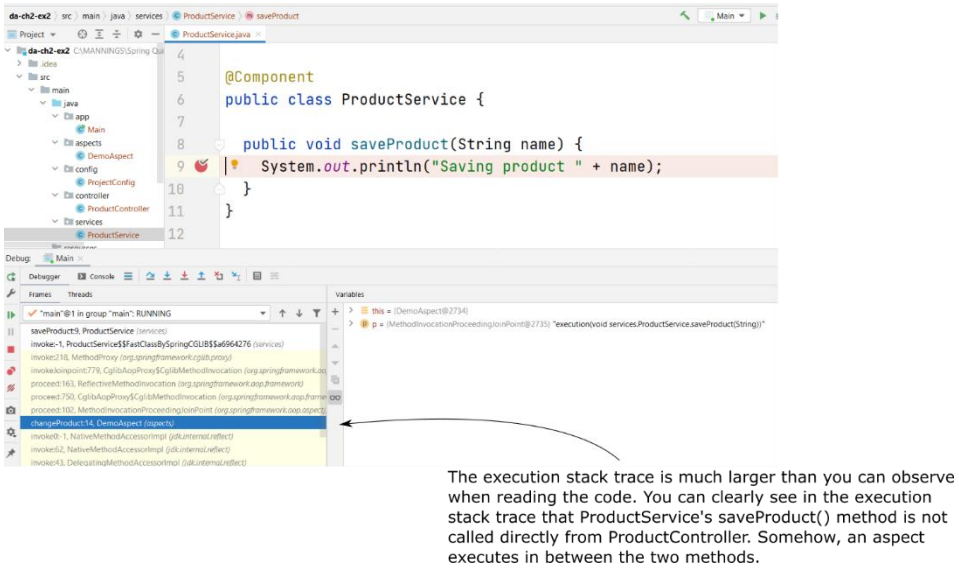


Figure 2.11 The execution stack trace shows that an aspect has altered the execution. This aspect is the reason why the value of the parameter changes. Without using the stack trace, finding why the app has a different behavior than expected would have been more difficult.

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 GUI, the IDE I use.

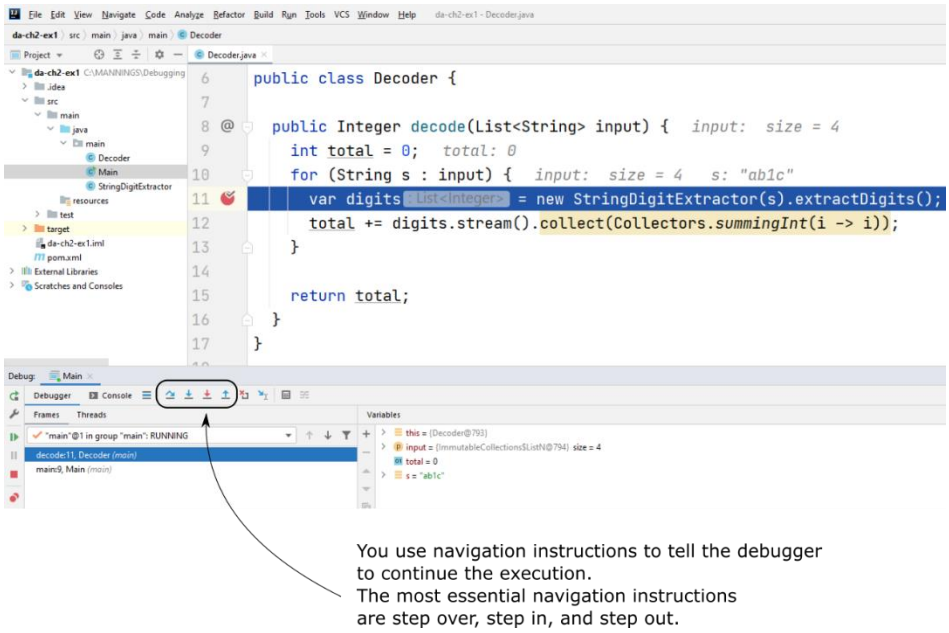


Figure 2.12 The navigation operations help you "walk" through the app logic in a controlled way to identify how the code works. To navigate through code, you can use the buttons on the IDE's GUI (as presented in the figure) or use the keyboard shortcuts associated with these operations.

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".

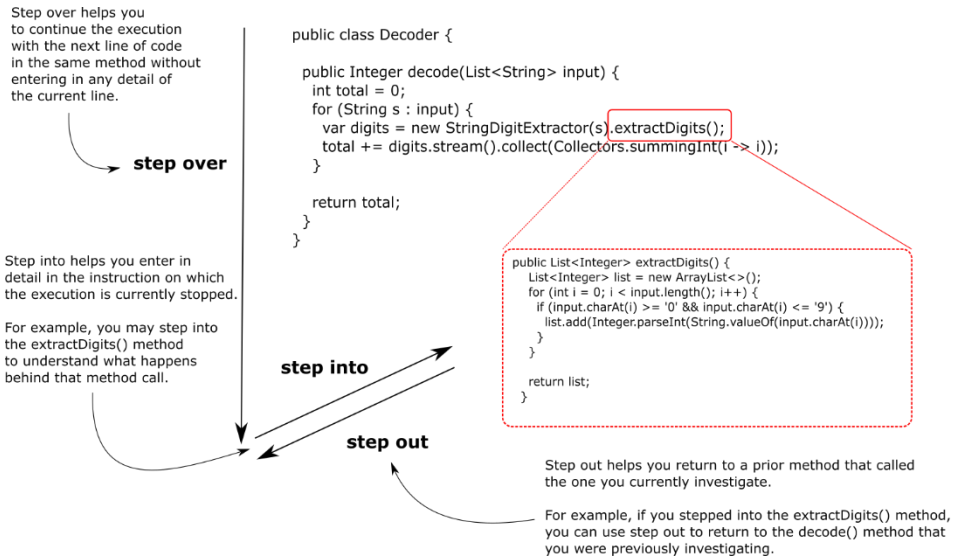


Figure 2.13 Navigation operations. Stepping over helps you go to the next instruction in the same method. When you want to start a new investigation plan and go into details in a specific instruction, you can use the step into operations. You can go back to the previous investigation plan with the step-out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

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

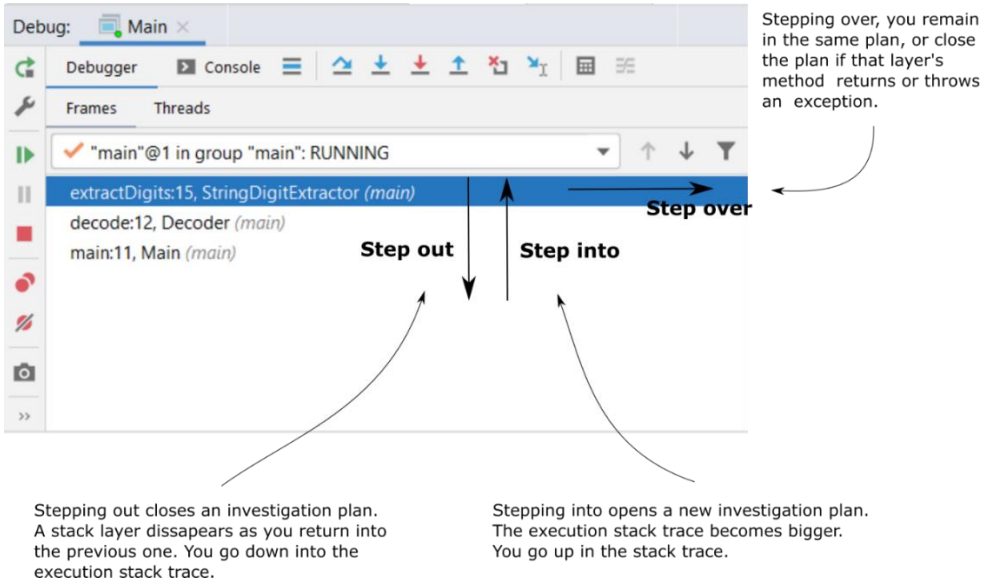


Figure 2.14 Navigation operation as seen from the execution stack trace point of view. When you step out, you go down in the stack trace and close an investigation plan. When you step into, you open a new investigation plan so you go up in the stack trace and the stack trace becomes bigger. Stepping 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) portraits 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 in 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.

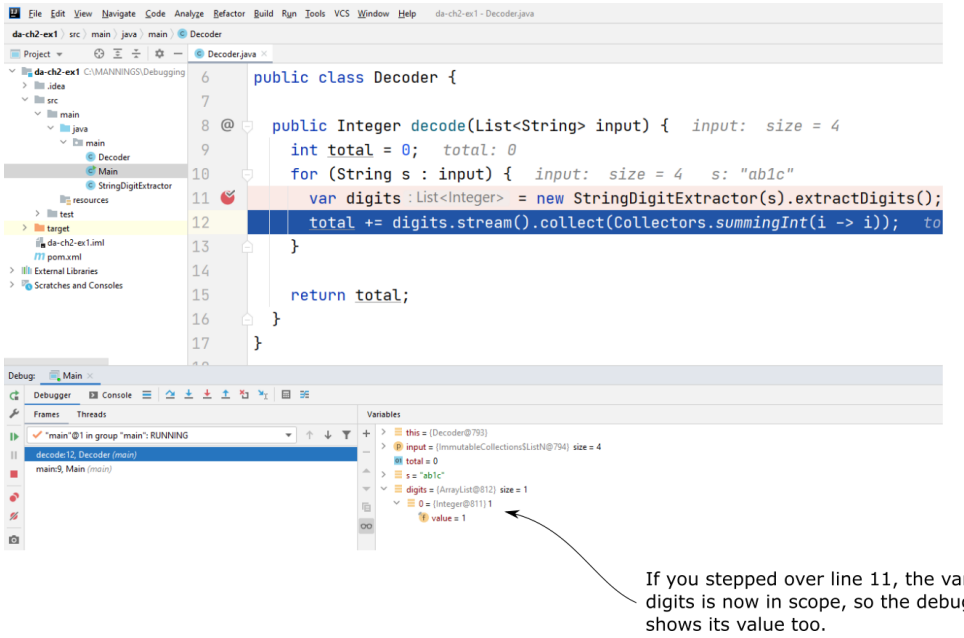


Figure 2.16 When you step over a line, the execution continues in the same method. In our case, the execution paused next on line 12, and you can see the value of the digits variable that was initialized by line 11. You can use this value to deduce what line 11 does without having to go into more detail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

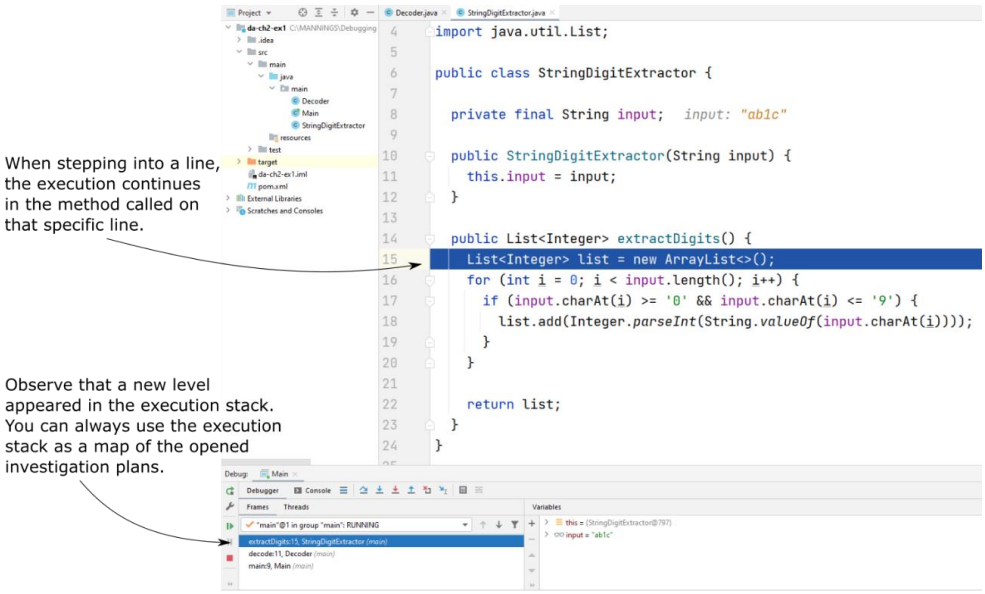


Figure 2.17 Stepping into allows you to observe the entire execution of the current instruction. This opens a new investigation plan, allowing you to parse the logic behind that particular instruction you stepped into. You can use the execution stack trace to retrace the execution flow.

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.

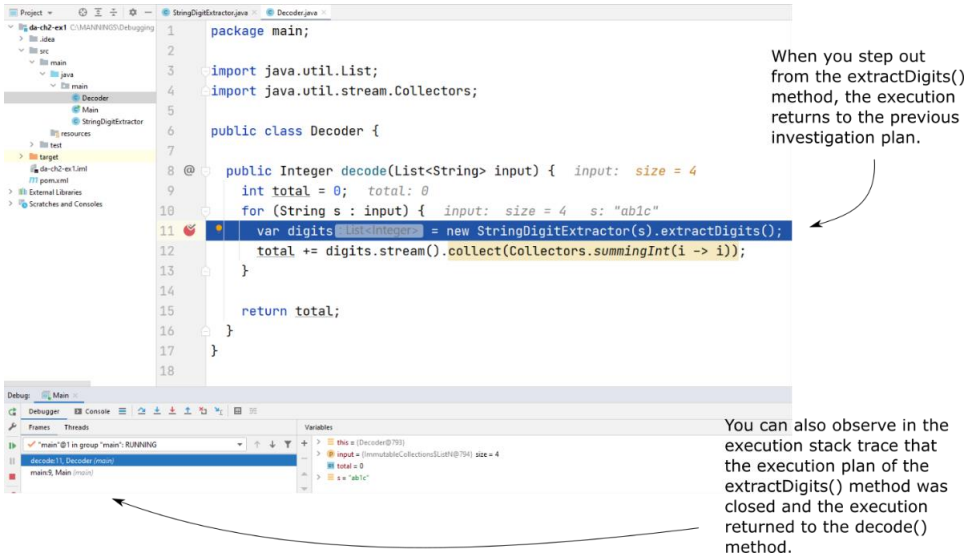


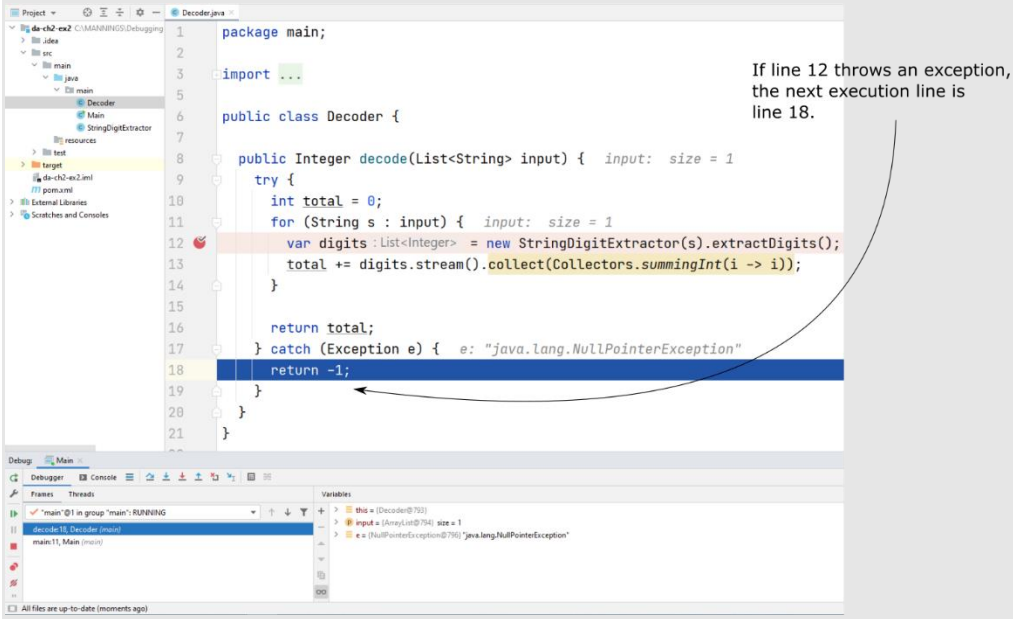
Figure 2.18 The step out operation helps you close an investigation plan and return to the previous in the execution stack trace. Using the step out operation is helpful to save time since you don't have to step over each instruction until the current execution plan closes by itself. Stepping out offers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



In this figure, we step over from line 12 and line 12 throws an exception, the execution continues on line 18, which is the next execution line. The next execution line is not always the next 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 not the right approach:

- Investigating output problems when you don't know which part of the code creates the output
- Investigating performance problems
- Investigating app crashes
- Investigating multithreaded implementations

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

TIP 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 stick 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:

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

- 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 efforts 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 if 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 already figure 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 situations**
- **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.

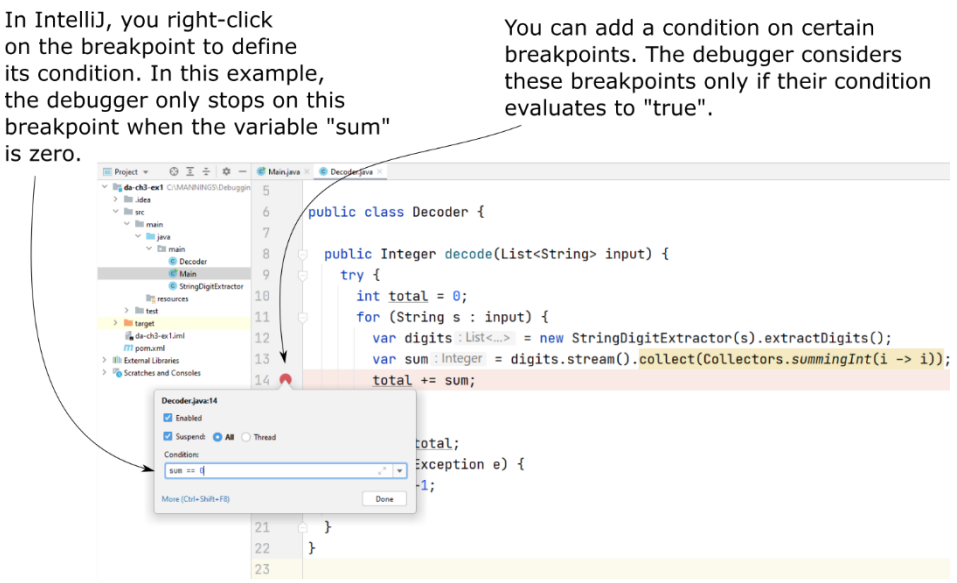


Figure 3.1 Using a conditional breakpoint to stop the execution just for specific cases. In this figure, we want to pause the execution on line 14 only if sum is zero. We can apply a condition on the breakpoint that instructs the debugger to consider that breakpoint only if the given state is true. This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

execution 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.

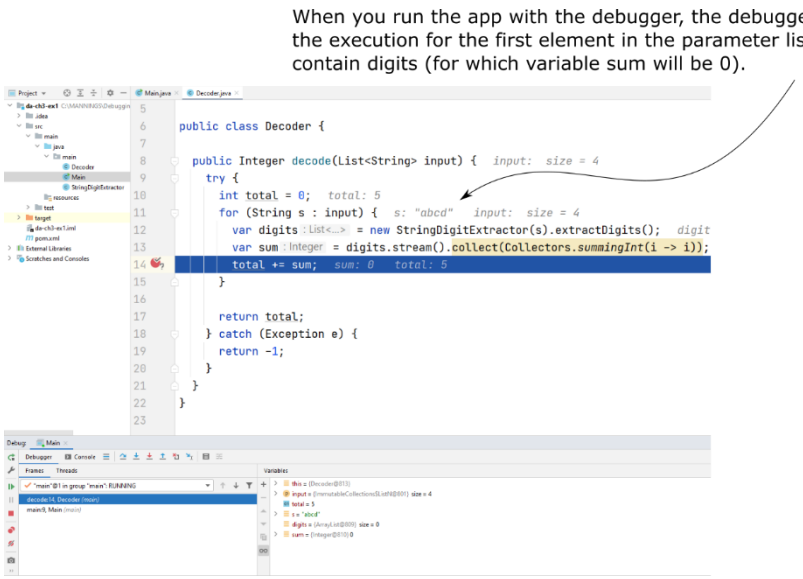


Figure 3.2 A conditional breakpoint. Line 14 in the figure was executed multiple times, but the debugger only paused the execution when the variable sum was zero. This way, we skipped over all the cases we were not interested in, and we'll start directly with the conditions relevant to our investigation.

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.

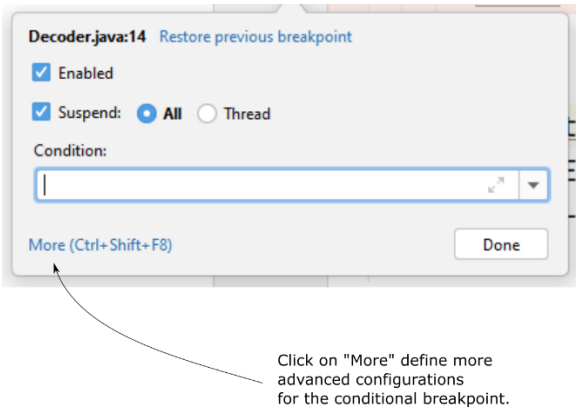


Figure 3.3 To apply advanced configuration on the breakpoint in IntelliJ, you can click on the More button as presented in the figure.

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.

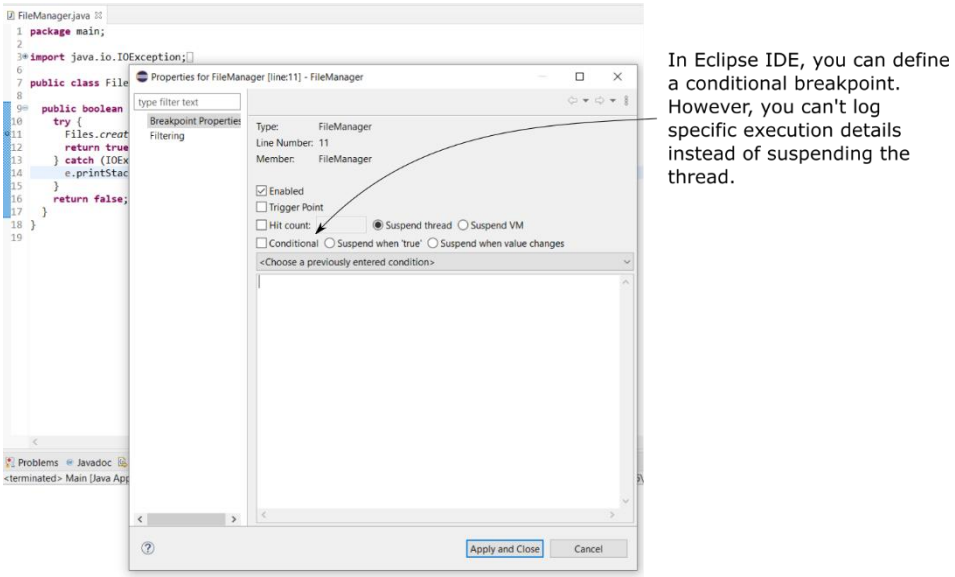
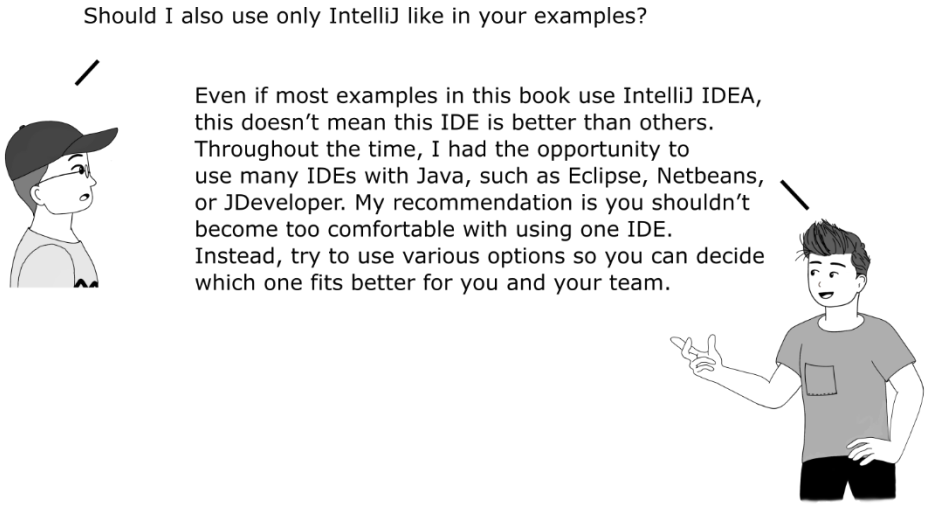


Figure 3.4 Not all IDEs offer the same debugging tools. All IDEs give you the basic operations, but some features, such as logging the execution details instead of pausing the execution, might be missing. In Eclipse IDE, you can define conditional breakpoints, but you can't use the logging feature.

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.



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 3.4 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.

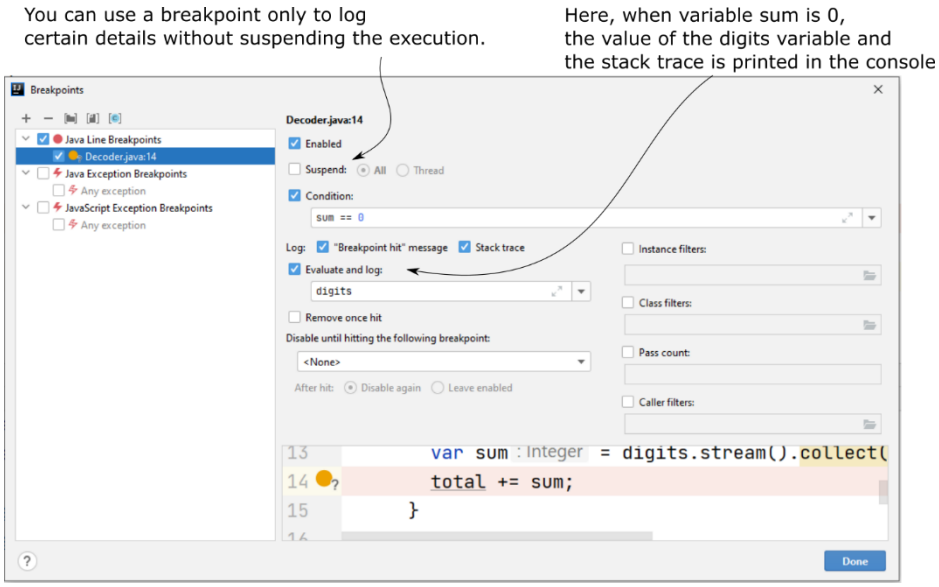


Figure 3.4 Conditional breakpoint advanced configuration. Besides specifying a condition for the breakpoint, you can instruct the debugger not to suspend the execution for the given breakpoint. Instead, you could simply log the data you need to understand your case.

Figure 3.5 shows the result of running the app with the conditional breakpoint configured, as presented in figure 3.4. 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.

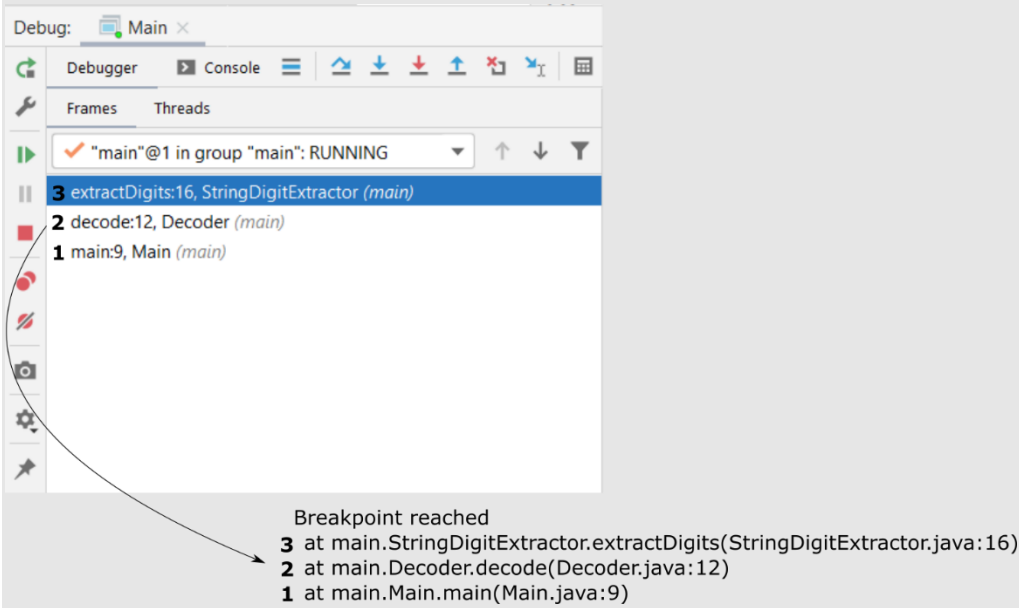
Figure 3.5 Using breakpoints without pausing the execution. Instead of pausing the execution on the line marked with a breakpoint, the debugger logs a message when the line has been reached. The debugger also logs the value of the digits variable and the execution stack trace.

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.



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 3.6).

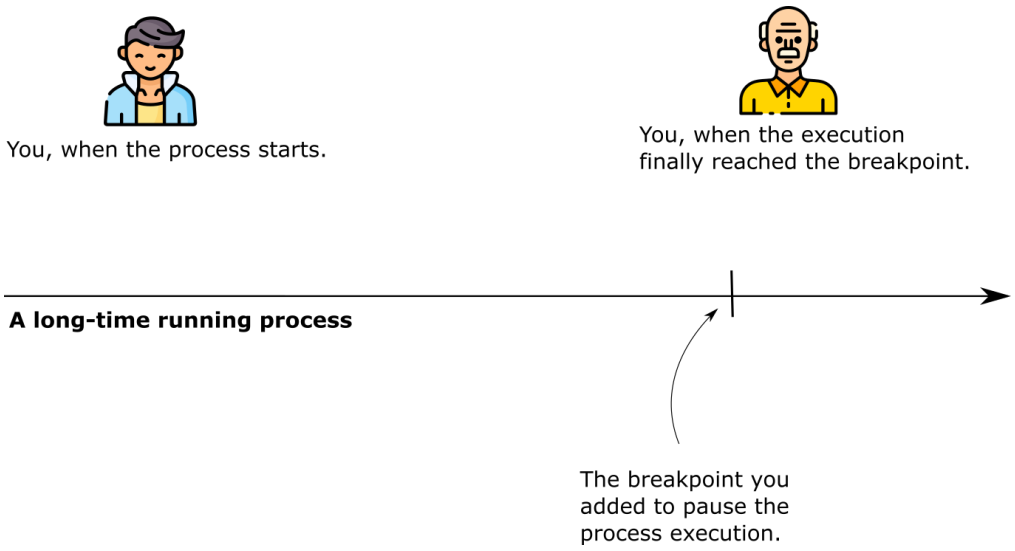


Figure 3.6 Usually, when investigating issues in a long-running process, using breakpoints is not really an option. This is because it can take a long time for the execution to reach the part of code you are investigating, and if you have to rerun the process a few times, you will definitely spend too 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 3.7 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.

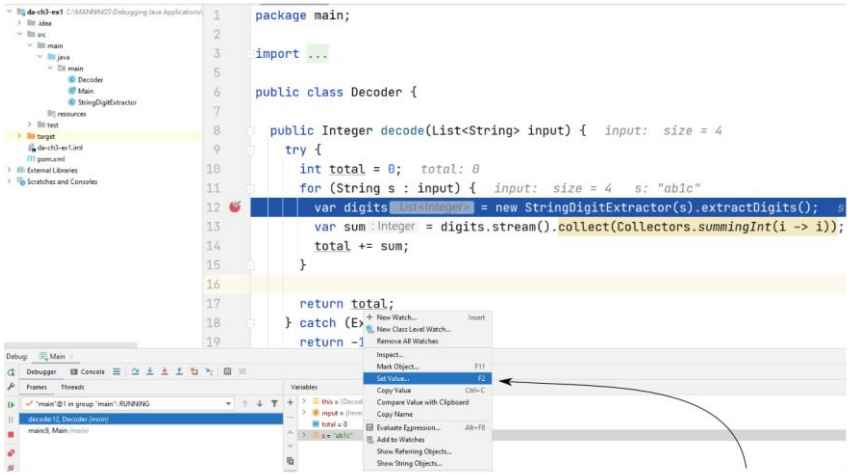


Figure 3.7 Setting a new value in a variable in scope. The debugger shows you the values for the variables in scope when it pauses the execution on a given line. Besides being able to see the variable values, you can also change them to create a new investigation case. This approach helps you, in some 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 3.8. 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`.

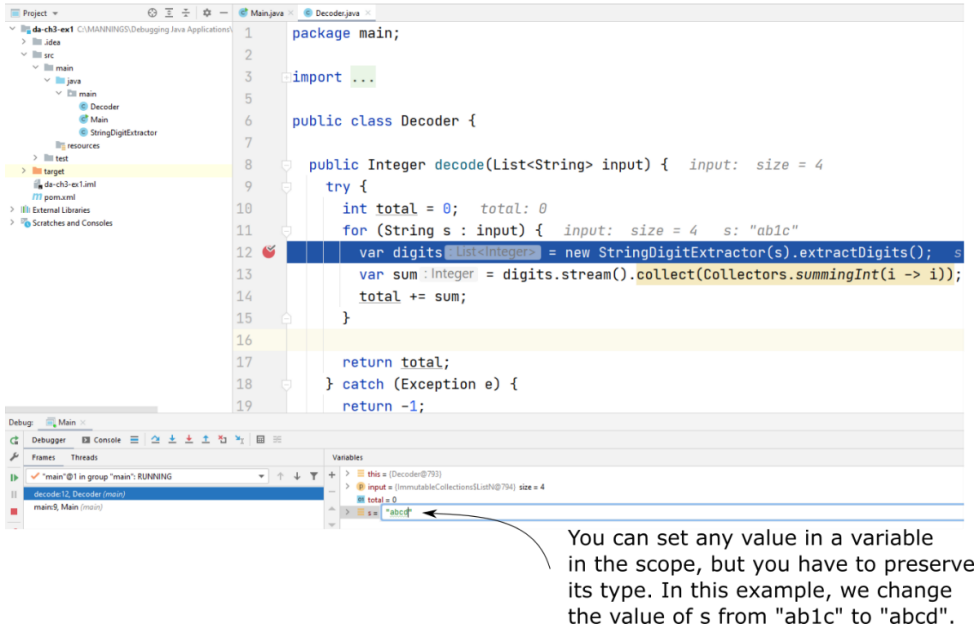


Figure 3.8 Change the variable's value to observe how the app's execution behaves in different conditions.

When you continue the execution, as presented in figure 3.9, 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.

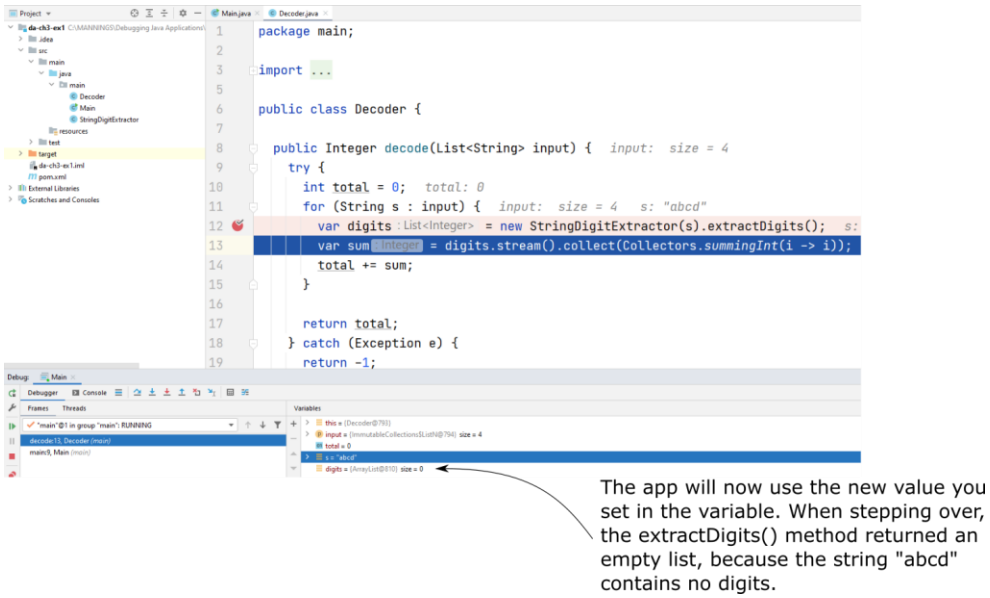


Figure 3.9 When using the step-over operation, the app now uses the new value you set to the "s" variable. Method extractDigits() returns an empty list because string "abcd" doesn't contain digits. Setting values in variables on the fly helps you test different scenarios even if you don't have the input data you need.

Let's compare the conditional breakpoints approach we discussed in section 3.1 with this approach 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 either don't have the right data to cause the scenario you want to investigate, or 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.

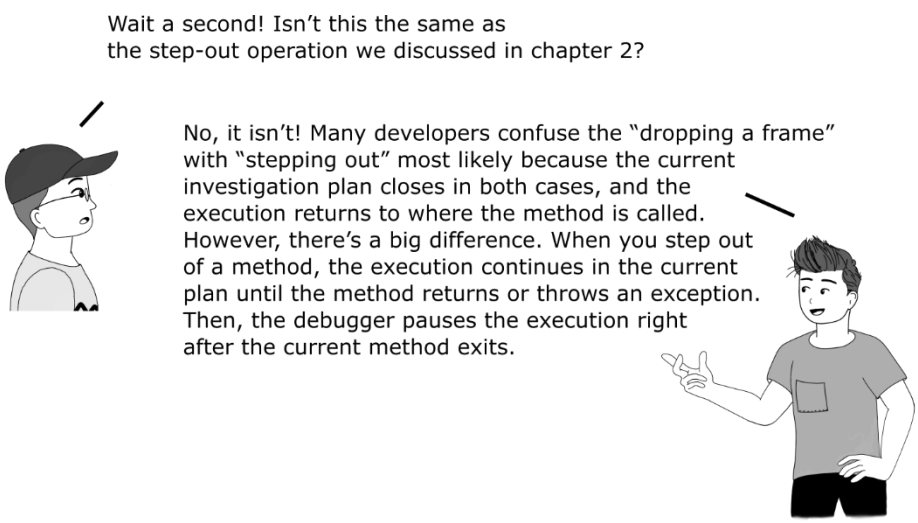


Figure 3.10 shows you how step-out works using the example you find in project da-ch3-ex1. You are in the `extractDigits()` method, which, as you can see from the execution stack trace, has been called from the `decode()` method in the `Decoder` class. If you use the step-out operation, the execution continues in the method 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.***

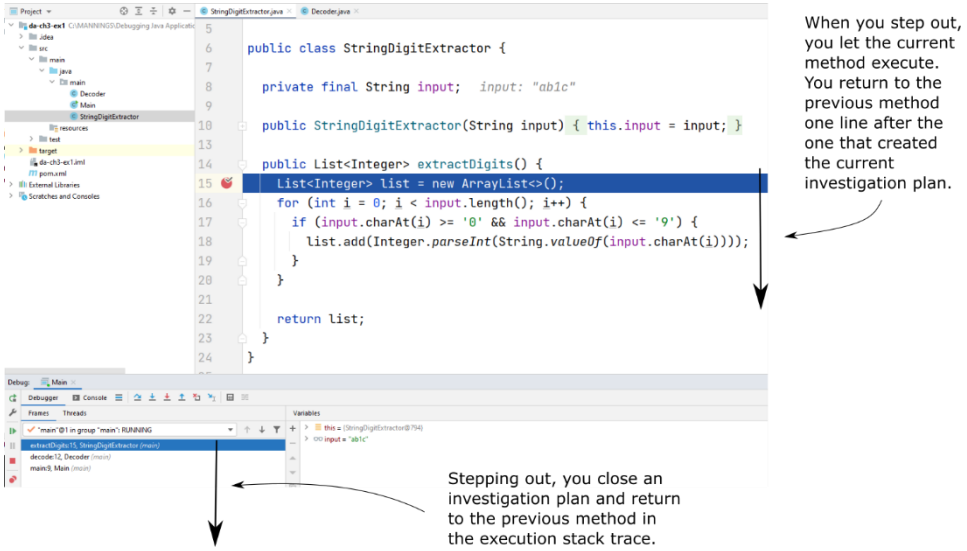


Figure 3.10 Stepping out closes the current investigation plan by executing the method and then pausing the execution right after the method call. This operation helps you continue the execution and return one layer in the execution stack.

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 3.11) is like rewind.***

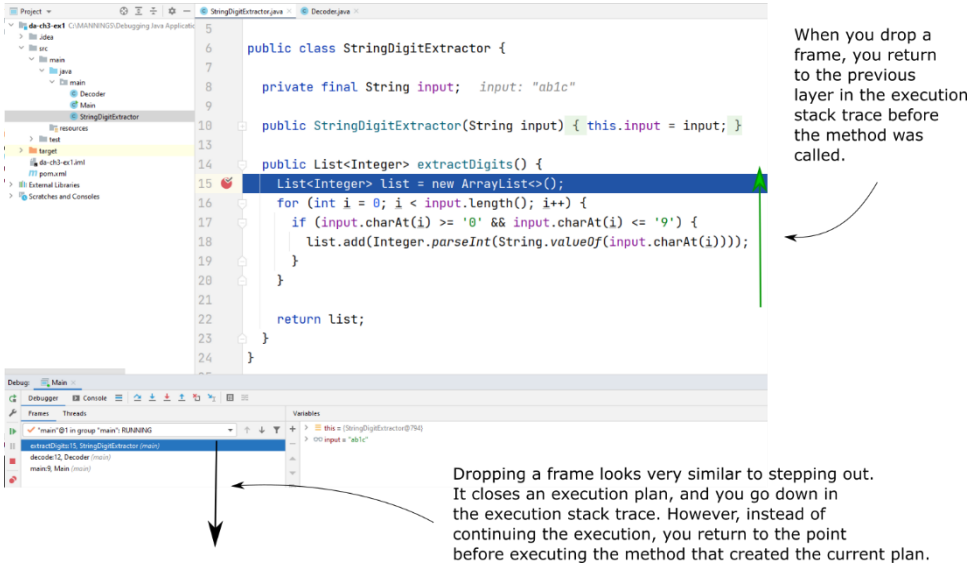


Figure 3.11 When you drop a frame, you return to the previous layer in the execution stack trace before the method execution. This way, you can replay the method execution either by stepping again into it or stepping over it.

Figure 3.12 shows you, relative to our example, a comparison between stepping out from the `extractDigits()` method or dropping the frame. 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.

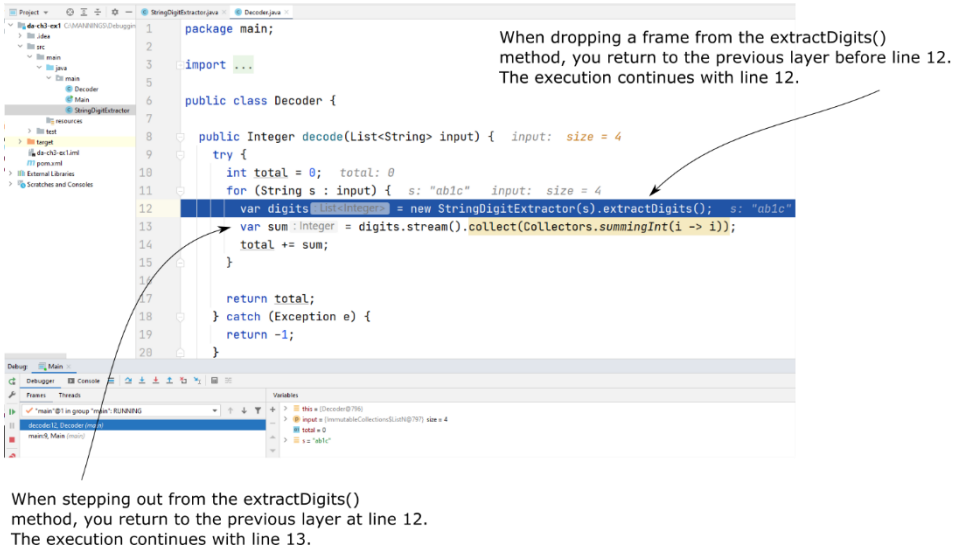


Figure 3.12 Dropping a frame vs. stepping out. When you drop a frame, you return before the method's execution. When you step out, you continue the execution but close the current investigation plan (represented by the current layer in the execution stack).

Figure 3.13 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".

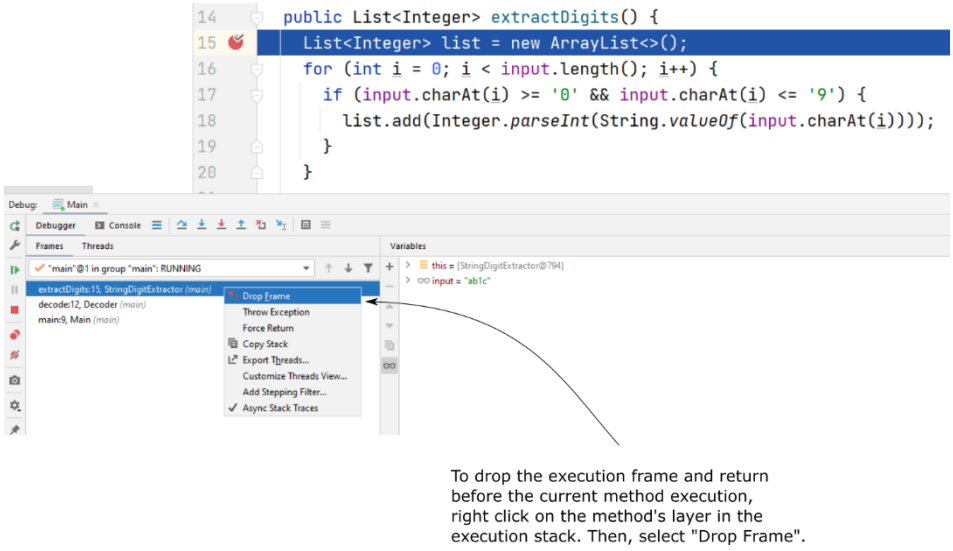


Figure 3.13 When using IntelliJ IDEA, you can drop a frame by right-clicking on the method's layer in the execution stack trace and then selecting "Drop Frame".

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 will 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 3.14):

- 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.

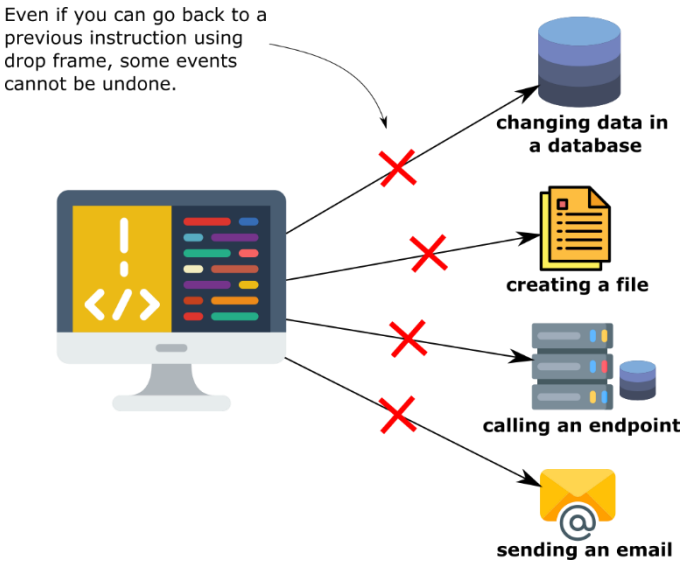


Figure 3.14 The drop frame operation cannot undo some events. Some examples of such events are changing data in the database, changing data in the file system, calling another app, or sending an email message. You need to be careful with such cases as they might affect your investigation.

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 but 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. 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.

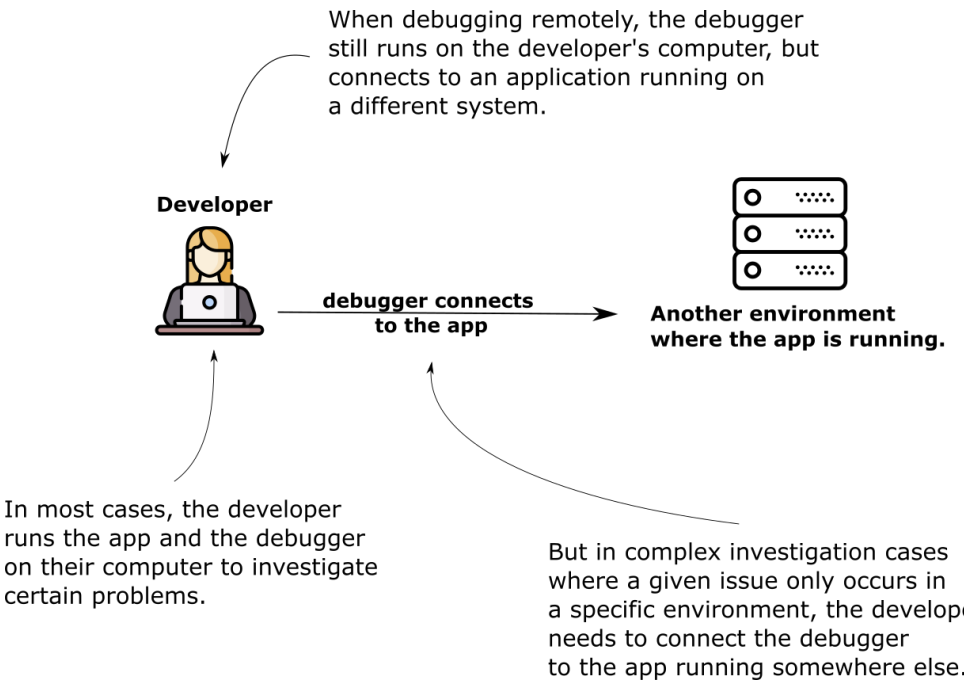


Figure 4.1 Remotely debugging an app. The developer can run the debugger tool locally on their computer but connect it to an app instance running in a different environment. This approach allows the developer to investigate problems that only occur in specific environments.

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 use 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 (UAT) environment. 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.

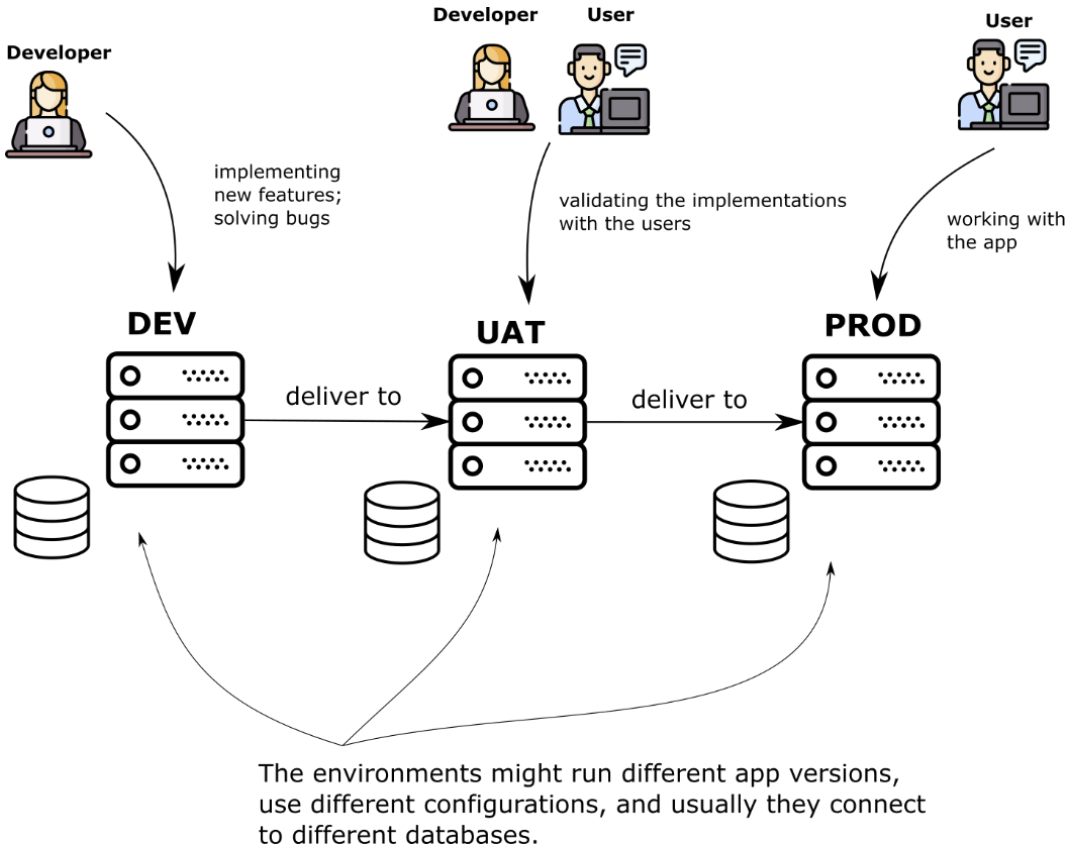


Figure 4.2 When working on a real-world app, developers often use multiple environments to run the app they build. First, the developers build the app in development (dev) environments. Then, once a feature or a solution is ready, they present it to the users (or stakeholders of the app) using a user 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.
- 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. A few examples of questions you should ask yourself are:

- Do all environments use only one database, or each environment uses its own?
- Do the environments rely on the same version of an operating system, or they use different operating systems?
- What resources (CPU and memory) are allocated for each environment?
- Do they have different security policies configured?

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 should never 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 on a capability 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.

NOTE Never use remote debugging in the production environment. Enabling remote debugging could slow down the app and cause performance issues, expose data and cause security vulnerabilities, or even completely stick the execution sometimes.

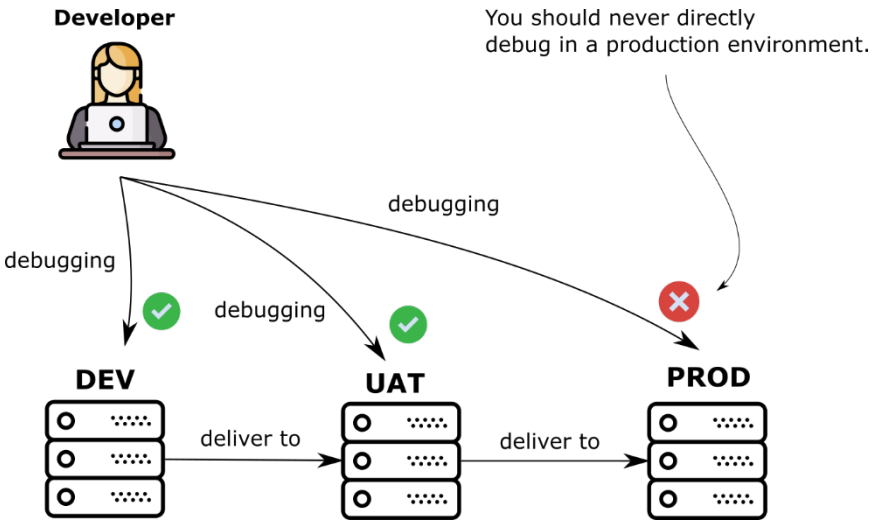


Figure 4.3 The developers implement the app using the development (DEV) and user acceptance test (UAT) environments. In these environments, it's OK to debug the apps. But remember never to debug apps in the production (PROD) environment. Allowing debugging in production could affect the app's execution, 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 investigate a given problem in an app we consider to be running 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.

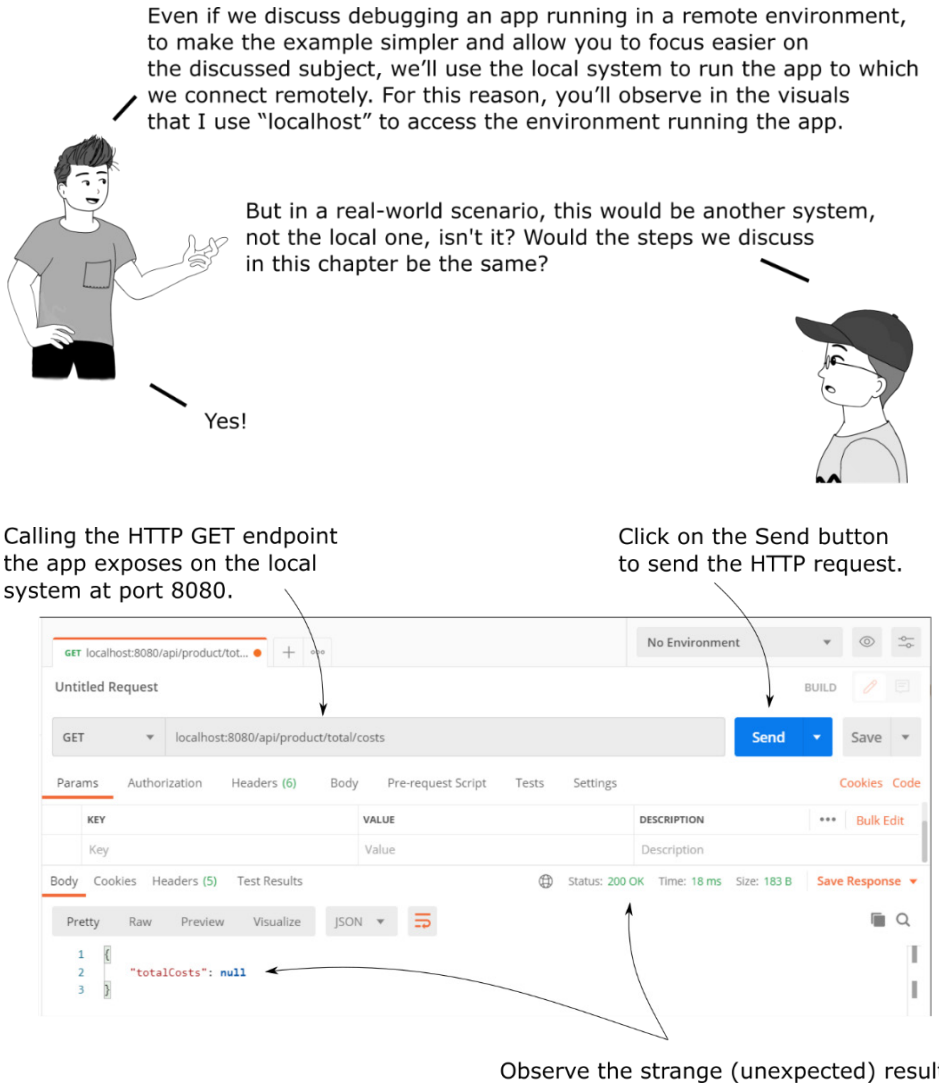


Figure 4.4 The scenario you have to investigate. The /api/product/total/costs endpoint should return the total costs from the database. Instead, when sending a request to the endpoint, the app behaves weirdly. The HTTP status is 200 OK, but the total costs, which you expected to be a list of values, 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, so, assuming it's configured to allow remote debugging, you would just need to connect to it. But, 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).

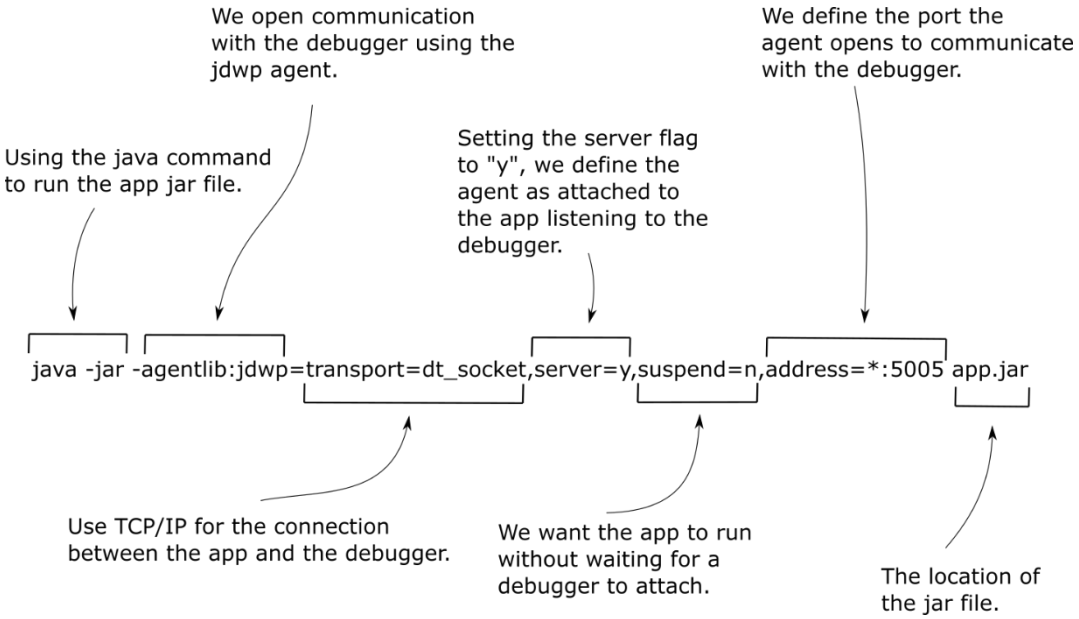


Figure 4.5 You have to attach a debugger agent to an app you want to debug. When debugging an app locally, the IDE attaches the debugger, and you don't have to worry about the debug agent. But when running an app in a remote environment, you need to attach a debugger agent at the app start yourself.

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.

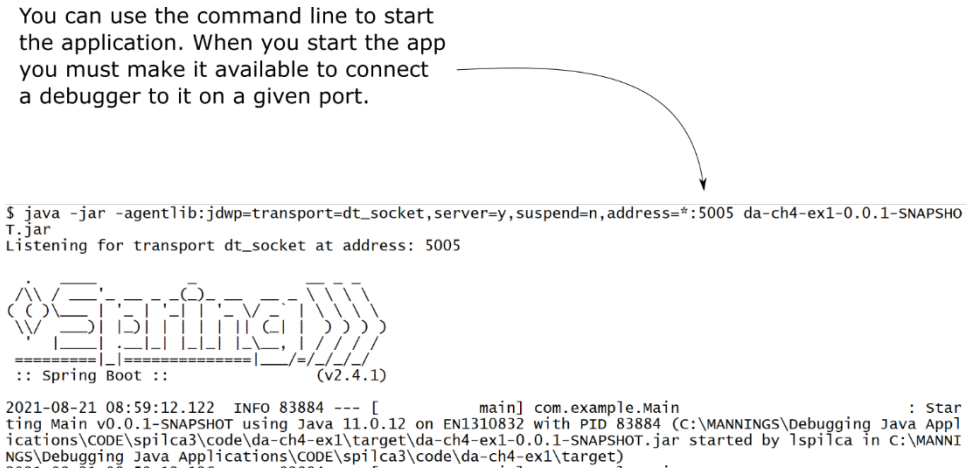


Figure 4.6 When you run the command to start the app, you can see the app begins executing. At the same time, you can observe in the console the debugging agent printed a line showing that it listens for a debugger to attach on the configured port 5005.

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. 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.

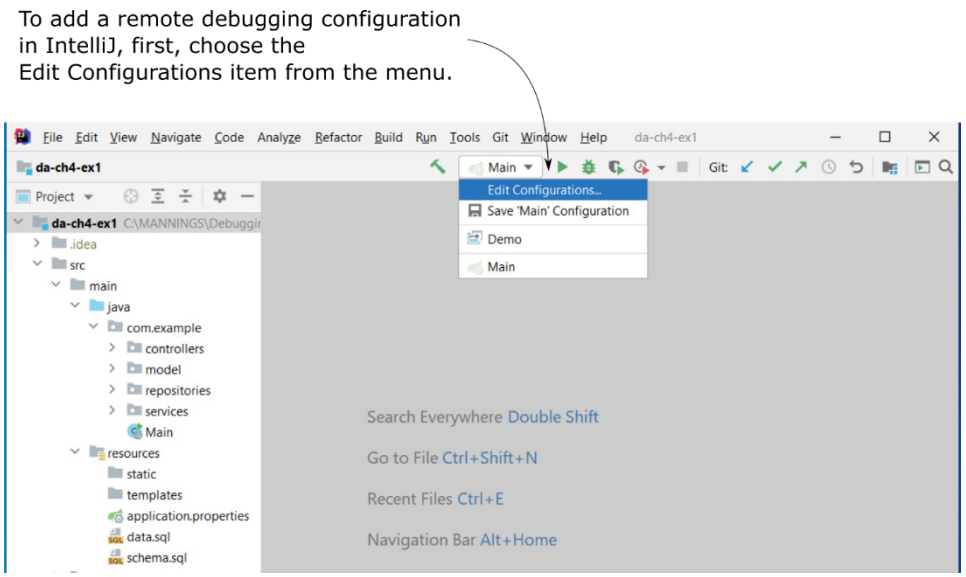


Figure 4.7 You can use an IDE to configure the debugger to attach to an already running app in a particular environment, as long as the app has a debugger agent attached to it. In IntelliJ IDEA, you need to create a new running configuration to tell the debugger to stick to an already running app. You 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.

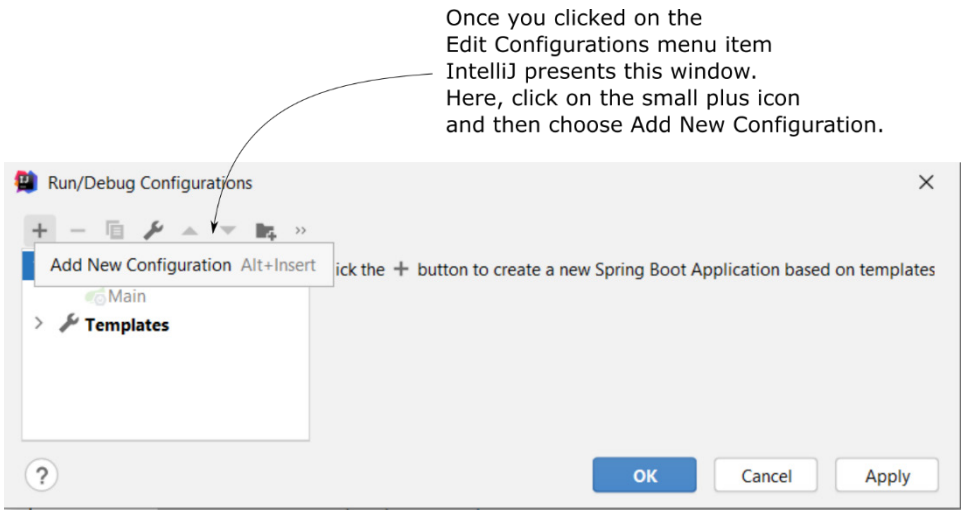


Figure 4.8 Once you selected Edit Configurations, you can add a new configuration. First, click on the plus icon and then Add New Configuration.

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.

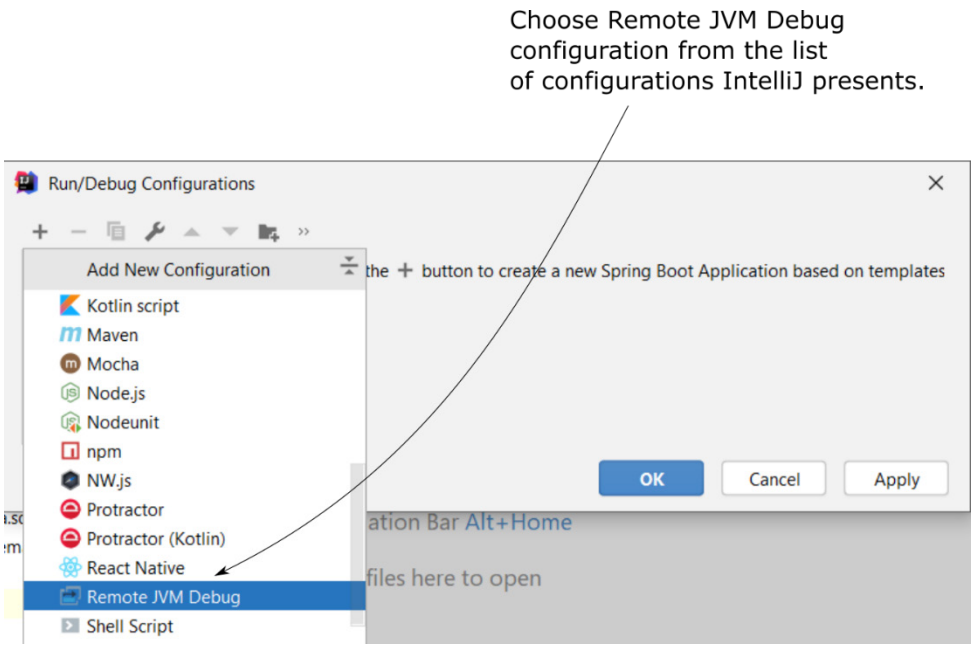


Figure 4.9 Since we want to attach to an app running in a remote environment, select the Remote JVM Debug configuration type.

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.

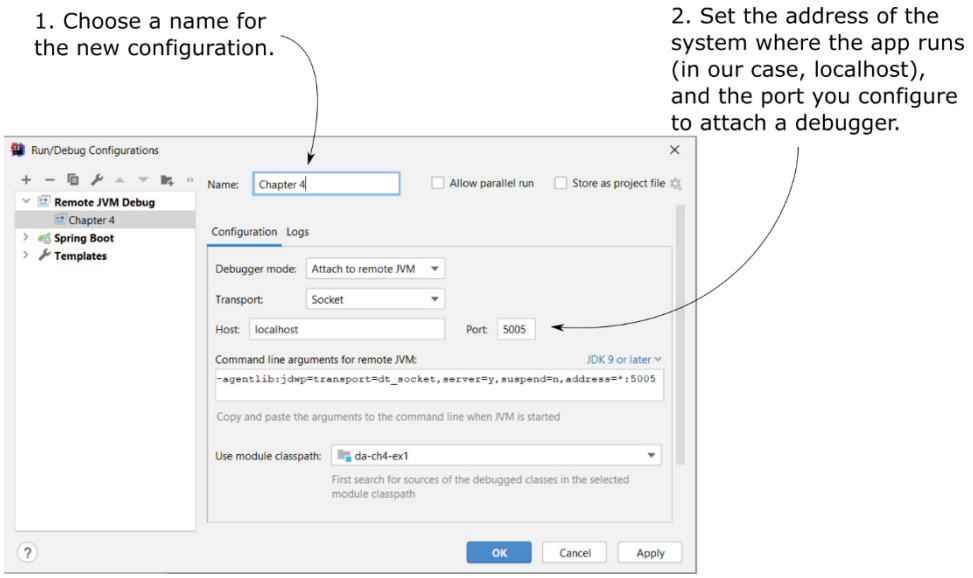


Figure 4.10 Give a name to the new configuration you add and specify the address of the environment and the port you configured the debugger agent to listen on. In our case, we configured port 5005 for the debugger agent when starting the app.

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).

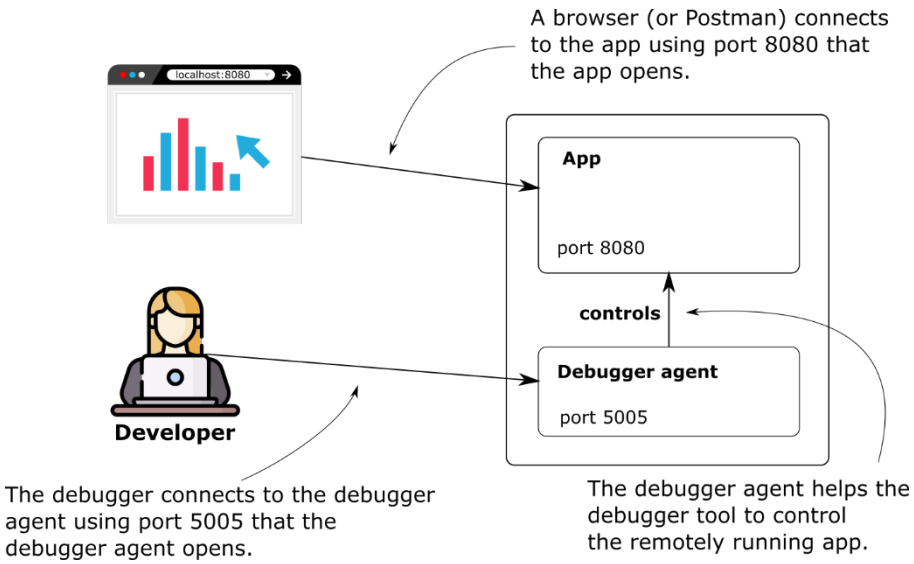


Figure 4.11 The debugger tool that runs on the developer's computer connects to the debugger agent on port 5005. The debugger agent helps the debugger tool control the app. The app also opens a port, but this port is for its clients (the browser in the case of a web app).

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.

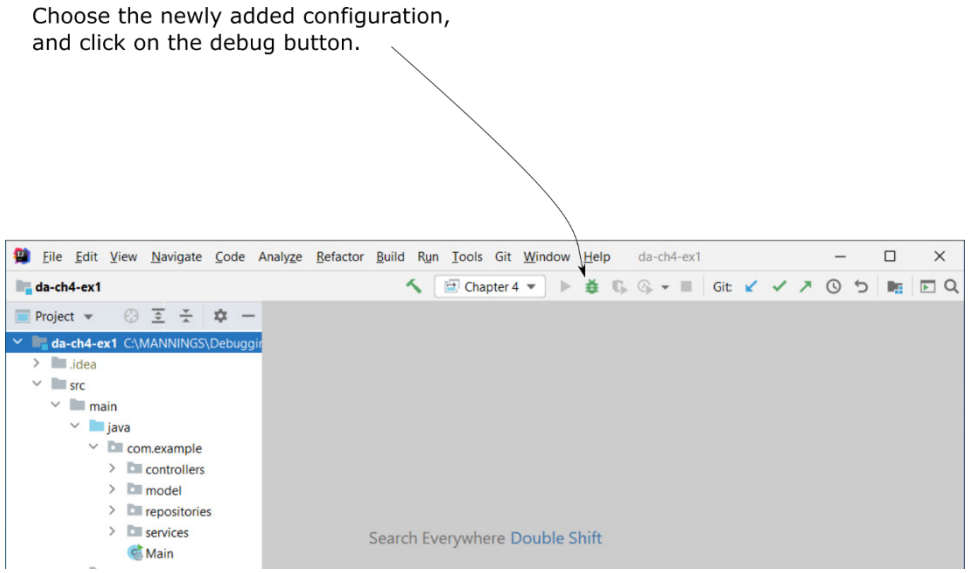


Figure 4.12 You can run the debugger now using the newly added configuration. Click on the small "bug" icon to start the debugger.

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.

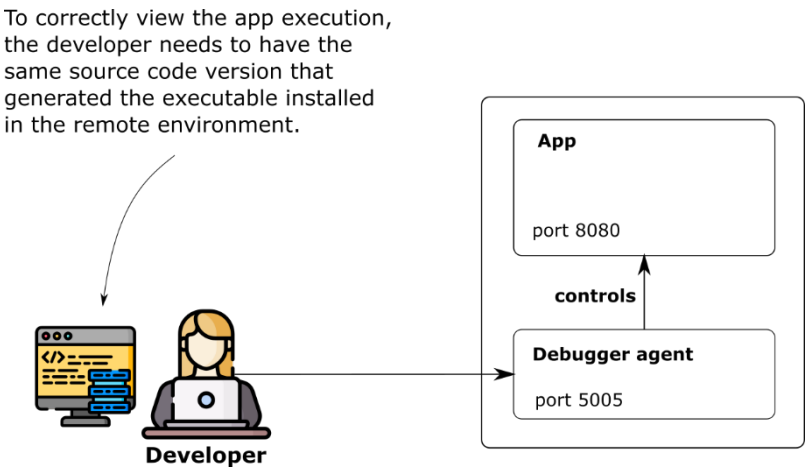


Figure 4.13 The developer needs to make sure they have the same version of source code as the one used to generate the app's executable running in the remote environment. Otherwise, the debugger's actions could become inconsistent with the code the developer investigates and would create more confusion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

NOTE Ensure you always check out the same source code version that created the app you are remotely debugging. If the source code differs, the debugger tool might act weirdly and confuse you more than help you solve the issue.

Here's an exercise for you!
Take a break from reading here
and try to solve the problem.

Which is the cause of the apps strange behavior,
and how would you solve the problem?

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.

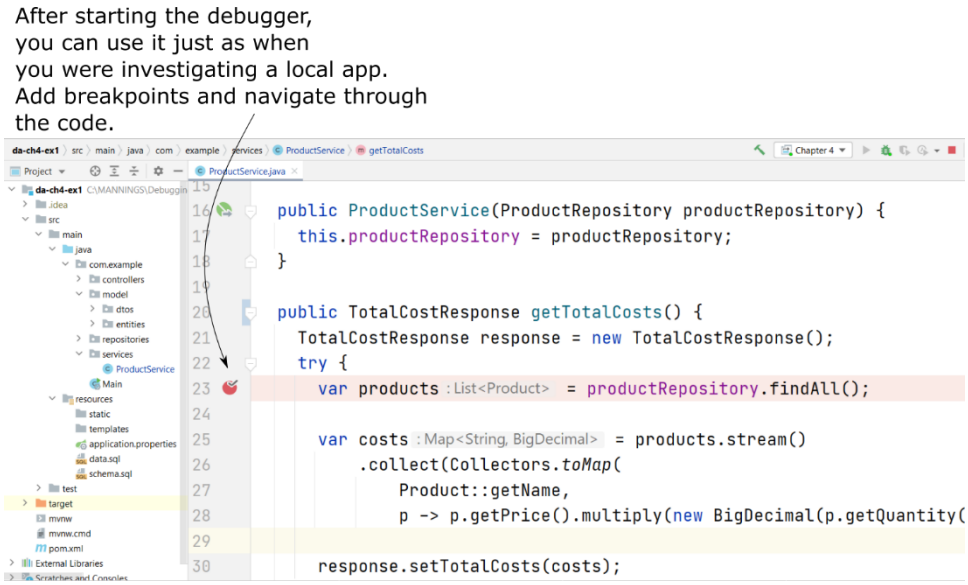


Figure 4.14 Just as when debugging an app locally, you can add breakpoints and use navigation operations. Add a new breakpoint on line 23 in the ProductService class, as presented in this figure.

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.

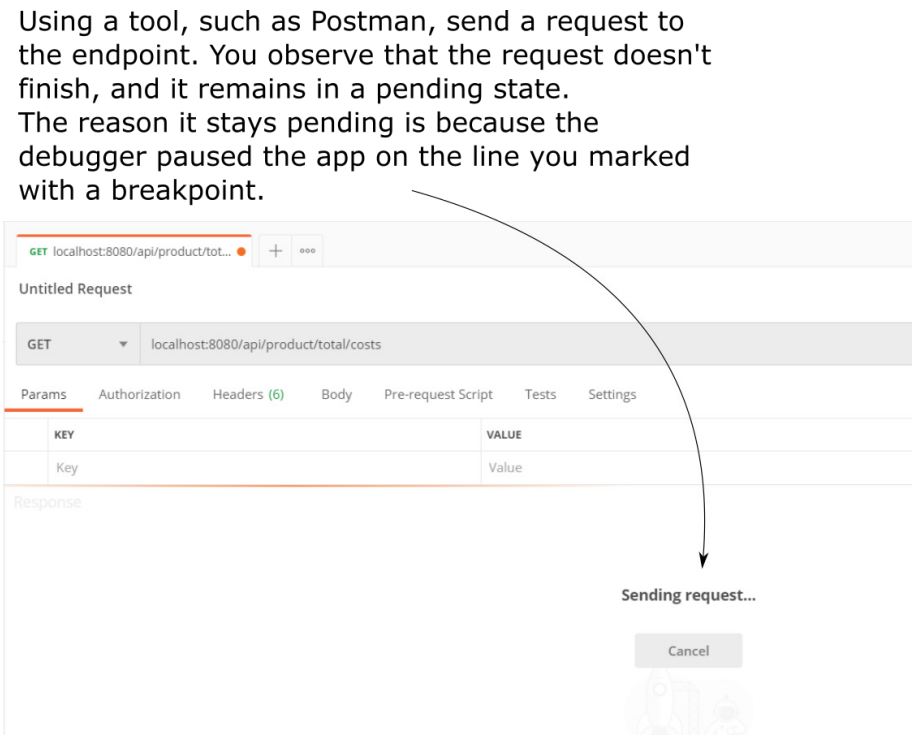


Figure 4.15 When using Postman to send the request, you observe the response doesn't come back immediately anymore. Instead, Postman seems to wait indefinitely for the response. The reason is: the app paused the execution on the line you marked with a breakpoint.

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.

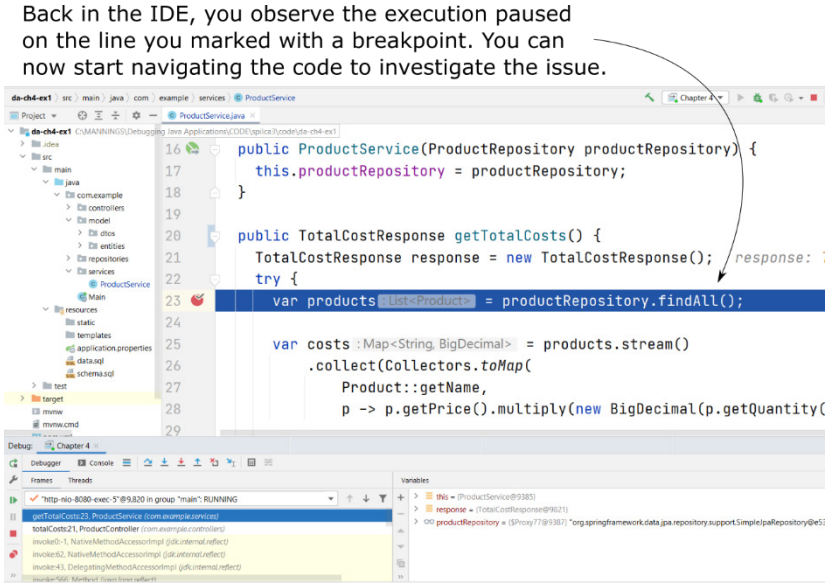


Figure 4.16 When you check the IDE, you observe the debugger indeed paused the execution on the line you marked with a breakpoint. Thus, you can use navigation operations to continue your investigation.

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.

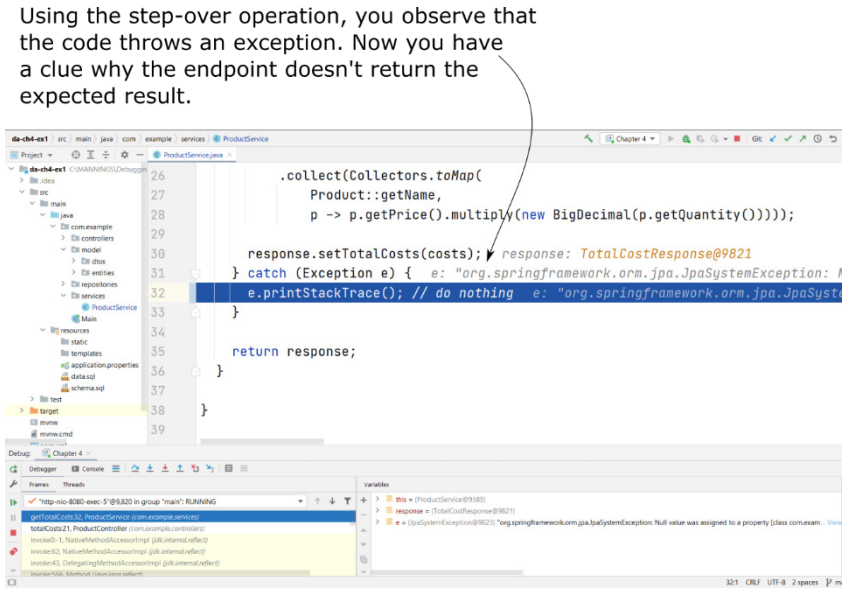
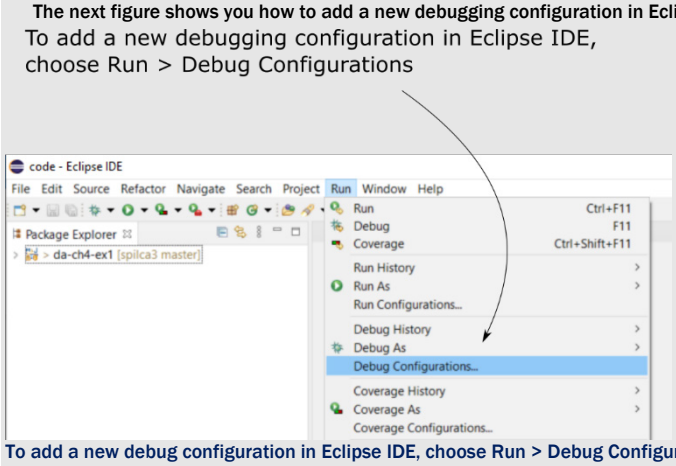


Figure 4.17 When using the step-over navigation operation, you observe the app throws an exception. Now you have an idea of what the problem is and decide further on how to solve it.

Creating a remote configuration in Eclipse IDE

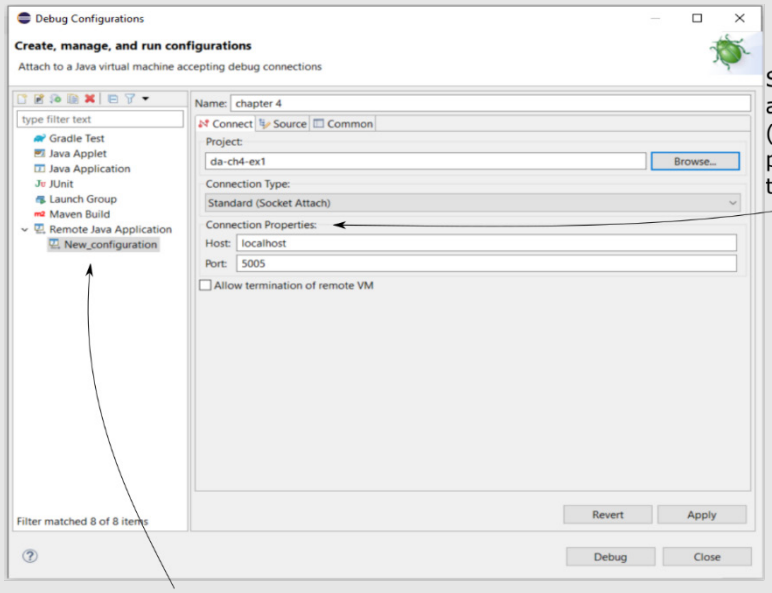
I use IntelliJ IDEA as the primary IDE for the examples of the book. But as I stated in earlier chapters, this book isn't about using a certain IDE. You can apply the techniques we discuss in this book with a variety of tools of your choice. For example, you can do remote debugging with other IDEs, such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



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.

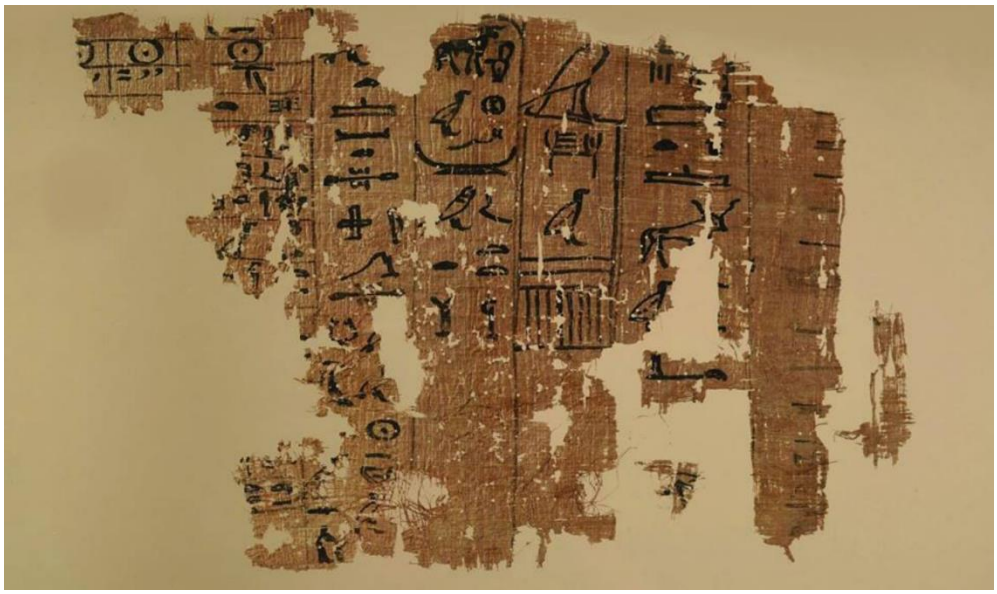


Figure 5.1 The oldest known logbook. It describes the process of building the Great Pyramid of Giza (Cairo, Egypt).

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.

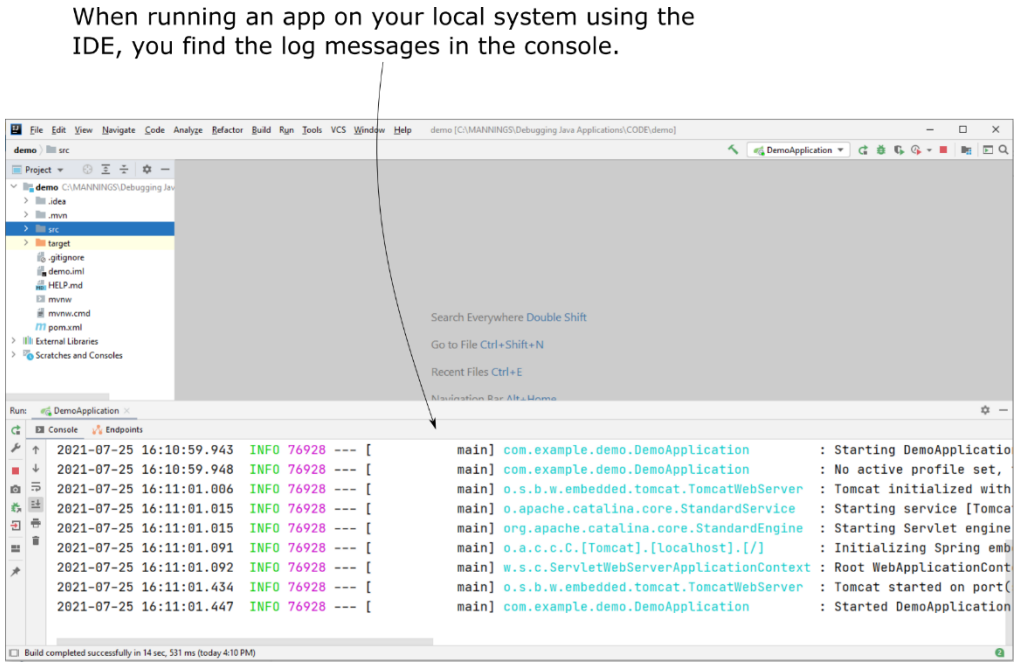


Figure 5.3 IDE log console. Any IDE shows a log console. While logging messages in the console is useful when running the app locally, real-world apps also store logs to use them if you need to understand how the app behaved at a given time.

Figure 5.3 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 the wrote the message (figure 5.4).

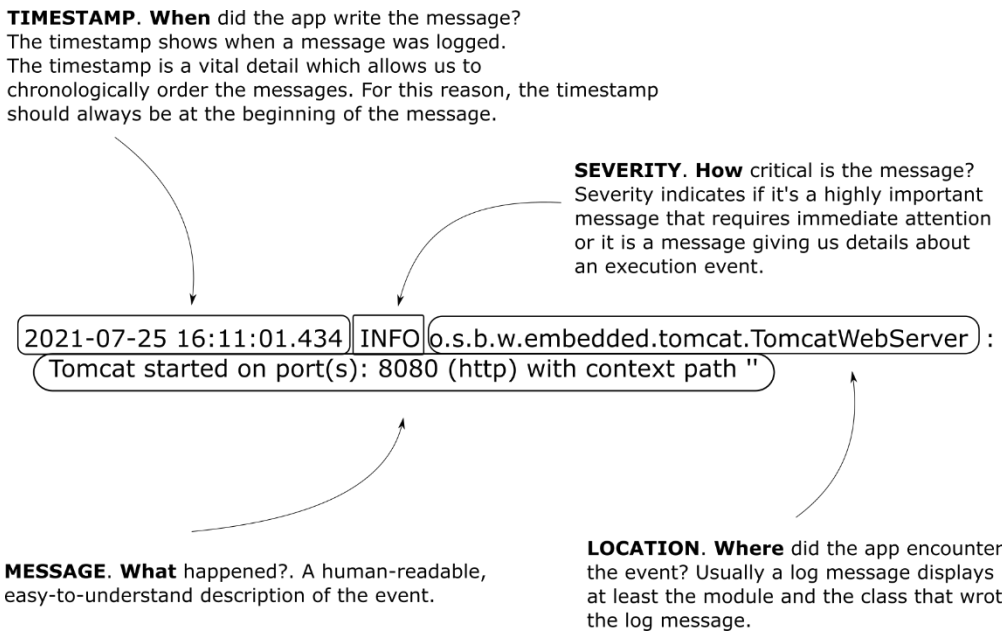


Figure 5.4 The anatomy of a well-formatted log message. Besides describing a situation or an event the app encountered, a log message should also contain several other relevant details. Any log message should contain at least the timestamp when the app logged the message, the event's severity, and where 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
- Rising 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!

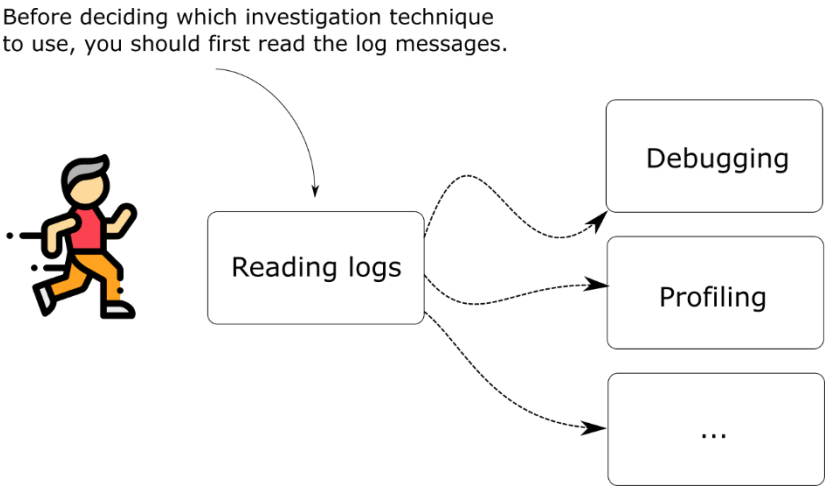


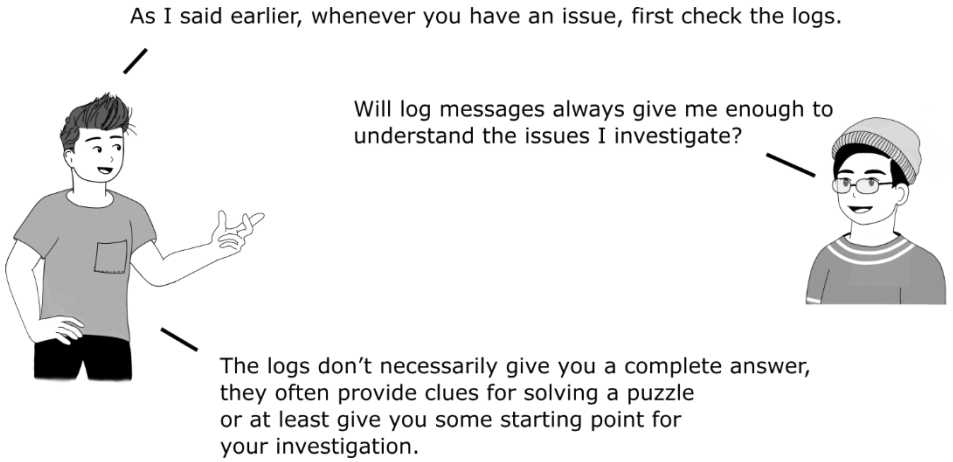
Figure 5.5 Whenever you investigate an issue, the first thing you do should always be reading the app's logs. In many cases, the log messages give you a starting point. Even if they can't always directly answer all the questions, they often give you valuable hints on what you should do next to solve 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 a 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, 2021). 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 with 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 your spend investigating a particular issue or behavior. In this section, we discuss various scenarios where using logs help you understand software's behavior easier. We'll begin with discussing several key points of log messages and then analyze why these characteristics help investigate various scenarios.



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 de 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.

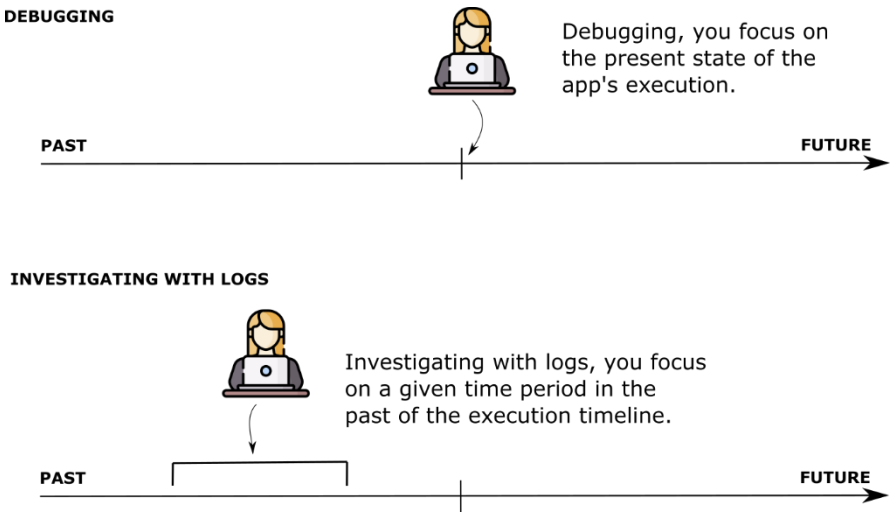


Figure 5.6 When investigating an issue with the debugger, you focus on the present. When you do the same thing with the log messages, you focus on a given period in the past. This difference is one of the details you'll consider when deciding whether to use logs or the debugger.

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).

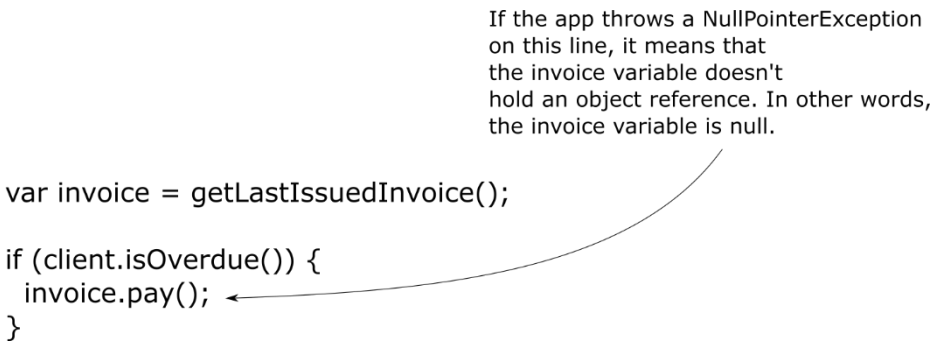
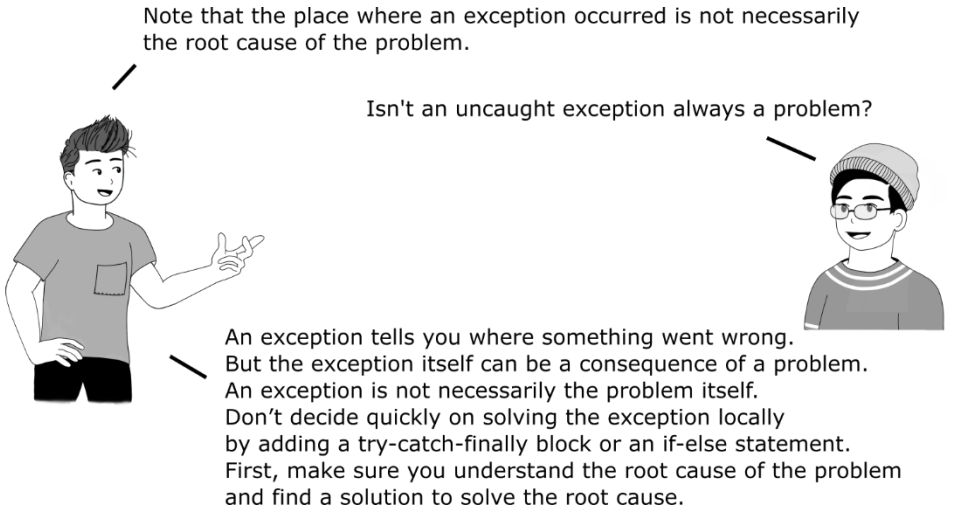


Figure 5.7 A NullPointerException indicates the app execution encountered a situation where a behavior had to be called without the behaving instance. But it doesn't mean that the line that produced the exception is also the cause of the problem. The exception could only be a consequence of the root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

You should remember that the place where an exception occurred is not necessarily the root cause of the problem.



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).

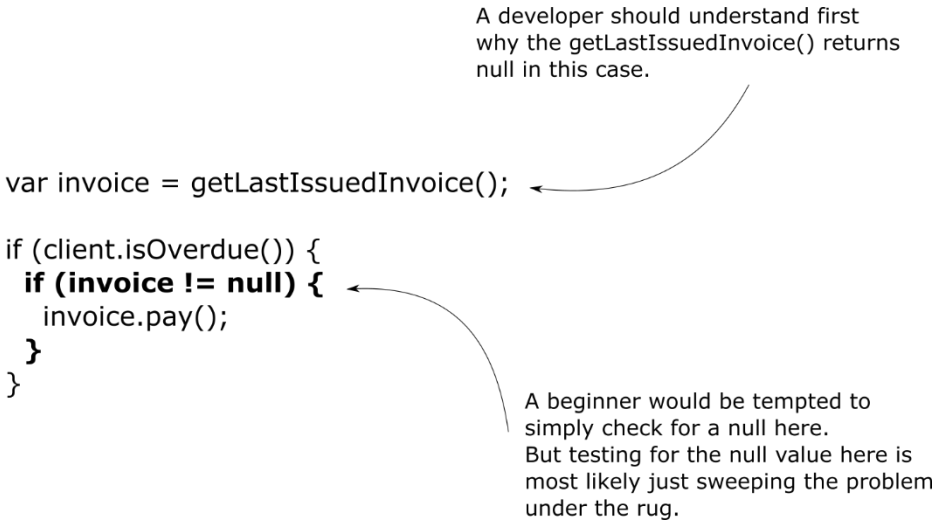


Figure 5.8 Locally solving the problem is in many cases equivalent to sweeping it under the rug. Without solving the root cause, more issues could appear later. Always look for the root cause and remember that an exception you find in the logs doesn't necessarily indicate the root cause.

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 Logging the execution time for a certain line of code

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

  long timeBefore = System.currentTimeMillis();    #A
  var products = producRepository.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.

In chapter 12, we'll discuss troubleshooting with stubs. At that time, we'll also discuss some alternative techniques that could help you in situations similar to this one.

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).

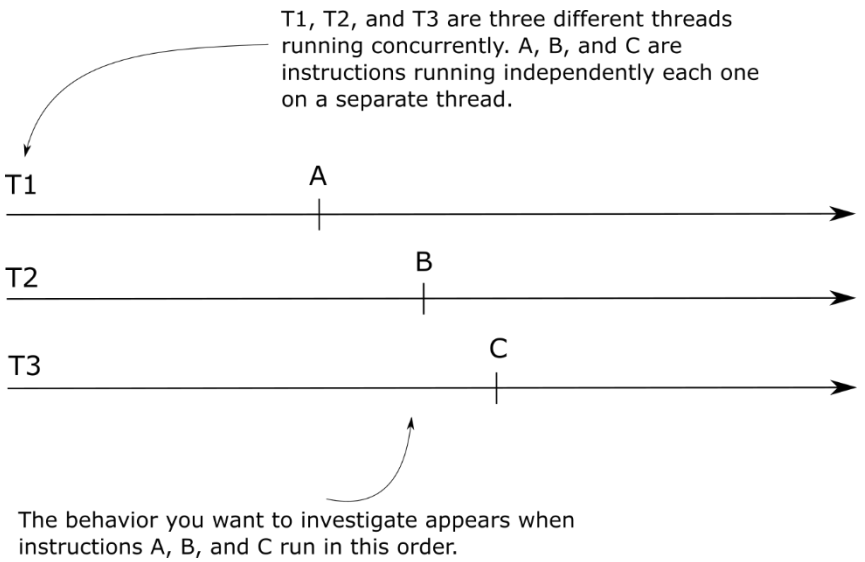


Figure 5.9 A multithreaded architecture. An app capability using multiple threads running concurrently to process data is a multithreaded app. Unless explicitly synchronized, instructions running on independent threads (A, B, and C) can run in any order. Suppose you want to investigate a certain behavior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

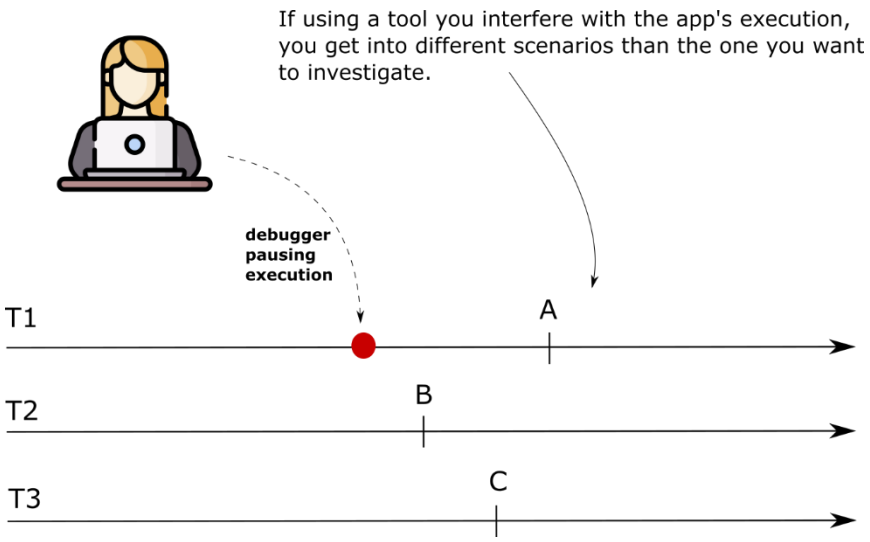


Figure 5.10 When using a tool such as a debugger or a profiler, the tool interferes with the execution, making some (or all) threads slower. Because of this, the execution often changes, and some instructions may execute in a different order than the scenario you wanted to investigate. In such a case, 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.

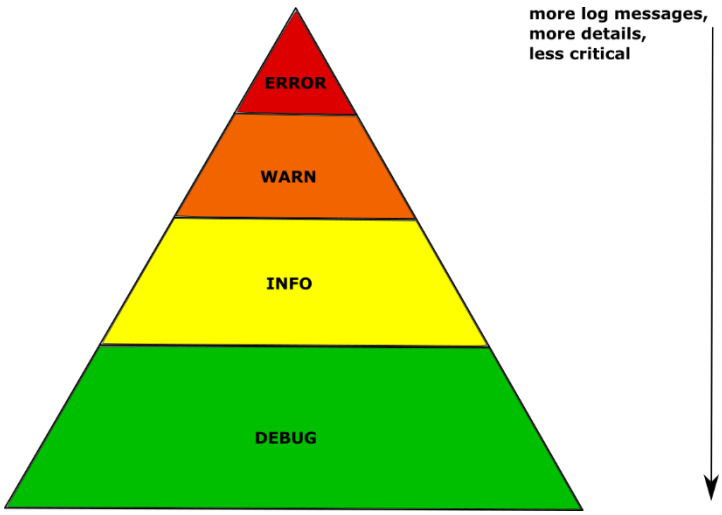


Figure 5.11 The log severity pyramid. On the top are the critical log messages that usually require immediate attention. The bottom represents the log messages that only represent details you'll rarely need. From the top to the bottom, the log messages become less essential but bigger in number. Usually, 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 instrance 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.

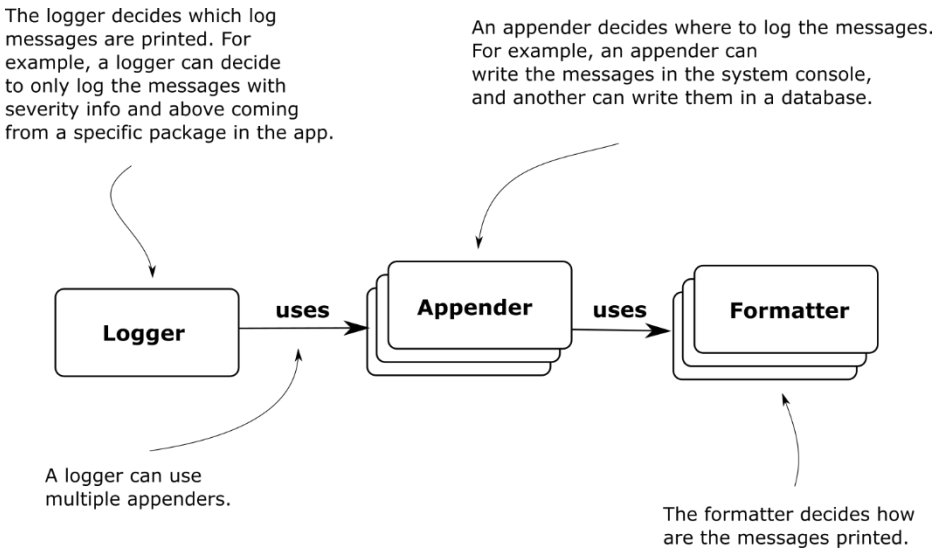


Figure 5.12 The relationship between the appender, logger, and formatter. A logger uses one or more appenders. The logger decides what to write (for example, only log messages printed by objects in the package com.example). The logger gives the messages to be written to one or more appenders. Each appender 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.

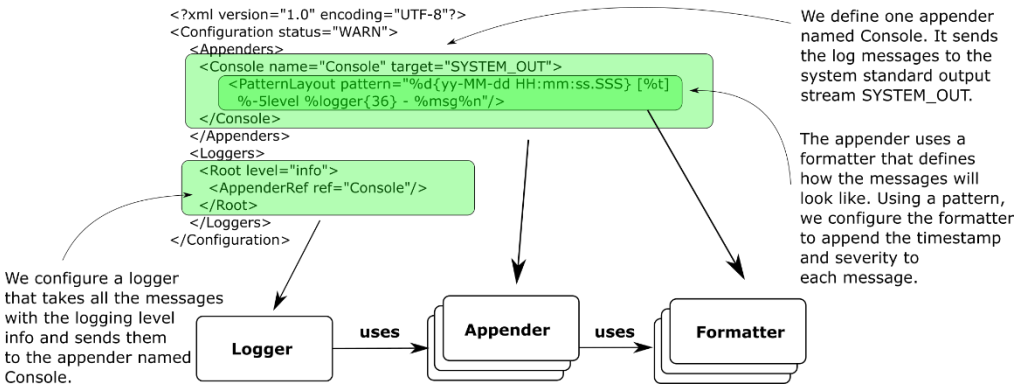


Figure 5.13 The components in configuration. The logger Root takes all the log messages with severity info the app writes. The logger sends the messages to the appender named Consoler. The appender Console is configured to send the messages to the system terminal. It uses a formatter to attach the timestamp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

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).

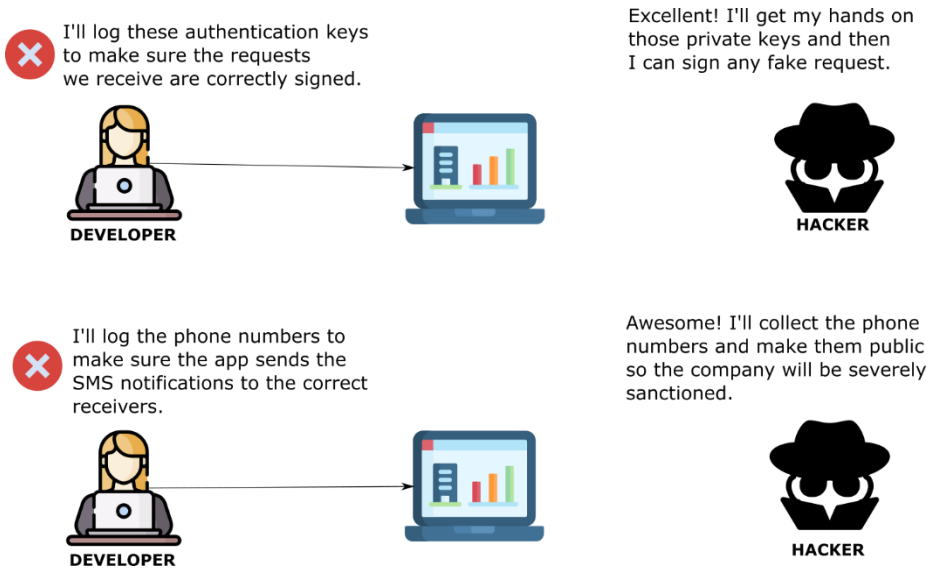


Figure 5.15 Log messages should not contain secret or private details. No one working on the app or the infrastructure where the app is deployed shouldn't access such data. Exposing sensitive details in logs can help a malicious person (hacker) find easier ways to break the system or attract severe sanctions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

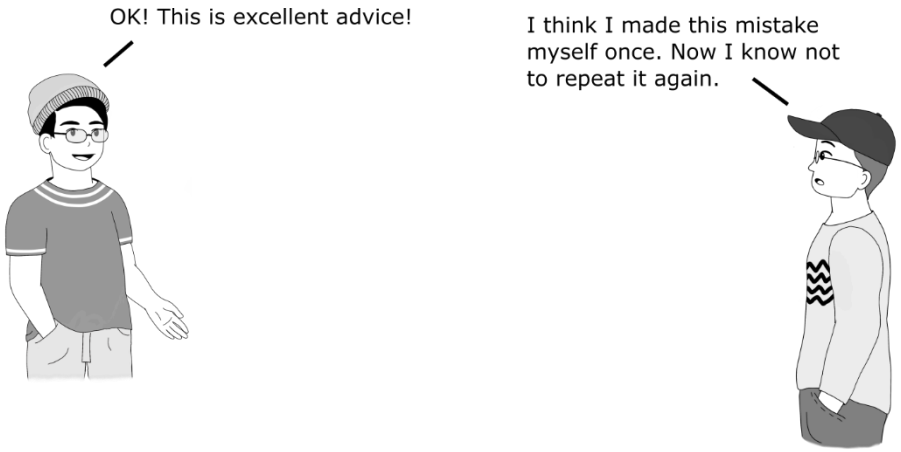
The following snippet shows some examples of log messages that expose sensitive details and cause vulnerabilities.

```
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 protected 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

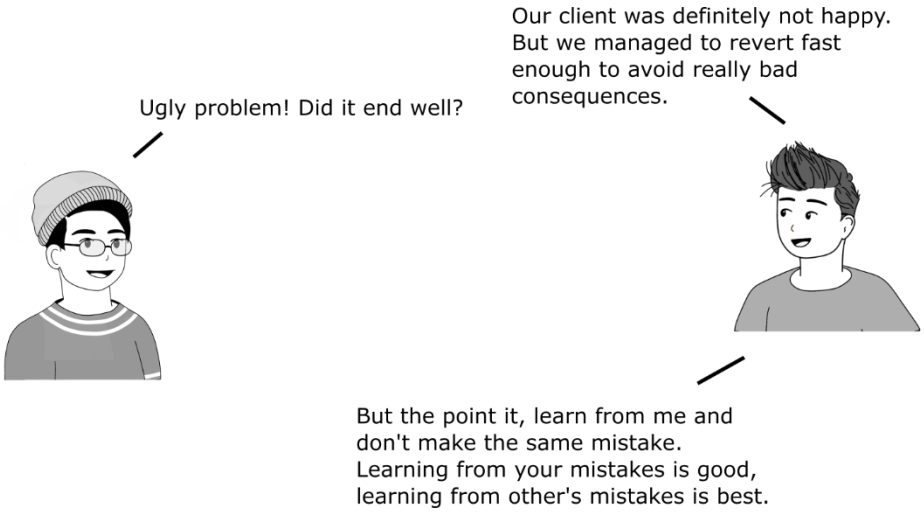
such details belonging to individuals. For example, the European Union applies a set of regulations named General Data Protection Regulation (GDPR) starting with May 2018. An application with users in any European Union state must comply with these regulations to avoid severe sanctions. Throughout various 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 needed 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:

- 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 this section, we discuss the differences between using logs and remote debugging in investigating specific app behavior. 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 logging 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.

Appendix 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/

Appendix 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**.

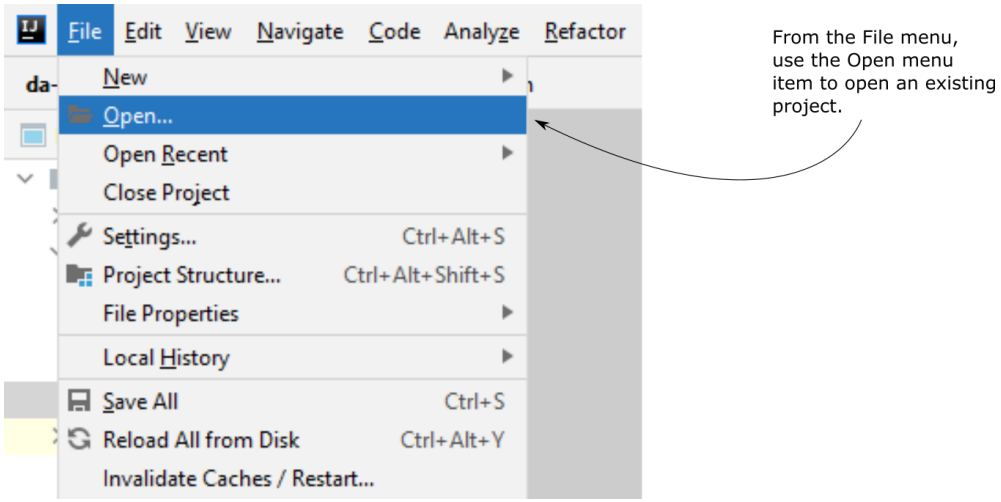


Figure B.1 To open an existing project in IntelliJ IDEA, use the Open menu item in the File menu.

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.

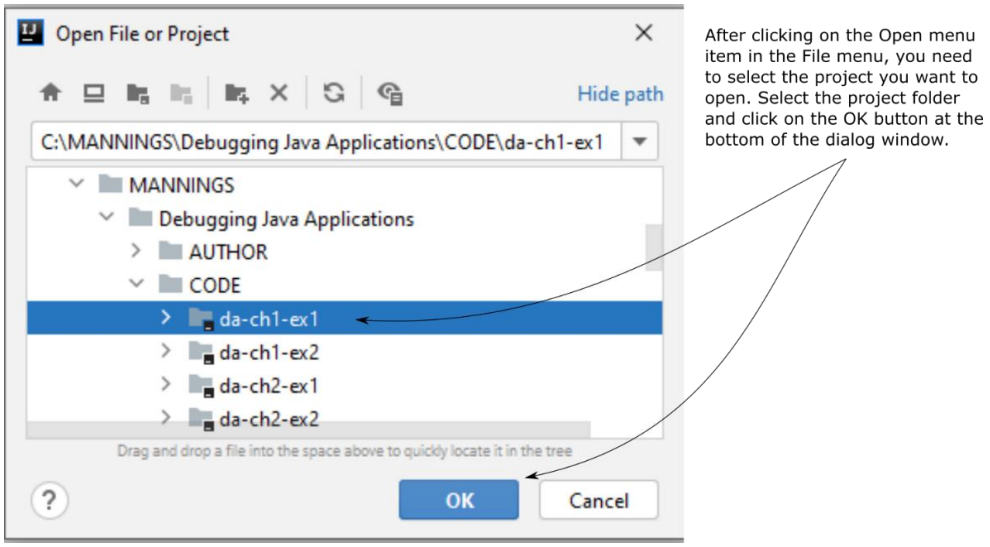


Figure B.2 After clicking on the Open menu item in the File menu, a popup window appears. In this window, select the project you want to open from the file system and click on the OK button.

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**.

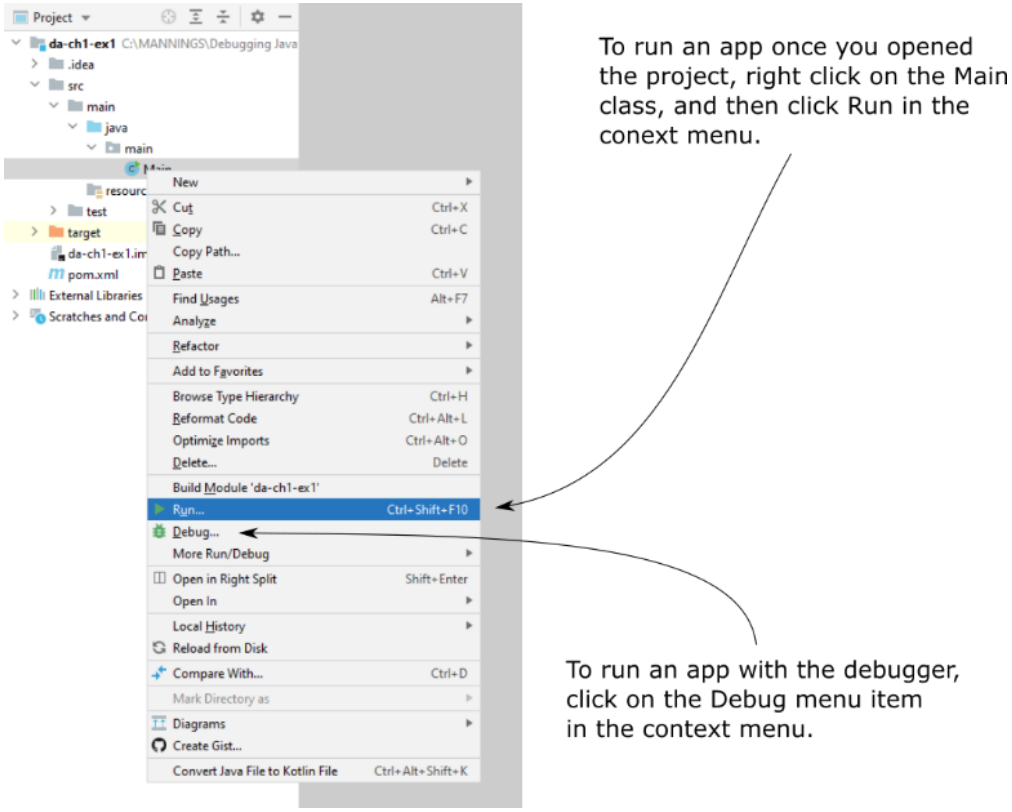


Figure B.3 Once you opened an app, you can run it. To run the app, right click on the Main class and the select the Run menu item. If you want to run the app with a debugger, click on the Debug menu item.