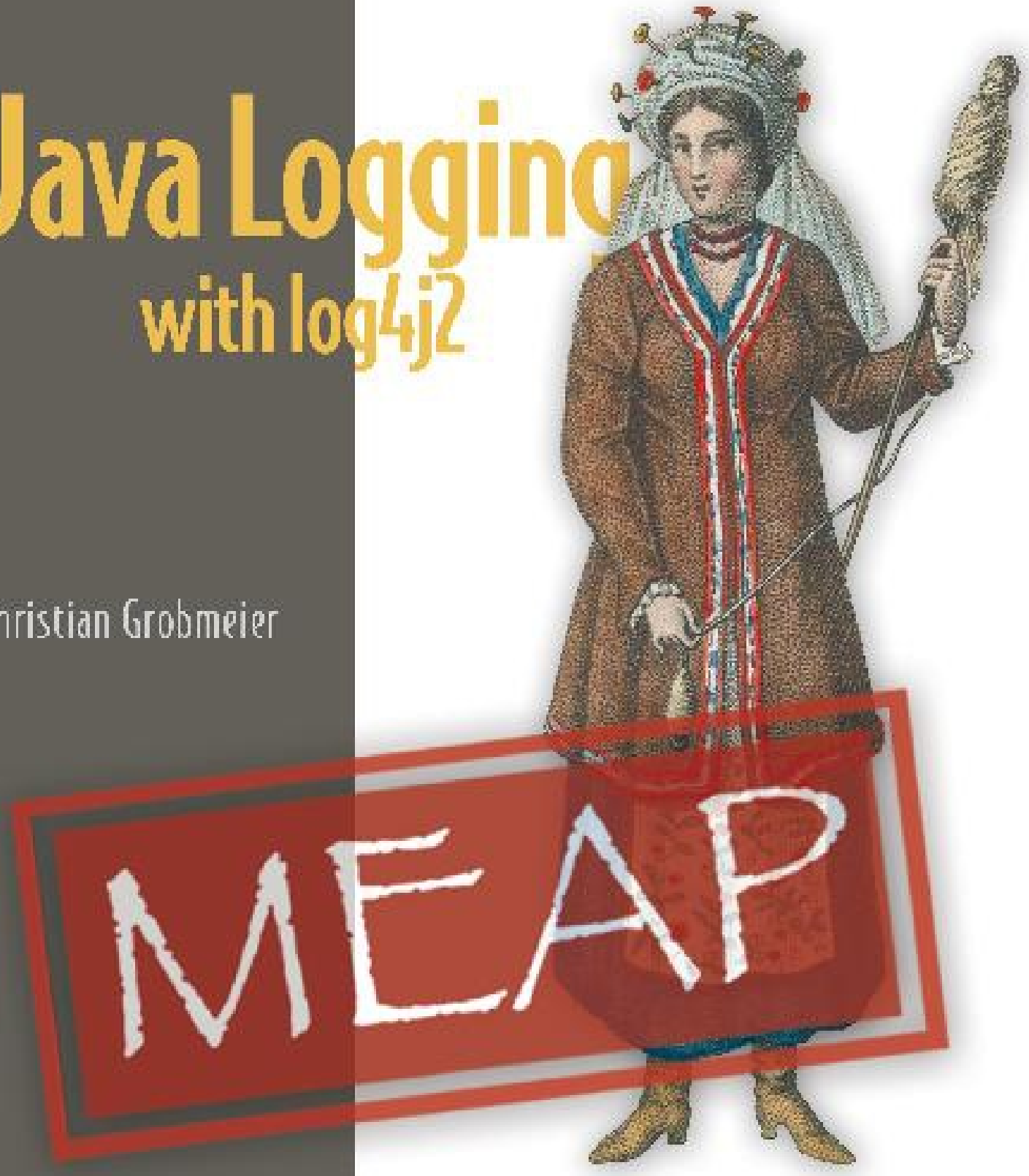
Java Logging with log4j2

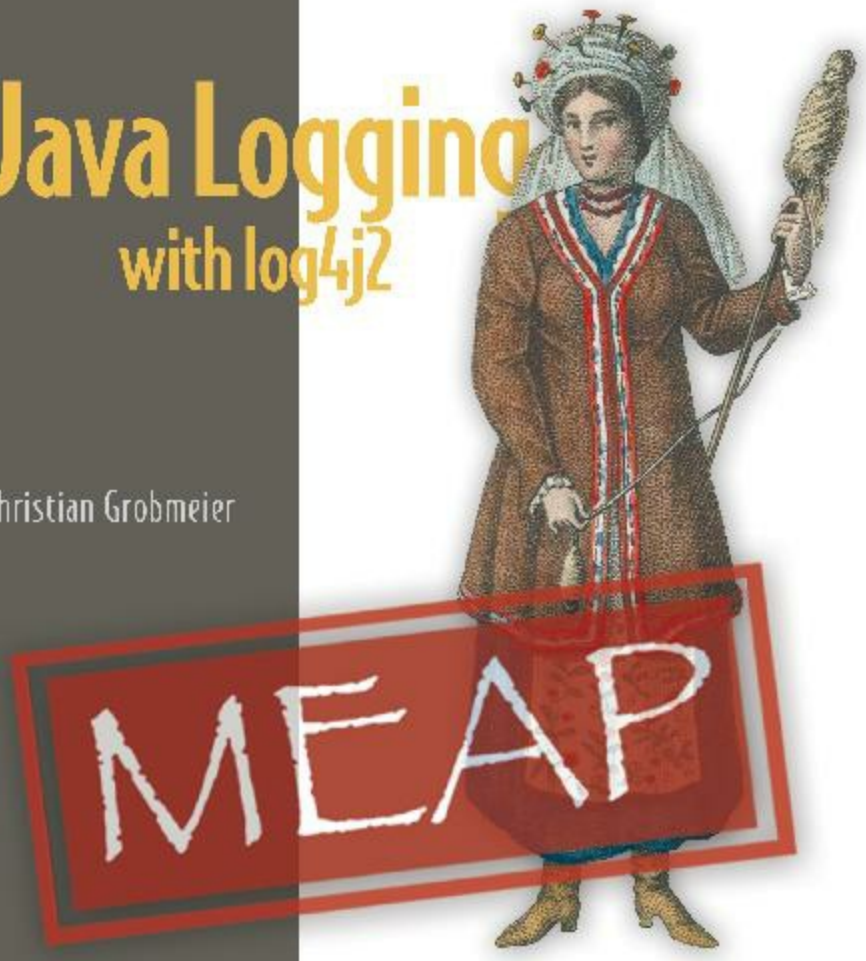
Christian Grobmeier



MANNING

Java Logging with log4j2

Christian Grobmeier



MANNING

Java Logging

Welcome

Thank you for purchasing *Java Logging with log4j2*.

Many years ago, I was a junior developer. My senior developer introduced the concept of logging to me. A few months later, I heard him explain the same ideas to another junior developer. Eventually, it was my turn to explain. What if everyone explaining logging had a book like this one to share? I imagine my mentor giving me a worn copy. He'd say, "Read it, we'll talk about it tomorrow."

It was also my mentor who sparked my interest in open source. I wanted to contribute code too, and what could be more natural than working on logging itself? I became a member of the Apache Log4j core team in 2009, and I am still there.

When Log4shell, a major security incident, hit the software industry in 2021, I didn't leave the project— quite the opposite. The incident opened my eyes to the importance of open source and logging, and I was reminded how essential it is to truly understand the tools we rely on.

Those challenges inspired this book—a guide that seniors can pass down to juniors. If you're reading this as a senior developer, I hope it prompts you to think: "Yes, that's what juniors need to know!" And if you're a junior developer, I hope you find it engaging and straightforward to read.

Programming is complicated enough. This book's goal is not to add to that complexity but to make logging easy to apply and accessible.

There are numerous options available for *Java Logging with log4j2*, and this book aims to explain the basics first, introduce Apache Log4j, and then compare it with other popular choices. At the end of this book, developers will be able to decide between them and—one day, hopefully—pass this book on to their juniors developers. Until then, enjoy the book.

Please share your thoughts in the liveBook's Discussion Forum. Your

feedback will help shape the final book—and I’m always grateful for questions and for hearing about your experiences.

—Christian Grobmeier

In this book

1 First steps with Java logging

This chapter covers

- Applying logging to your application
- Understanding log messages
- Introducing popular logging products

When computers had little CPU power and memory compared to today, we had two versions of software: the one that worked and the one that failed. If something broke, we could not see what went wrong. Fixing issues was expensive and time-consuming. The only silver lining? We got paid for the overtime.

At some point, programmers had the idea to let software write each step it took into a file. Now, we could see what was happening in real time. We could understand issues better and fix them sooner. Of course, no more overtime.

Today, developers optimize logging systems for speed, reliability, and flexibility. Now, we can write logs in production systems and monitor certain software behaviors. Logging systems have become a critical part of software development-but what is it?

1.1 What is logging?

Logging is the art of making hidden things visible. In a nutshell, we need logging to:

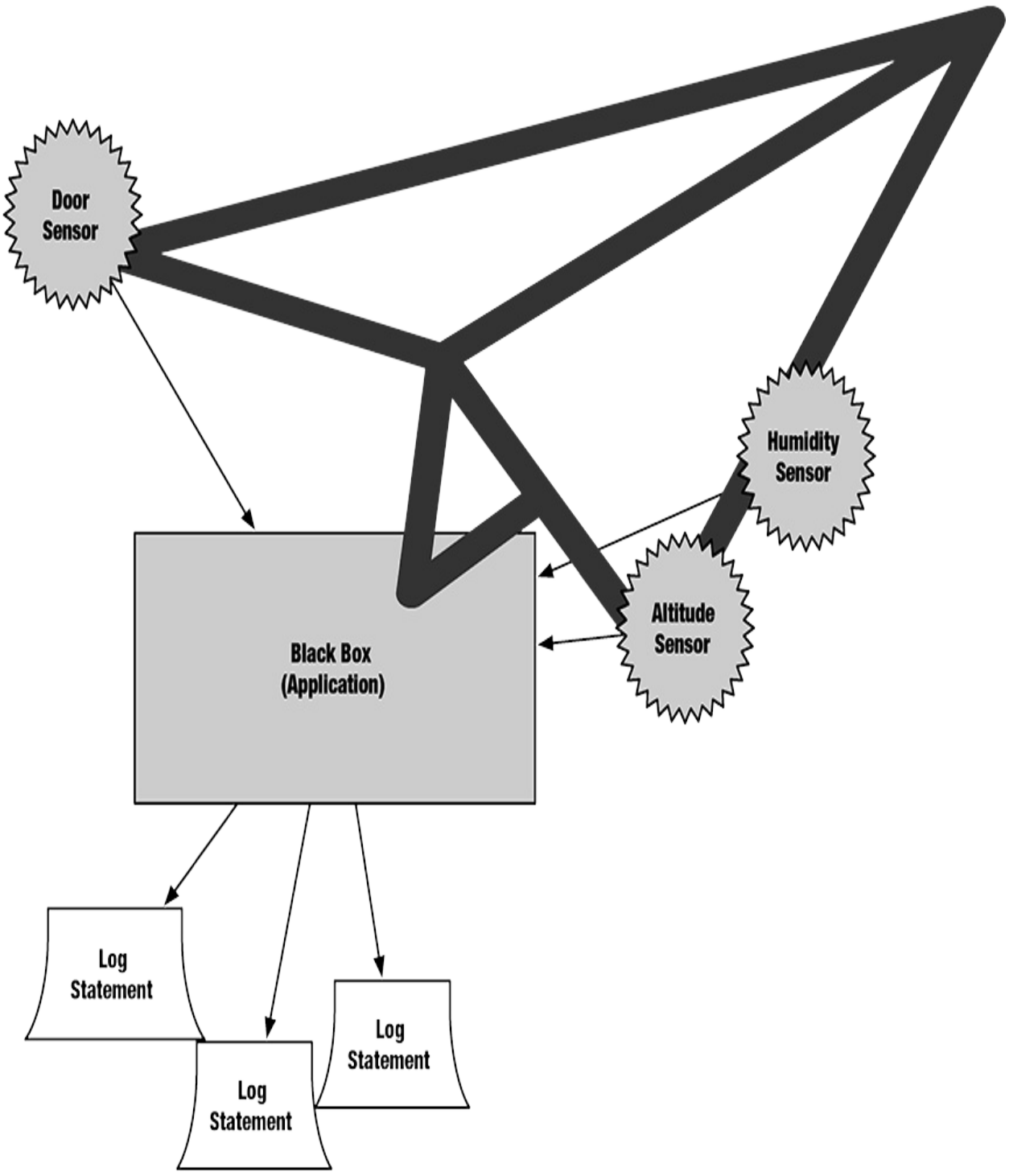
- Monitor the System: Logs reveal memory leaks and unexpected behavior before customers encounter problems.
- Understand Application Behavior: Optimization is only possible when we know what is happening.
- Debug Problems: Logging provides valuable insights when something goes sideways.

- Tune Performance: Identifying slow-running processes and bottlenecks can reduce the time and resources needed for tasks, saving costs.
- Perform Audits: Logs prove that our system performed the correct actions and ensure compliance.

This list was pulled straight from a boring university lecture. Luckily, logging plays a critical role in real-world systems. Let us look at an exciting one.

If you imagine an aircraft, think about all the built-in sensors. They track everything, more than any social media network, from altitude and speed to fuel levels and external temperatures. The sensors are relevant for the pilots to fly safely and are directly visible to them in the cockpit. Pilots need real-time information to make quick decisions. Others need that information after a flight: ground personnel. Instead of relying on the pilot's memory, ground crews could retrieve sensor data from the black box, which is like an extra-safe hard disk for logging data. This black box is not just a hard disk or a physical device for us developers–it is our application. The application records these events as *log statements*—readable messages that capture what happened. Figure 1.1 shows how a plane's sensors send data to the black box, which generates log statements.

Figure 1.1 Sensors feed data into an application (black box), which generates log statements.



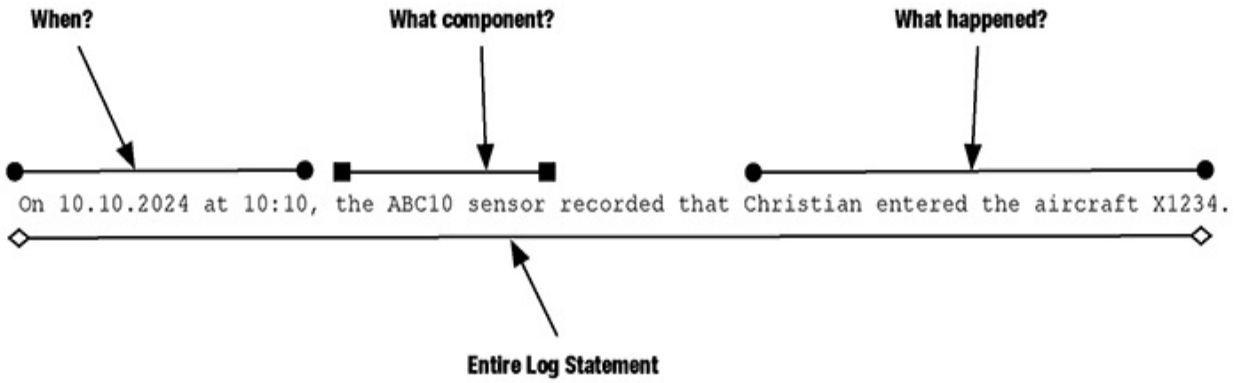
Both humans and machines read log statements. Think of a log statement as a simple sentence, such as:

```
"On 10.10.2024 at 10:10, the ABC10 sensor recorded that Christian
```

In reality, log statements are often more technical and less human-readable. However, the core information stays the same no matter how complex it looks.

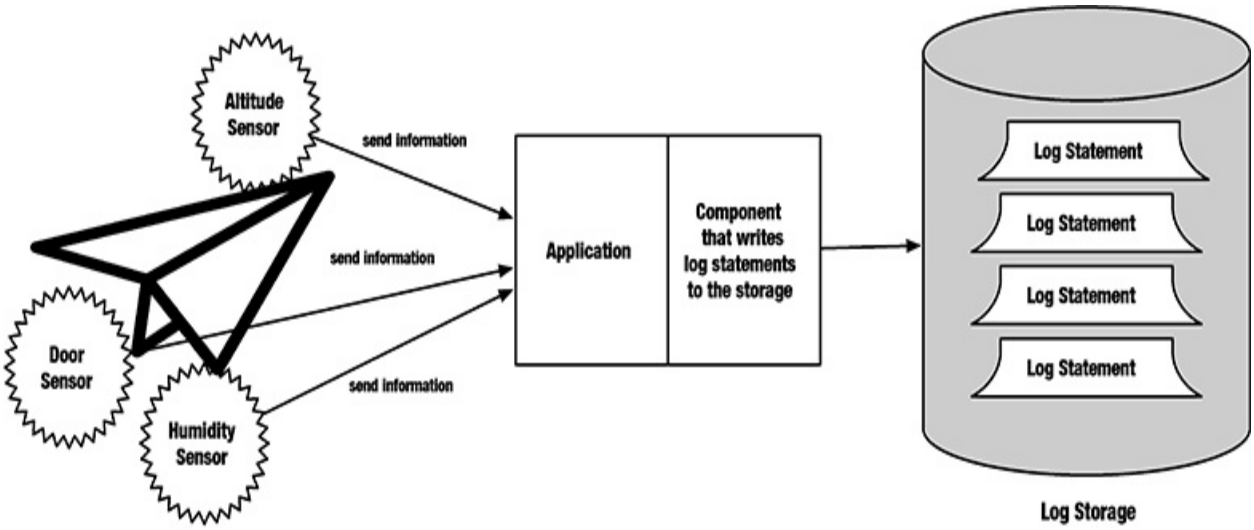
A log statement is more than just a sentence. It follows a structure. Figure 1.2 breaks it down into three key parts: when the event happened, which component recorded it, and what occurred.

Figure 1.2 The structure of a log statement—when, what component, and what happened.



Like the aircraft's black box, computer systems similarly store log statements. Statements are appended individually, as they occur, in simple files–*log files*. Sometimes, they are even stored in specialized database systems. I prefer the term *log storage* over log file. Figure 1.3 shows how log storages fit into our architecture.

Figure 1.3 Log statements are stored in a storage, often a log file.



1.2 Who benefits from your log files?

Now that we know logging records what's happening, the question becomes: What should you log? Who might be reading what you write? The sad truth? Some poor soul will try to debug an issue long after you're gone. And sometimes, that poor soul is you, a few years older. A version of you, trying to remember the specific code you wrote.

George Santayana once said, "Those who cannot remember the past are condemned to repeat it." He wasn't talking about software, but the lesson applies. Logs are history books for your application, letting you revisit the past when you need answers.

Developers are not the only ones who check log files. Logs serve many roles because different people ask different questions. Developers want to understand what's happening under the hood—so they look at debug logs. Others, like operators or auditors, are more interested in actions than internals—they want audit logs to answer what happened, not how. Table 1.1 shows some examples of how logs answer different users' questions.

Table 1.1 Questions different people ask and their answers.

User	Question	Answer

Developer	Which parameters did this query use?	Selected from 'users' table in database XYZ with username 'grobmeier'.
	Why did this calculation fail?	IndexOutOfBoundsException - remember, arrays are zero-based!
Operator	How long did it take to connect to the 3rd party system?	Connection completed in 100ms.
Auditors	Who sent the money yesterday?	List of transactions:…
	When was invoice 1234 sent?	Sent to Mrs. Lili on October 31st.

It’s not just about knowing who reads your log files. Logging forces you to decide what information is worth recording—a daunting task. You also have to anticipate what might be needed in the future.

Things you should never log

Some things should never be logged. Never log personal data like email addresses, usernames, or social security numbers. And never, ever log passwords—whether in plain text or hashed.

Privacy and security are critical, and as software maintainers, we can’t predict who might access our log files in the future. Logging sensitive data can cause privacy breaches, regulatory penalties, and loss of customer trust.

1.3 The 6+2 questions to ask

Writing has always been a creative process, even or especially for developers. We developers don't need to write poetry. We need to write, objectively, and concisely. That's easier said than done—especially when you don't know where to start or what to include.

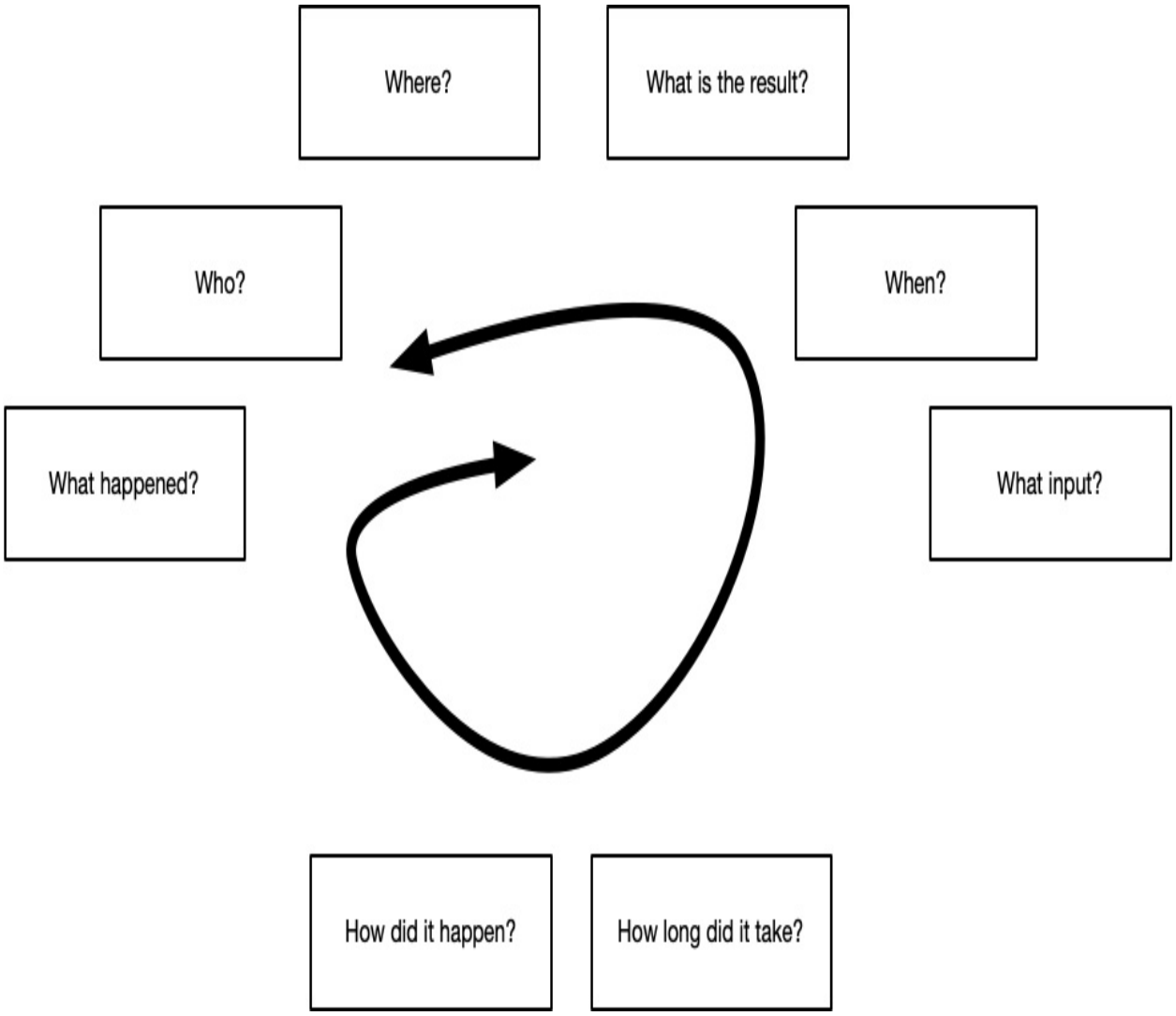
I saw this struggle firsthand while helping my kid with schoolwork. His teacher returned an essay, saying it was short and lacked details. He was frustrated—he didn't know what to add. So, I gave him a list of questions to help him figure out what information to include.

Six starts with W, and two with H. I called them 6+2. The W-questions form the foundation—essentials you should always ask. The H-questions add detail, but not every situation needs them.

My son could easily modify these questions as needed. I printed them out and stuck them on my desk. That's when I realized they work just as well for logging.

When I sketched them out in Figure 1.4, they reminded me of a giant mushroom. The six key questions stand tall, like a canopy protecting it from rain, while the two smaller ones support it from below.

Figure 1.4 The circle of questions: Start with one and circle until you ask them all. It will help you find the proper context.



Let’s take a closer look at these questions.

Table 1.2The 6+2 questions

Question	Answer
Who performed the action, and who was affected?	Name accounts, processes, IDs
What happened?	Factual description of the event

Where did it happen?	In the database, in a specific component, or in a third-party system
What input or data was provided?	User-provided input, additional information from another system
When did it happen?	Information on time and date
What was the actual result, and how did it differ from expectations?	You expected to get money, but you didn’t get any.
How did it happen?	Name variables and context; are you missing information?
How long did it take?	Was it extraordinarily long or short? How many seconds?

When something in your system puzzles you, you ask, "Why did this happen?"—your log file should have the answer. If not, return to the 6+2 questions—maybe you missed something.

But if you ever feel like drowning in useless information, revisit the 6+2 questions and ask if they matter.

Writing log statements is complicated. Ten programmers will give you eleven opinions on what makes a good one.

You’re not a senior developer until you say, "I should have logged this."

Applying logging to real life

Logging is invaluable in these real-world scenarios.

Legacy and Complex Systems

Logs are a lifesaver for aging or complex systems—like that ten-year-old Swing application, your company left you to fix. These systems are often hard to understand, debug, or follow—especially if they use outdated programming models. Detailed logs reveal system behavior and save hours of guesswork.

Real-Time Systems

Logging is indispensable for real-time or safety-critical systems—like medical devices. Take a defibrillator: you need to know if it worked in the field, but you can't debug it live. Logs verify its operation without interrupting critical tasks.

Multi-User Systems

For web apps with thousands of users, logs help trace actions—especially when things go wrong. They reveal what happened, letting you recreate the issue and find the cause.

Frequent Deployments

In fast-moving Agile or XP teams, frequent deployments mean constant production changes. Even with tests, surprises happen. Logs give you a near real-time view of changes, filling in the gaps tests miss and catching unexpected behaviors.

1.4 How to log

Applications don't log data automatically. No programming language does this out of the box. It would need to decide what's important to log and what isn't.

So, who decides what gets logged? You do—the developer!

1.4.1 Real-world logging

What is the simplest way to log something? Use the `System.out.println()` method. Unfortunately, this only prints to the Console, not to a file or anywhere else. You'd also have to add timestamps and other details manually. A timestamp is the current date and time and provides information about when something happened. We don't just need to know what happened—we need to know when. That's why timestamps matter. This becomes tedious and messy, as you can see below.

```
LocalDateTime now = LocalDateTime.now(); #1
DateTimeFormatter formatter = DateTimeFormatter.
ofPattern("yyyy-MM-dd HH:mm:ss"); #2
String dateTime = now.format(formatter); #3
System.out.println(dateTime + " - Customer " +
customerName + " entered flight "
+ flightNumber); #4
```

Now, imagine repeating these lines throughout your application. They would be a nightmare to maintain, and you'd write even more code just to manage them. What if someone else handled this for you? What if you could use their code in your application? Using someone else's code makes your project dependent on theirs. That's why we call third-party code a dependency. Today, using dependencies is standard practice. Dependencies can be quickly added to your project using Maven (Appendix A), and from there, you can use them in your code. What if a developer created a dependency that handled logging for you? If the developer provided clever code for timestamps, we might write one line instead of many. All our "logging code" could be simplified.

Many people have already built logging solutions. They take the burden off you. You don't have to build or maintain them yourself. Developers call these dependencies *logging frameworks*. They provide a structured way to handle logging.

1.4.2 Logging frameworks

Manually handling log statements gets messy fast. Writing logs, formatting timestamps, and managing output locations might initially seem simple—

until your application grows.

What if you need logs written to a file, a database, and an external monitoring service simultaneously? What if you need different levels of logging—detailed logs for debugging but minimal logs in production?

This is where logging frameworks come in.

A logging framework is a software component that provides methods to help you write log statements to your preferred destination. It makes it easy to control where and how logs are stored—without cluttering your application code.

One of the most common methods of a logging framework that you'll see is `log.info()`. Other common methods include `debug()` and `error()`, which we will discuss later. It prints a message like `System.out.println()`. It doesn't just print to the console; it can log to a file or another destination. Here's the simplest way to use it: just a log message.

```
log.info("Customer Christian entered flight 1234");
```

Every time you call this method, the framework writes a log statement to the log storage. This example always logs the same message because it is hardcoded and contains no variables. A log statement that never changes isn't very helpful. To make logs meaningful, we need to include variables. That's why logging statements often include variables, like this example using string concatenation:

```
log.info("Customer " + customerName + " entered flight " + flight
```

This approach is often discouraged because it creates unnecessary objects, which can slow down an application. To solve this, modern frameworks use placeholders instead. In frameworks like Log4j, placeholders look like `{}`. You can pass variables to the log method to replace them in the log statement.

```
log.info("Customer {} entered flight {}", customerId, flightId);
```

This change makes the code more readable. However, this still isn't ideal in a

real-world application like our flight system. In practice, this kind of logging happens multiple times within a method. For example, a method might log an event, perform some business logic, and then log again, as shown below.

```
public void enteredFlight(String customerId,
 String flightId) { #1
    log.info( #2
        "Customer " + customerId + " entered flight " + flightId)

    Integer reservationId = checkReservationId(
      customerId, flightId); #3

    log.info(
      "Customer " + customerId + " checked in for flight " + flig
      " with reservation ID " + reservationId);

    // Do something else
}
```

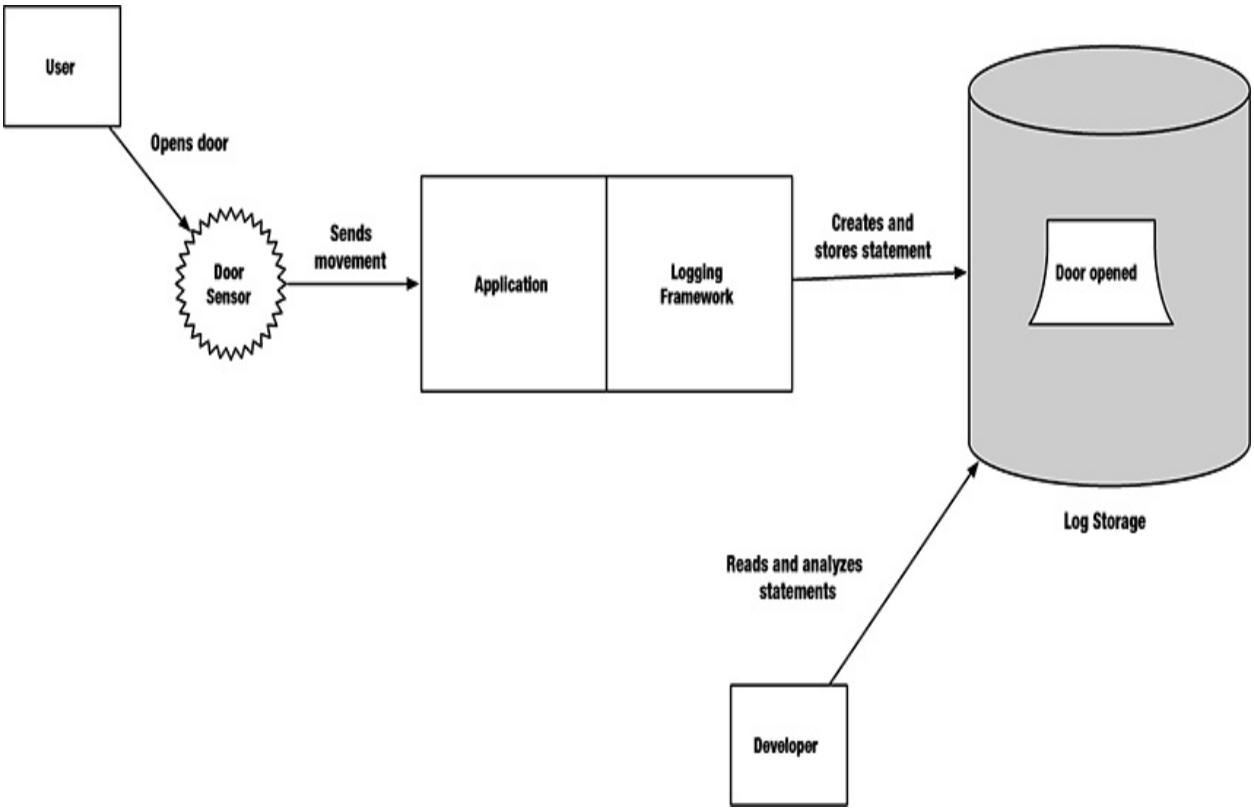
When this code runs, the log output may look like this—an example from a logging framework. The `info()` method automatically adds timestamps, so we don't need to code them manually.

```
2024-03-15 12:15:19,198 INFO FlightApplication - Customer Christi
booked flight X1234.
```

The logging framework automatically adds a timestamp. This log entry is slightly different from our first example. It includes more information, but it's a more technical version of what we've already seen. A key advantage is that other systems can easily parse it.

At some point, either a person or a machine must read and interpret these log files. They need to understand the logged data and take action if needed. If a person reads the logs, they'll likely use advanced log viewers to search and analyze them more efficiently. You can see this in action in Figure 1.5.

Figure 1.5 When a user generates an "open door" event, the system records it in the log file. Developers can read it for debugging purposes.

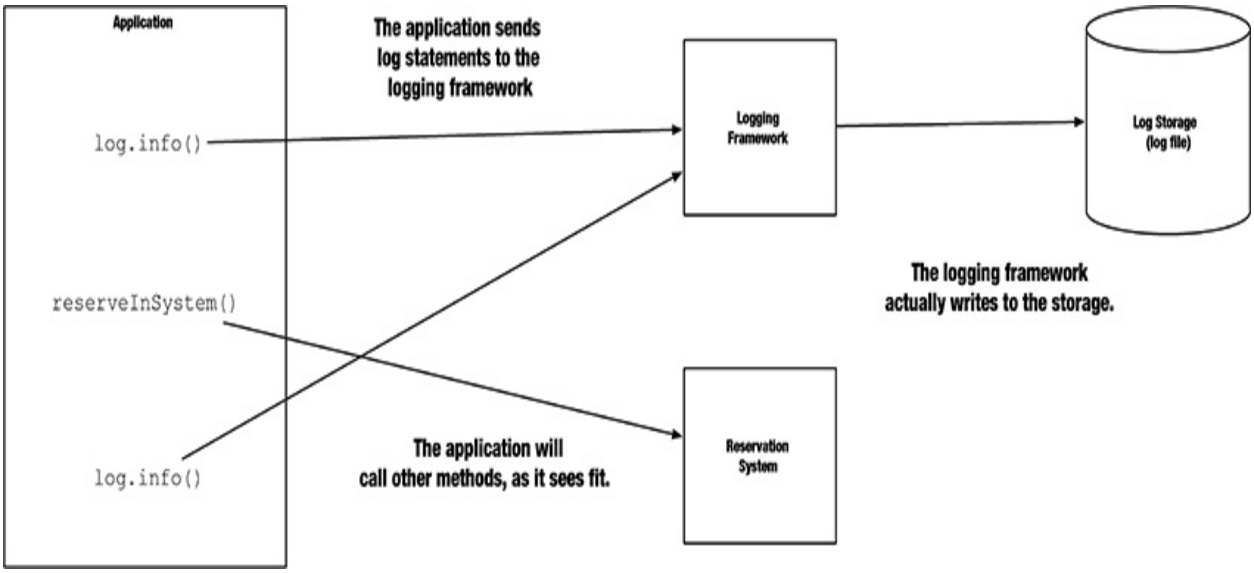


1.5 High-level architecture

As a software developer, you decide when your system should log information. You also determine what to log. Log statements are created only at runtime—when the application is running.

Figure 1.6 visualizes this concept: the flight app calls not only the`checkReservationId()` method but also the `info()` methods from the logging framework.

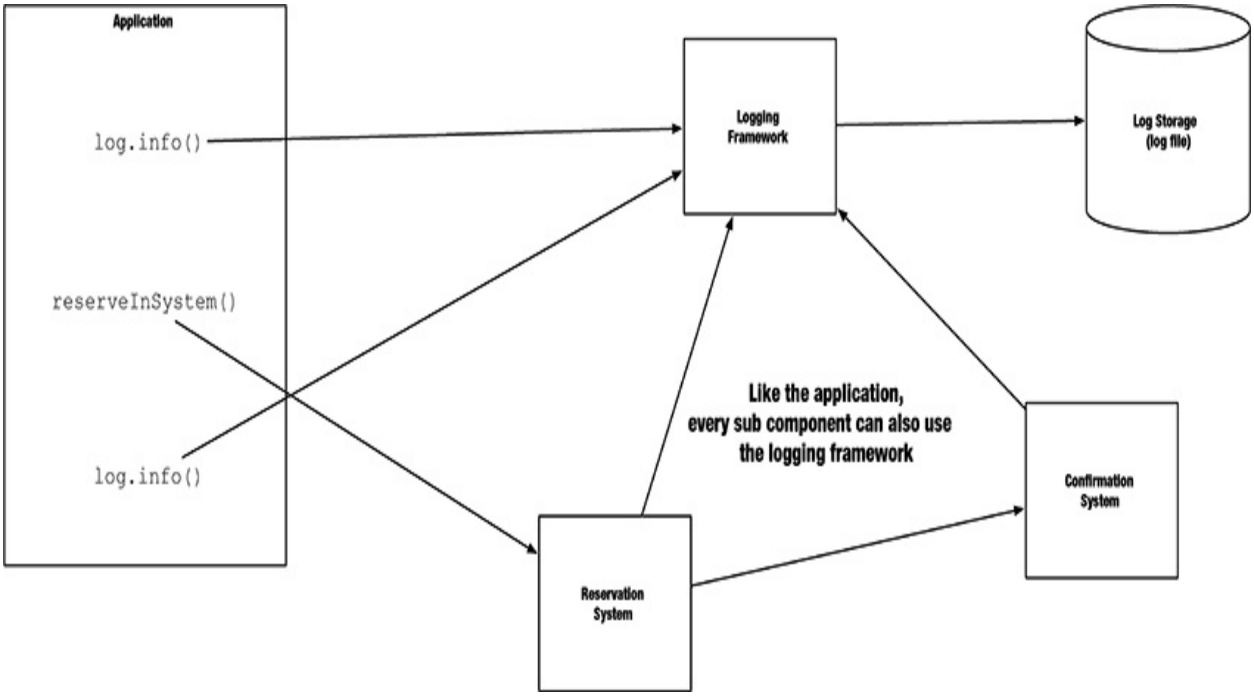
Figure 1.6 The flight app is responsible for booking flights. It will also use sub-components or microservices as the reservation system to perform its actions. Also, it wants to write logs directly.



Any class can call the logging framework—including the reservation system.

Ultimately, every component and subcomponent in your system can write log statements. This can result in many log statements across different parts of your system, as you can see in Figure 1.7.

Figure 1.7 Like the flight app, the reservation system wants to write logs. The logging framework will provide the same service to this component.



The logging framework is the central hub for processing log data from all components. With so much data, the key question is: what makes a log meaningful? Meaningful logs provide precise details that help you understand what’s happening, identify issues, and keep the system running smoothly.

1.6 What product do I need to log?

All you need is your usual Java setup and a logging framework.

Choosing a framework is often a matter of preference and trust. Most offer similar core features, such as turning on or off logging and simplifying log management.

Some well-known frameworks include:

- Log4j 2.x
- Java Util Logging (JUL)
- Logback (with SLF4J)

Since I’m involved with Log4j, this book primarily focuses on it. Log4j is robust, widely used, and maintained by a large team, making it a reliable choice.

However, we’ll also explore JUL and Logback. Both are widely used and well-regarded. JUL comes bundled with Java, so no additional dependencies are needed. The same developer behind Log4j 1 created Logback, which has proven its reliability over time.

Much of what you learn with Log4j also applies to JUL and Logback. In the following chapters, we’ll explore logging frameworks' unique challenges and features, followed by a closer look at the most popular ones.

1.7 Summary

- Logging records a history of events during an application’s execution.
- Logging is essential for capturing meaningful context through log statements.

- Log statements are messages recorded in log files or storage.
- Log files are essential for debugging, monitoring, and auditing.
- Logging frameworks offer methods like `log.info()` for writing log statements.
- Popular logging frameworks include Log4j 2, Java Util Logging, and Logback.

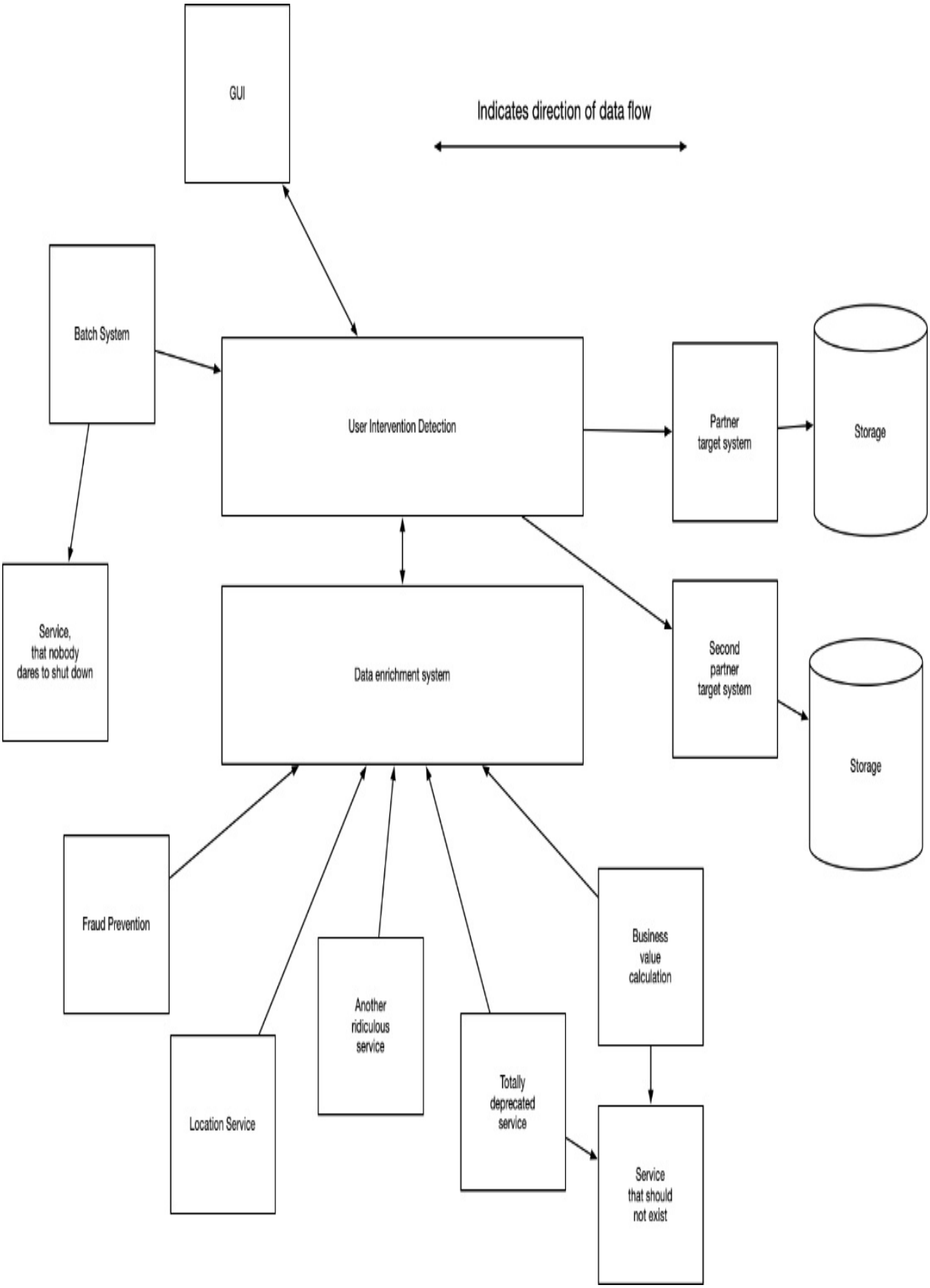
2 The use case for log4print

This chapter covers

- Essential terms and concepts in logging
- Building log4print, a simple logging framework
- Understanding the challenges that logging frameworks solve

Twenty years ago, I worked on a massive project. The system was overly complex–more than 30 subsystems ran independently. Today, we'd call this a microservice architecture, except these were more macroservices. The challenge was the same: tracing how data moved through the system. I sketched the architecture in Figure 2.1 to give you a sense of the scale.

Figure 2.1 A real-world example of a complicated architecture with multiple subsystems. In some cases, developers connected hundreds of so-called microservices, making it hard to understand how data flows.


GUI
Indicates direction of data flow
Batch System
User Intervention Detection
Partner target system
Storage
Service, that nobody dares to shut down
Data enrichment system
Second partner target system
Storage
Fraud Prevention
Business value calculation
Another ridiculous service
Location Service
Totally deprecated service
Service that should not exist

I remember staring at a sketch like this and wondering how anything worked. My job was to help identify and fix problems at runtime. Unfortunately, developers had turned off logging. The company policy demanded it to save costs. Suddenly, every system was on alert. Something terrible had happened. Our investigation led nowhere until we convinced the company to turn logging back on. Minutes later, we found the problem.

The message read:

```
---
I could not reach the deprecated service after 30 seconds of wait
Here is a NullPointerException because we did not expect this.
---
```

Logging showed us what was happening while the system was running. We would have prevented a lot of gray hair (and saved a lot of money) if logging had been turned on earlier.

In this chapter, we walk through a logging framework’s core concepts and terms. By the end, you’ll understand how logging works–and what makes a good logging statement. To make this concrete, we’ll build a simple logging framework called "log4print."

Let’s imagine we are building software for an aircraft. Every aircraft has a flight recorder, a device that records everything happening during the flight. It captures data like outside temperature, flight direction, or when the pilot pushes a button. Flight recorders are built to survive a crash. Whatever went wrong it’s recorded there.

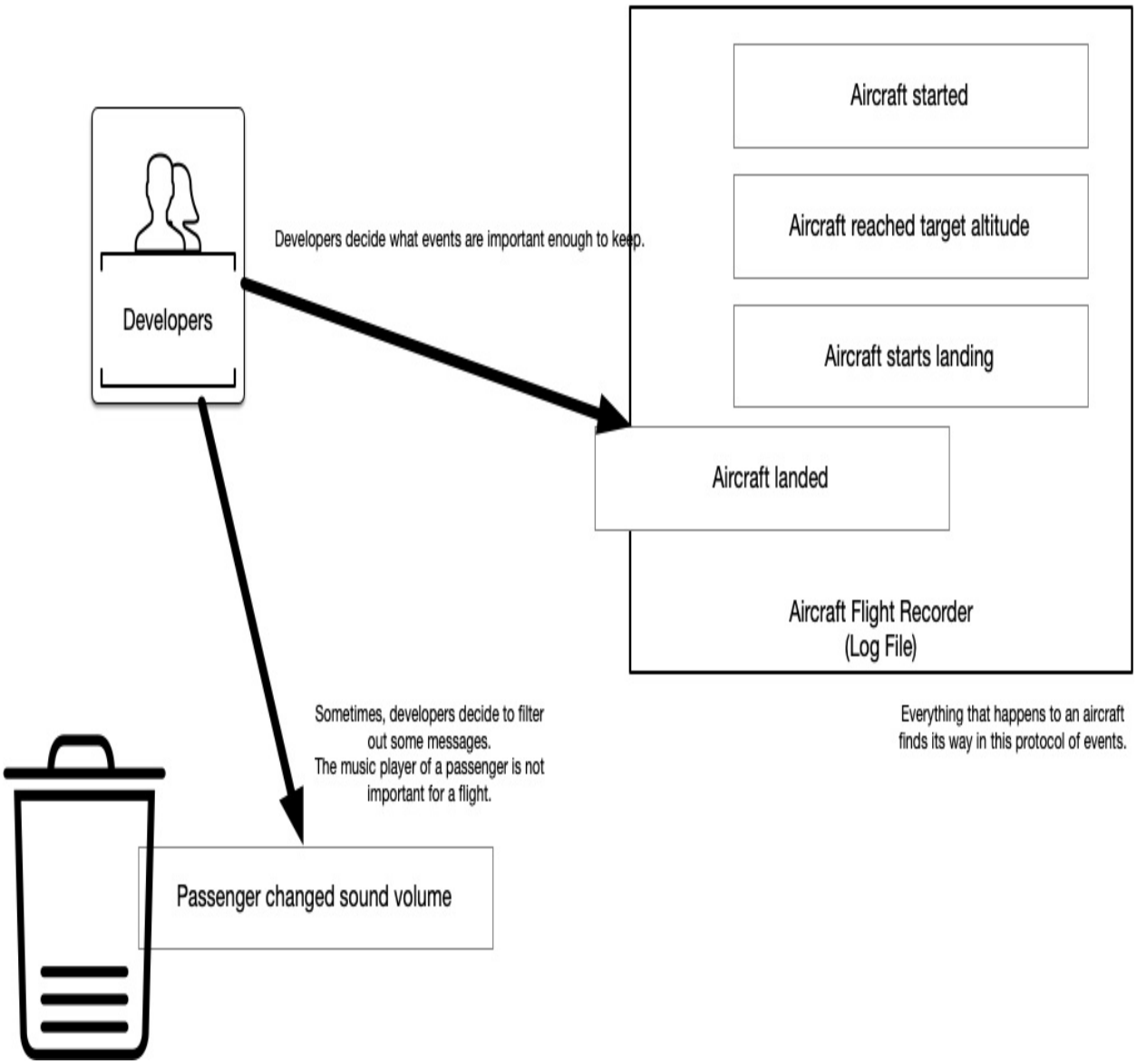
A logging framework is like a flight recorder for your application. It records important events while the system is running. If something breaks, it’s the first place you look.

In this chapter, we’ll build our version of such a system. We’ll call it "log4print".

2.1 Log files and log messages

Before we dive into the details, we should define what exactly we want to record in the flight recorder. The most interesting events are starting, taking off, and landing. A typical logging statement is a line of information that tells us, for example, that the pilots started the aircraft. Developers refer to the textual representation of such an event as *log statement*. Sometimes, it's also called *log event* or *log message*. Developers often use the terms interchangeably, but they all describe the same thing: a record of a specific event in a system. Figure 2.2 shows a log file with multiple log events.

Figure 2.2 A log file can store many log events that developers create.



Now that we know what a log statement is let’s think about what we can do about it. Traditionally, developers stored these statements in a *log file*: a simple file where all log statements are appended, one after another.

While log files are still widely used, modern systems increasingly rely on specialized databases or cloud-based storage services for more advanced features. Therefore, I prefer the term *log storage* over *log file*, even though it’s less common.

2.2 Log4print: let’s start!

We know what we want to log in and need some configuration support. That’s enough to get started. Here’s what we want to achieve:

- Add extra details to each log message, like timestamps
- Turn logging on and off as needed
- Assign different criticality levels to messages
- Configure the logging framework easily
- Send log messages to different destinations

Time to start coding.

2.2.1 Our first message

Let’s start by imagining how we want to write something to the console. That’s usually the first thing you learn in any Java book: `System.out.println()`. If we want to log that our system started and ended, it might look like this:

```
System.out.println("System started");
// some other code
System.out.println("System ended");
```

Time matters. The "When" question is part of our 6+2 questions and adds crucial context. It’s fundamental because it adds context to everything else. I still remember a nasty bug that only appeared late in the evening. Nobody could figure out why.

In the end, the timestamp gave us the clue. My senior developer went to the office at exactly the right time, determined to catch the bug. He was staring at the screen when the door opened. The cleaning lady unplugged the computer to plug in her vacuum. Without the timestamp, he might have waited there all night.

Let's add a *timestamp* to this message. A timestamp is nothing special; it's just the the moment when the message was created. Usually, it comes in a format like `2023-04-20T22:15:19.264991`.

Luckily, Java's built-in Date & Time classes make this easy. One of the most useful is `LocalDateTime`, which represents the current date and time. It's *local*, because it doesn't deal with the complexity of time zones. For our simple program, that's perfect.

You can create a `LocalDateTime` object by calling the static `now()` method. Since the object overrides `toString()`, you can use it directly in string concatenation or pass it to `System.out.println()`.

```
LocalDateTime dateTime = LocalDateTime.now(); #1
System.out.println(dateTime + ": System started"); #2
```

This prints a readable date and time. The output is simple and effective:

```
2023-04-20T22:15:19.264991 System started
```

The `LocalDateTime` format might feel unusual. It includes fractions of a second and uses a date order you might not know. If that bothers you, you can always reformat it to your preference. Here's what our full example looks like now. As you can see, the repeated timestamp code is already starting to bloat our main logic:

```
LocalDateTime startDateTime = LocalDateTime.now(); #1
System.out.println(startDateTime + ": System started");

// some other code

LocalDateTime endDateTime = LocalDateTime.now(); #2
System.out.println(endDateTime + ": System ended");
```

While this code works, it’s already clear we should move the logging logic into a separate method. One that handles time creation logic and string concatenation. Before we get there, let’s add another feature: the ability to turn logging on or off. As you’ll see, that makes the code bloat even harder to ignore.

2.2.2 Turning logging on and off

Every time we log something, we create a new `LocalDateTime` object. That not only bloats our code but also eats up memory. When memory is limited or expensive, someone may ask us to turn off logging. A simple switch could help here. We could solve this by adding a boolean variable at the top of our code. If `loggingEnabled` is `true`, we log. Otherwise, we skip it.

```
boolean loggingEnabled = true; #1
if (loggingEnabled) {
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now + ": System started");
}
```

This solution is all or nothing—and it’s fair to be skeptical. As the program grows, we might encounter several variables controlling logging in different parts of the code. Maintenance quickly becomes a nightmare. It leads to inconsistent behavior and makes debugging a frustrating experience. And then there is… readability. All these `if/else` blocks clutter the logic and are hard to ignore.

```
boolean loggingEnabled = true;

if (loggingEnabled) {
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now + ": System started");
}

new App().process(); #1

if (loggingEnabled) {
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now + ": System ended");
}
```

Developers tend to overlook important code when it’s buried in noise. In this case, there is only one interesting line. Unfortunately, it is hard to spot with all the logging code around it.

Code like this breaks every rule in the clean-code book. It is repetitive, distracting, and hard to maintain. Even worse, it adds to our *technical debt*, a measure of the growing cost of cleaning up our messy code later. Over the years, this debt only grows as others add duplicate lines or patch new functionality. There is one proven strategy against this kind of bloat: create classes and methods to encapsulate the logic and reuse it.

2.2.3 Using classes for logging

Writing flexible code is challenging, but once you start using methods, classes and objects, you are already on the right track. Our code above could be much simpler if we had a method that handled logging for us Let’s assume we add such a method somewhere in our code:

```
private static boolean loggingEnabled = true;

public static void log(String message) { #1
    if (loggingEnabled) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now + " " + message); #2
    }
}
```

Instead of repeating the same lines, we simply call the static `log()` method whenever we need to write a log message. That makes the code much cleaner:

```
log("System started"); #1
new App().process(); #2
log("System ended");
```

The next step is to move this method into its class. The heart of object-oriented programming (OOP) is breaking code into methods and classes. Don’t hesitate to make full use of Java’s features. They improve readability and make your code easier to maintain. Aim for one method to do one task as a general rule of thumb. Let’s take that idea further and create a dedicated

class for logging.

2.2.4 Extracting to the Logger class

Just like methods, classes should also do one thing. This idea is known as the *Single Responsibility Principle*, a core principle of object-oriented programming (OOP). I won't dive deeper here; plenty of books and articles on the topic exist. For log4print, it means creating a dedicated class that handles only logging. This also means, we can remove the static modifiers, since we create an object of this following class.

Listing 2.1 Wrapping our code into a class for better reusability.

```
public class Logger { #1
    private boolean loggingEnabled = true;

    public void log(String message) { #2
        if (loggingEnabled) {
            LocalDateTime now = LocalDateTime.now();
            System.out.println(now + " " + message);
        }
    }
}
```

With that, we have two separate classes: `App` and `Logger`. The `App` class handles the application logic, while the `Logger` class handles logging. From here on, we'll use the `Logger` class to handle our log messages. The only real change is that we need to create a `Logger` object first.

```
Logger logger = new Logger(); #1
logger.log("System started"); #2
new App().process(); #3
logger.log("System ended");
```

Like before, we want to control logging with a simple flag—this time inside the `Logger` class. We'll call it `loggingEnabled`.

The idea is simple: if the flag is `true`, we log. If it's `false`, we skip the message.

Listing 2.2 Using a simple flag to enable logging.

```
public class Logger {
    public boolean loggingEnabled = true; #1

    public void log(String message) {
        if (loggingEnabled) { #2
            LocalDateTime now = LocalDateTime.now();
            System.out.println(now + " " + message);
        }
    }
}
```

This approach works if you only need to control logging for each `Logger` instance. For simple cases, that might be enough. But a single flag like this won't be enough in most real-world systems. Instead of scattering these switches everywhere, it's better to introduce a configuration object that multiple `Logger` instances can share.

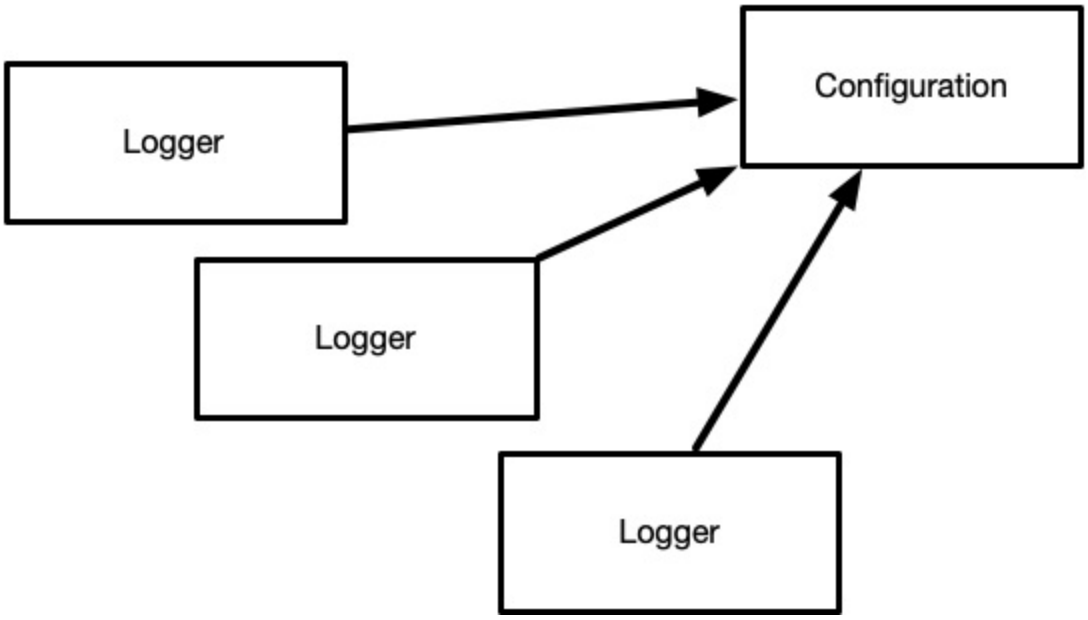
2.2.5 Introducing logging configuration

Every logging framework offers a way to configure its behavior. Usually, you configure this via XML, JSON, or plain Java. With log4print, we'll keep it simple and stick to plain Java to focus on the concepts rather than the tooling.

The first step will be to extract the configuration from the `Logger` class. Following the *Single Responsibility Principle*, `Logger` should focus only on logging. This separation makes the code easier to maintain, reduces duplication, and ensures consistent behavior across the system.

For now, the only configuration we care about is the `loggingEnabled` flag. Moving it out allows multiple `Logger` instances to share the same configuration. That's a significant improvement: instead of changing settings in many places, we'll have a single source of truth. Figure 2.3 shows the new architecture.

Figure 2.3 Extracting the configuration



Think of a configuration as a simple object holding the setting. This first version only controls whether logging is on or off.

Listing 2.3 A first configuration class with a single setting: turning logging on or off.

```
public class LoggingConfiguration {
    private boolean turnOn = true; #1

    public void setTurnOn(boolean turnOn) {
        this.turnOn = turnOn;
    }

    public boolean isTurnOn() {
        return this.turnOn;
    }
}
```

Before we can use the configuration, we need to inject it into the `Logger`. *Injection* means we don't create the configuration in the `Logger`. Instead, we pass it in. This way, multiple `Logger` instances can reuse the same configuration. To do that, we add a field to hold the configuration and set it through the constructor.

Listing 2.4 Injecting the configuration into the Logger

```
public class Logger {
```

```
    public LoggingConfiguration configuration; #1

    public Logger(LoggingConfiguration configuration) { #2
        this.configuration = configuration;
    }

    public void log(String message) {
        if (configuration.isTurnOn()) { #3
            LocalDateTime now = LocalDateTime.now();
            System.out.println(now + " " + message);
        }
    }
}
```

It's still simple to use. First, we create a configuration object, then a `Logger`, passing the configuration in. Now, the benefit of reusing the configuration becomes visible. Both `Logger` instances share the same configuration. In the example below, logging is turned off with a single setting.

Listing 2.5 Sharing the configuration

```
LoggingConfiguration configuration = new LoggingConfiguration();
configuration.setTurnOn(false); #2

Logger logger1 = new Logger(configuration); #3
logger1.log("Good morning");

Logger logger2 = new Logger(configuration); #4
logger2.log("Good night");
```

The example looks very good already, except that we still inject the configuration manually into every `Logger` object. This is error-prone. To make this safer, we need a better way to create our `Logger` instances. The next step is simple: we create a class responsible for building `Logger` objects for us.

2.2.6 The LoggerManager

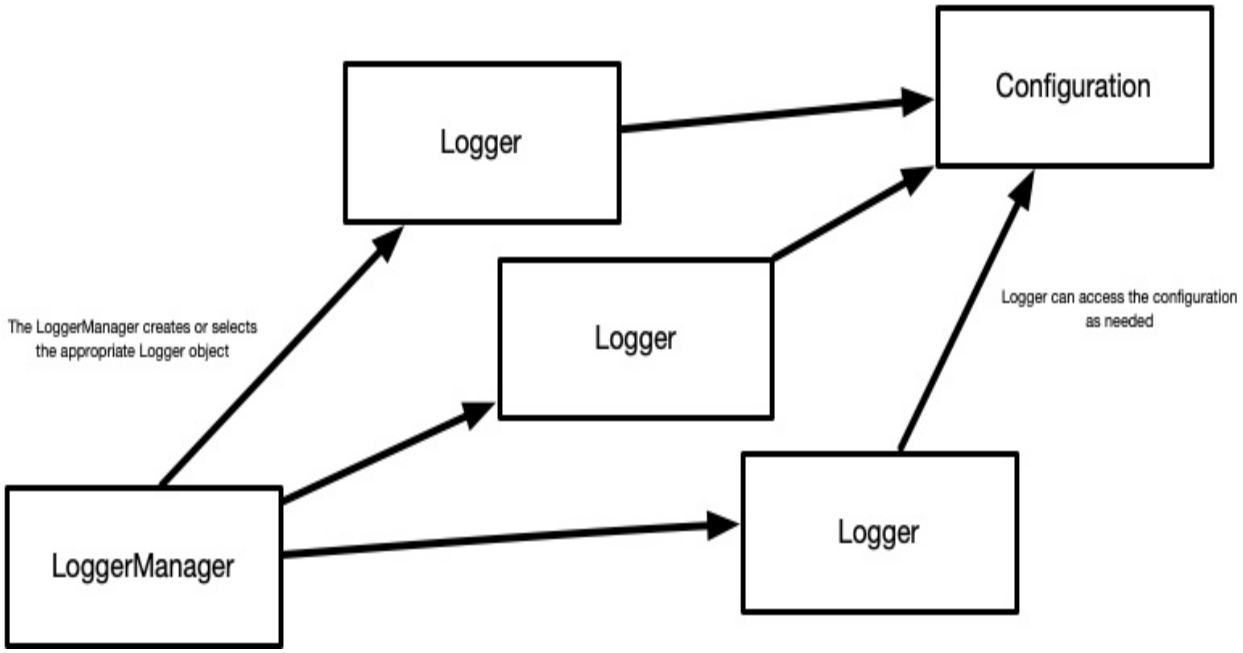
Whenever we need a `Logger` object, we need to receive the configuration object and pass it to the constructor, which adds unnecessary hassle. It would be better to have an object or class that:

- creates the configuration for us,
- creates the logger by adding the configuration,
- and returns us the ready-to-use object.

We'd avoid typos, duplicated setup, or worse—multiple configurations with conflicting settings. Such a concept deserves a name; let's go with `LoggerManager`, a term I borrowed from the Log4j framework.

The `LoggerManager` is a central object that creates `Logger` objects. It ensures consistent configuration and reduces general duplication across the application. Figure 2.4 visualizes, how the new component fits in.

Figure 2.4 The Logger Manager



The `LoggerManager` is a so-called factory class. Its only purpose is to create `Logger` objects for us. The simplest implementation would be a static method that returns a new `Logger` object. Factories are everywhere in Java code; you'll see them used constantly.

Listing 2.6 A simple factory

```
public class LoggerManager {
    public static Logger getLogger() { #1
```

```
        return new Logger(); #2
    }
}
```

We are missing the configuration object in this example. The next step is to create a new configuration each time we create a `Logger`. After creating the configuration object, we need to set the preferred settings.

Listing 2.7 Configuring using a LoggerManager

```
public class LoggerManager {
    public static Logger getLogger() {
        LoggingConfiguration configuration = new LoggingConfigura
        configuration.setTurnOn(true); #1

        return new Logger(configuration);
    }
}
```

Of course, we can do better. We create a new configuration each time, wasting memory and effort. It is not shared between the various loggers and a reconfiguration would require us to change all existing configuration objects. By making the configuration object static, we can share it between all loggers. This usage ensures that only one instance exists, making it effectively a singleton. Some developers consider Singletons an anti-pattern. In this case, they allow us to share the configuration object between all loggers. We must check if the variable is null and creates the configuration only if needed. If it is, we create a new configuration object.

Listing 2.8 Creating the configuration

```
public class LoggerManager {
    private static LoggingConfiguration configuration; #1

    public static Logger getLogger() {
        if (configuration == null) { #2
            configuration = new LoggingConfiguration();
            configuration.setTurnOn(true);
        }
        return new Logger(configuration);
    }
}
```

Note

We need to make sure multithreaded programs also work. If two threads call `getLogger()` at the same time, they might create two configuration objects. Luckily (for me), this isn't a book about Java multithreading. We'll stick to a single-threaded example. Software like Log4j solves these problems—and should be preferred over homegrown solutions. Otherwise, we would have to deal with the keywords `synchronized` or `volatile`.

We can improve our `LoggerManager` even further. We could extract the configuration creation using a separate method. This method needs to be static, too. It will reduce the necessary code in the `getLogger()` method and follow the "one method should do one thing" rule.

Listing 2.9 Creating the configuration

```
public class LoggerManager {
    private static LoggingConfiguration configuration;

    private static void configure() { #1
        configuration = new LoggingConfiguration();
        configuration.setTurnOn(true);
    }

    public static Logger getLogger() {
        if (configuration == null) {
            configure(); #2
        }
        return new Logger(configuration);
    }
}
```

The `configure` method sets the correct values to our configuration object. It could read settings from XML files, other configuration files, or maybe even the database. "log4print" only supports plain Java, which makes the `configure` method simple. Instead of handling the configuration ourselves, we use `LoggerManager` to create our `Logger`. Our source code becomes much easier to read and maintain.

Listing 2.10 Using two different loggers

```
Logger logger1 = LoggerManager.getLogger();
logger1.log("Good morning");

Logger logger2 = LoggerManager.getLogger();
logger2.log("Good night");
```

An "Aircraft" class might use the `LoggerManager` to receive a `Logger` instance. Usually, you'd use a static variable to keep the reference to loggers.

Listing 2.11 A typical class using LoggerManager

```
public class Aircraft {
    private static Logger logger = LoggerManager.getLogger(); #1

    public void takeOff() {
        logger.log("Taking off"); #2
    }
}
```

The next feature of our logging framework is handling message importance. Not every log entry is equally critical, so we need a way to classify them. That's the job of log levels. Now that we have a centralized way to manage loggers let's deal with the next challenge.

2.2.7 Log-Levels

How severe is a log statement? It should come with a tag or label that indicates its importance. A message about a `NullPointerException` feels very different to `User logged in successfully.`

A *log level* is a way to categorize log messages and helps you find what matters. You can filter out messages you are not interested in. You could turn on logging for only your errors or general information. If you are looking for a specific issue, you could turn on detailed logging to help debug.

Out of all the log levels available, I use two every day. They are called `DEBUG` and `ERROR`.

Logging frameworks introduced `DEBUG` primarily for developers. These statements help you understand the system and track down issues. They are

rich in information, and you'll see them a lot. A passenger turning on their TV or adding headphones might cause a `DEBUG` statement. It may show the event, the headphones' volume, the TV channel's sender, and the seat number.

`ERROR` is a log statement that tells us something went wrong. Not only developers but also everybody who maintains our system should read them. Sensors unable to read outside humidity or intercoms failing to connect with the rest of the cabin crew might trigger `ERROR` statements.

As discussed earlier, `OFF` is helpful for completely silent logging. No statements are processed, saving system resources.

To implement it, we need to look at Java enumerations. Enumerations are a powerful Java feature for defining a fixed set of constants. For `log4print`, we will implement the mentioned `DEBUG, ERROR,` and `OFF` log levels.

Listing 2.12 The LogLevels are expressed as an enumeration

```
public enum LogLevel {
 DEBUG,
 ERROR,
 OFF;
}
```

When we think about the behavior of these levels, we might come up with this: A developer might set the log level in the configuration. If they set the log level to `DEBUG,` we still want to see `ERROR` statements. If they set it to `ERROR,` we might only care about the problems and skip the rest. Last but not least, `OFF` means off. With a hierarchy, we can bring this into a logical order. If I wrote it down on a sheet of paper, it might look like this:

```
OFF > ERROR > DEBUG
```

I should use user numbers since we have to code it in Java. If OFF is three, ERROR is two, and DEBUG is one, I can easily compare them. This number is the "weight" of a log level, meaning that `OFF` has much more weight and importance than `DEBUG`. Enumerations can have constructors and variables, which helps me to assign the weight to each log level.

Listing 2.13 The LogLevel enumeration assigns a weight (importance) to each log level.

```
public enum LogLevel {
    DEBUG(1),
    ERROR(2),
    OFF(3); #2

    private final int weight; #1

    LogLevel(int weight) {
        this.weight = weight;
    }

    public int getWeight() {
        return weight;
    }
}
```

When I first heard about log levels, I found them confusing. I tried multiple metaphors to remind me how they worked. My favorite one was the story of Atlantis City, in which things went so wrong that logging would not even help.

2.2.8 The Atlantis Angle

In real life, there are far more log levels than just `DEBUG` and `ERROR`. The more log levels you have, the harder it becomes to understand.

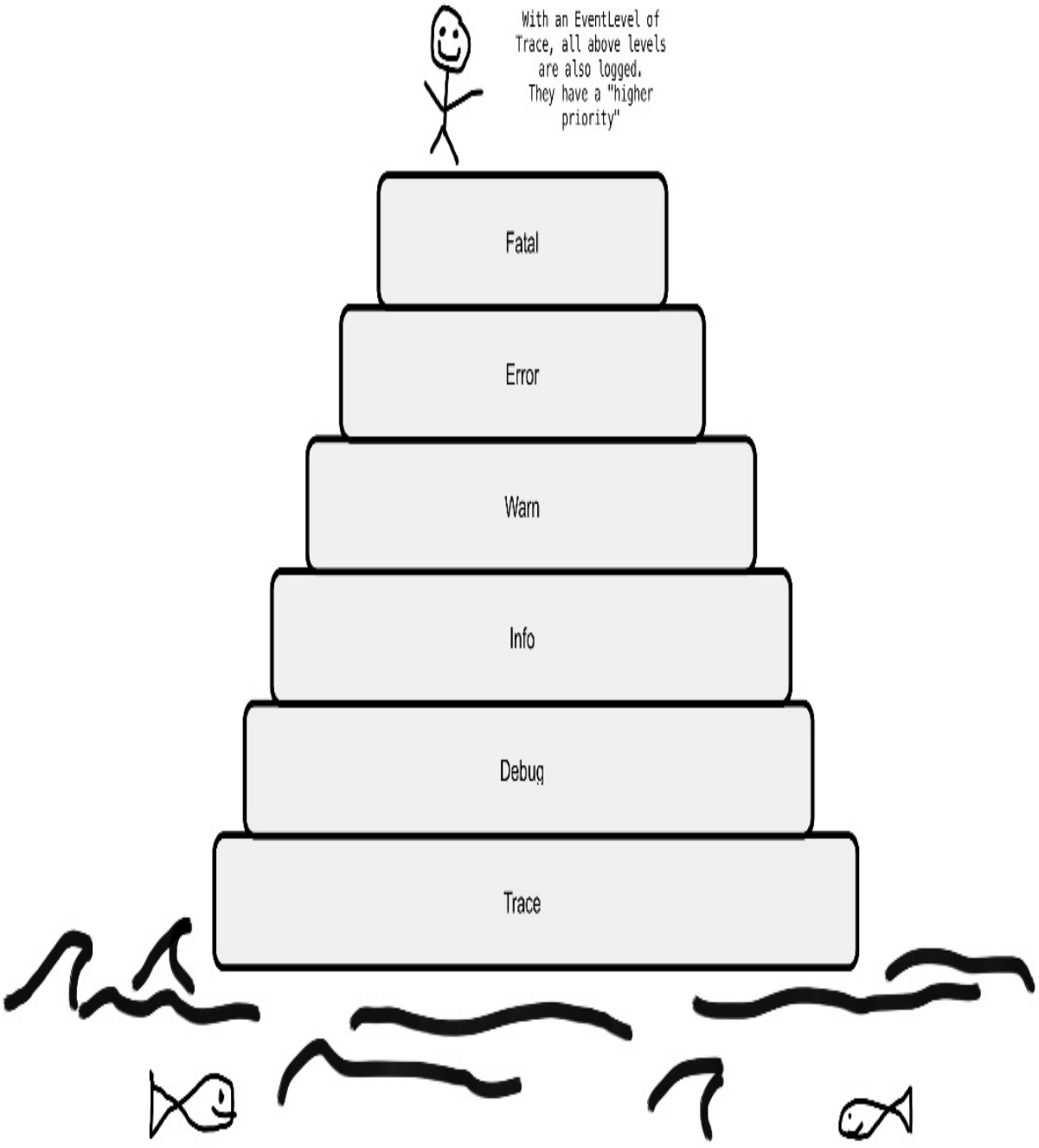
What helped me many years ago was a scene from a movie about Atlantis. Vikings rammed the island with their boat, causing it to sink. The Vikings warned the inhabitants, but they started a debate instead of acting. They used a Roman forum, a podium with many steps. Step by step, the water rose, flooding the lower levels and covering more and more participants, silencing more and more of the participants. Eventually, the person on the highest step disappeared, and the debate was over.

That's how I picture log levels: `ERROR` is the last to vanish. Chatty levels like `DEBUG` disappear first.

In Figure 2.5 , I added all potential log levels you might see. If we log a message with a level we haven't seen yet: `TRACE`, all other levels are visible

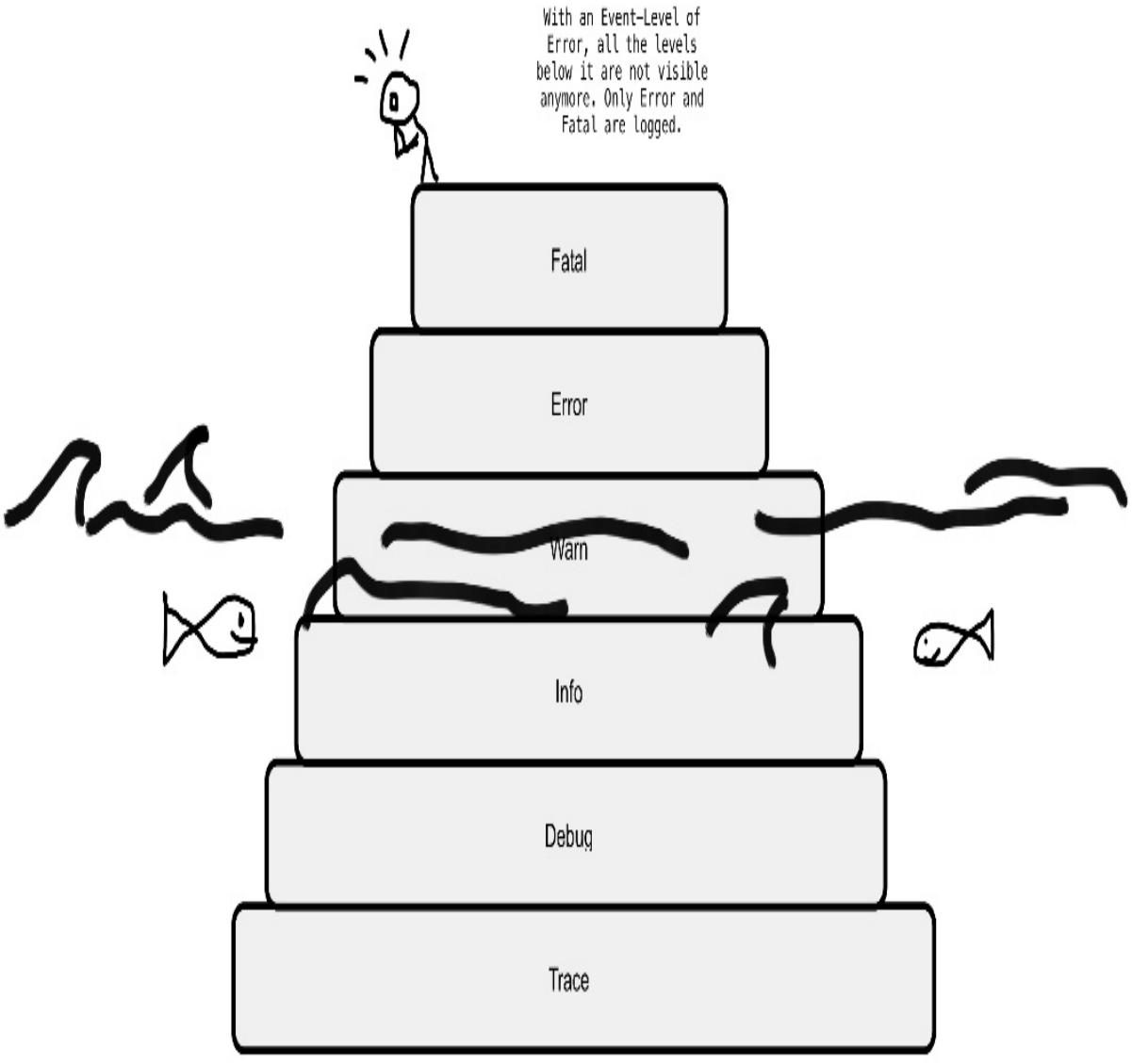
too. The logger will process everything that’s still above the waterline.

Figure 2.5 The logger level is set to `TRACE`. All messages on `TRACE` and above are visible, and the logging framework processes them.


With an EventLevel of Trace, all above levels are also logged. They have a "higher priority"
Fatal
Error
Warn
Info
Debug
Trace

As you can see in Figure 2.6 , the water rises when developers configure the logger to the `ERROR` level. Fewer steps are visible, and fewer log statements are processed. Only `ERROR` and `FATAL` statements make it through—everything else is ignored.

Figure 2.6 The logger level is set to `ERROR`. Only `ERROR` and above are visible. Everything else is below the surface, and the logging framework completely ignores all messages on that level.


With an Event-Level of Error, all the levels below it are not visible anymore. Only Error and Fatal are logged.
Fatal
Error
Warn
Info
Debug
Trace

Hopefully, this Atlantis angle helps you remember how log levels work. Next, we'll look at configuration—where you'll also set the log level.

2.2.9 Configuring log-levels

Next, we allow developers to configure the minimum log level they want to see. For us, it's just another configuration setting. This time, we use the enumeration.

Listing 2.14 Extending the configuration to support LogLevel

```
public class LoggingConfiguration {
    private LogLevel minLogLevel;

    public void setMinimumLogLevel(LogLevel minLogLevel) { #1
        this.minLogLevel = minLogLevel;
    }

    // ...
}
```

Next is, to set the minimum log level. this needs to be done in the `LoggerManager`, so we have to extend it a bit.

Listing 2.15 Extending the LoggerManager to set a default

```
public class LoggerManager {
    private static LoggingConfiguration configuration;

    private static void configure() {
        configuration = new LoggingConfiguration();
        configuration.setMinimumLogLevel(LogLevel.DEBUG); #1
        configuration.setTurnOn(true);
    }
}
```

Once we can set the minimum log level, we need a way to decide whether to process a log statement. The configuration class handles this with a `shouldLog()` method.

It compares the minimum log level from the configuration to the log level of

the statement.

Listing 2.16 The `shouldLog` method

```
public class LoggingConfiguration {
    // Minimum log level omitted for brevity

    public boolean shouldLog(LogLevel logLevel) {
        if (minLogLevel == LogLevel.OFF) { #1
            return false;
        }

        if (logLevel.getWeight() >= minLogLevel.getWeight()) { #2
            return true;
        }

        return false; #3
    }
}
```

The configuration compares the minimum level with the given log level. If the given one is higher or equal, we log. The only exception is when the minimum log level is set to `OFF`. In that case, the method returns immediately —there's no point checking further.

Next, we return to the `Logger` class and update the `log()` method. It now needs to accept the log level of each statement.

```
logger.log(LogLevel.DEBUG, "Taking off");
```

Inside the `log()` method, we call the `shouldLog()` method of the configuration. It compares the statement's log level with the minimum level set in the configuration.

Listing 2.17 Logger with a log method

```
public class Logger {
    public LoggingConfiguration configuration;

    public Logger(LoggingConfiguration configuration) {
        this.configuration = configuration;
    }
```

```
    public void log(LogLevel logLevel, String message) { #1
        if (configuration.shouldLog(logLevel)) { #2
            LocalDateTime now = LocalDateTime.now();
            System.out.println(now + " " + message);
        }
    }
}
```

Quite an improvement: Developers can now use our framework and specify a log level. The framework decides if the statement is important enough to be logged. Our `Aircraft` class stays clean and easy to read.

Listing 2.18 Aircraft with a log level

```
public class Aircraft {
    private static Logger logger = LoggerManager.getLogger();

    public void takeOff() {
        logger.log(LogLevel.DEBUG, "Taking off"); #1
    }
}
```

While this code works well and isn’t too annoying, we can even make it more beautiful by adding a "convenience method." Cases like this—designed only to make other code easier to read—are often called "syntactic sugar."

The next example shows exactly that: a method wrapping the log level and the message. Adding it to the `Logger` class even further improves the readability of the `Aircraft` class.

Listing 2.19 Some syntactic sugar

```
public class Logger {
    // Omitted for brevity

    public void debug(String message) { #1
        log(LogLevel.DEBUG, message); #2
    }

    public void error(String message) { #3
        log(LogLevel.ERROR, message);
    }
}
```

By adding these methods, we no longer need to use the enumeration directly. We just call the convenience methods. Usually, the IDE helps us with that. The following code shows how we'd use them:

```
Logger logger = LoggerManager.getLogger();
logger.debug("All systems look fine"); #1
logger.error("Oh no!"); #2
```

We're already close to having the most critical feature of a logging framework. But there's one thing left: we need to decide where our log statements should go. Right now, everything ends up on the console. We may want to store logs in a file, too. With log levels, we might even route specific messages to different destinations—separate files, databases, or monitoring dashboards.

2.2.10 Destinations: Logging appenders

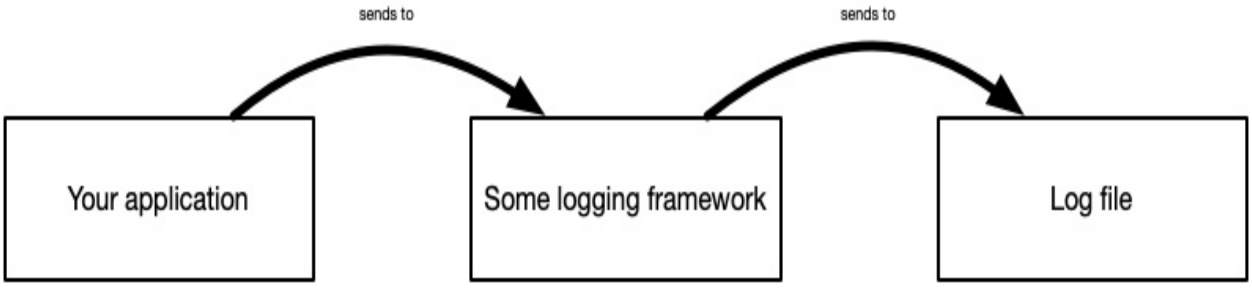
When writing log statements, we must decide where to store them. If we don't handle it, the system prints the log to the screen, and it's gone. That would mean a developer sitting in front of the console all day, waiting for potential issues.

That doesn't sound like an appealing job. It's better to store those statements somewhere and review them when needed.

Very quickly, developers turned to log files to store these statements.

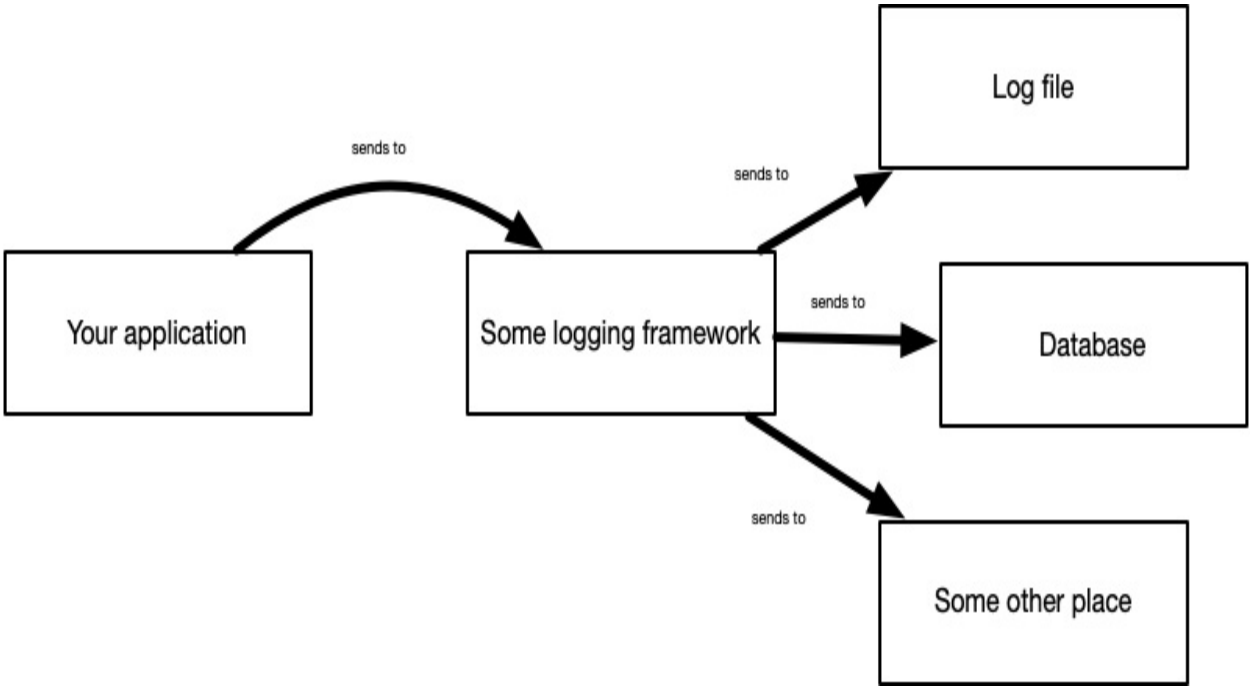
Dealing with files can be tricky, so it's better to let log4print handle that part. Developers create log statements in their application code. Figure 2.7 show, how the logging framework stores them in a log file.

Figure 2.7 An application sends log statements to a framework, which decides how to handle them and where to store them.



Sometimes, developers don’t want log statements in just one file. Maybe they want them in two–who knows? Or they want the logs on the console without missing anything from the file. Modern logging frameworks make configuration easy. The flexibility, shown in Figure 2.8 , means we can manage multiple destinations without changing the application code.

Figure 2.8 If you prefer, you can add multiple destinations for log statements. Your application does not need to handle this; it is all configured in the logging framework.



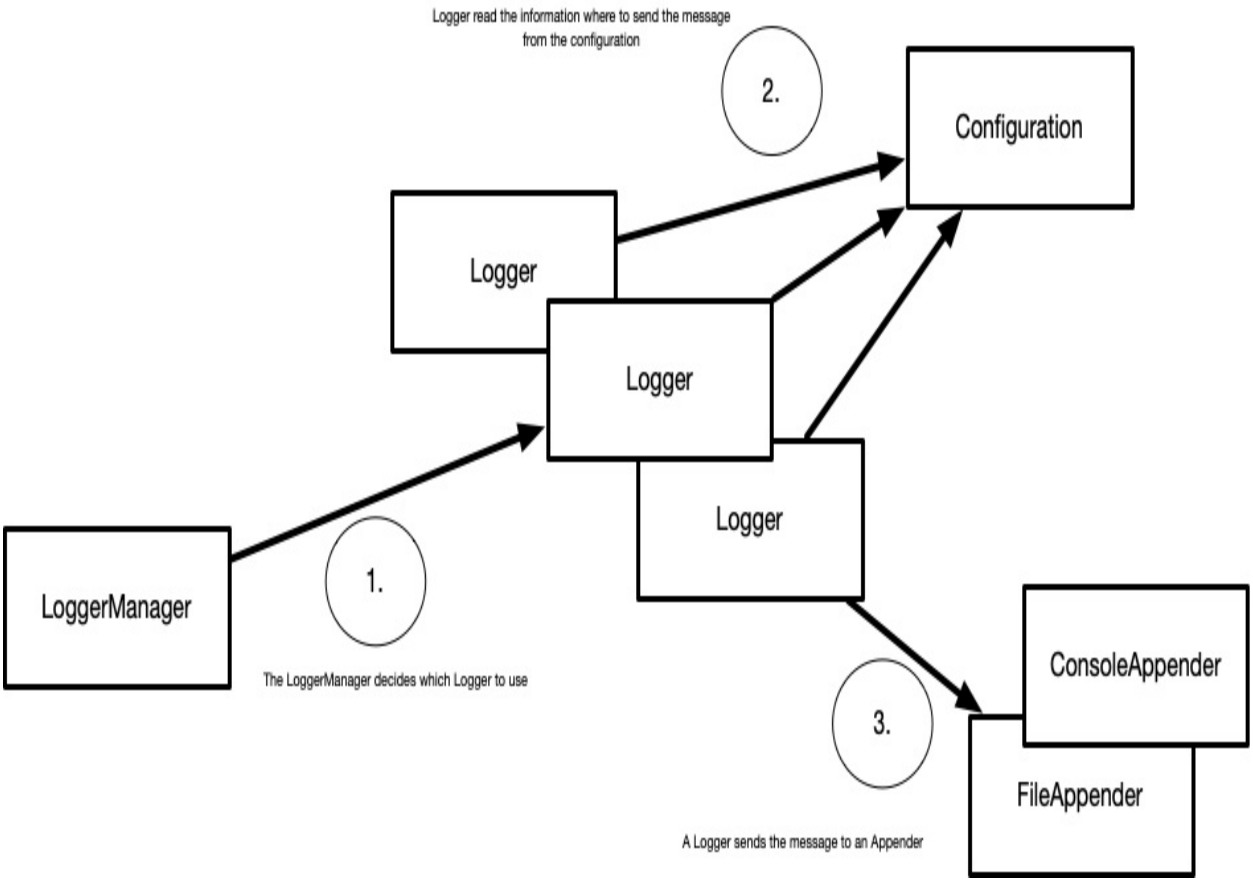
Generally, logging frameworks offer a variety of destinations for your log statements. If you’re experimenting, start simple with console or file output. Files are easy to work with. You can open them in Notepad, Visual Studio Code, or any text editor. If your log files grow too large for your editor, specialized tools can help, as seen in Chapter 5, section How to read log files.

Currently, we append one statement after the other to the console, and that's what we want to change. If we extract the functionality that writes the log statement to the console and move it to a separate class, we could switch between those classes and replace the console with a file. This thought is the idea behind the concept of an *appender*. Appenders collect and process logging statements, but this does not mean they are exclusive. You can use multiple appenders to make the same log statement. This means that security personnel can have their log file with authentication issues, while developers can have their log files with debugging information. Our architecture changes a bit when we introduce appenders.

Currently, we append one statement after another to the console—but that's what we want to change. We can swap the console for a file or other destinations if we extract the part that writes log statements and move it to a separate class.

This is where the concept of an *appender* comes in. Appenders collect and process log statements—and they're not exclusive. You can use multiple appenders at once. For example, security might log authentication issues while developers get detailed debugging information. Adding appenders changes our architecture slightly, as you can see in Figure 2.9 .

Figure 2.9 Adding appenders to the architecture



Next, let's think about the structure of an `Appender` class. Since we might want to add different types of Appenders—like `FileAppender` or `ConsoleAppender`—it makes sense to define a common interface they all implement. This interface requires a method that takes the log level and the message. Each Appender can then decide how to handle the message.

```
public interface Appender {
    void writeLog(LogLevel level, String message); #1
}
```

If we write the `ConsoleAppender`, we must implement the `Appender` interface. But in reality, it only needs a simple `System.out.println()`.

Listing 2.20 The Console Appender

```
public class ConsoleAppender implements Appender { #1
    public void writeLog(LogLevel level, String message) {
        System.out.println(message); #2
    }
```

```
}
```

With that code, it becomes clear: the Appender's job is simple: take a message and move it to its final destination. Writing a `FileAppender` is a bit more complex than `ConsoleAppender,` but not by much.

The `FileAppender` needs a file path usually passed from the configuration. We use it to create a `PrintWriter,` which wraps a `FileWriter`. `PrintWriter` can write formatted text and extends `FileWriter's features. Once we have the writer, we use it to write the message to the file instead of the console.

Listing 2.21 The File Appender

```
public class FileAppender implements Appender {

    private PrintWriter writer;

    public FileAppender(
        String logFilePath) throws IOException { #1
        writer = new PrintWriter(new FileWriter(
        logFilePath, true)); #2
    }

    public void writeLog(LogLevel level, String message) {
        writer.println(message); #3
        writer.flush(); #4
    }

    public void close() { #5
        writer.close();
    }
}
```

In logging, we must ensure that log messages are written to the file immediately. That is why it is wise to call `flush()` after writing the message.

About the flush method

You might be wondering about the `flush()` method. It's crucial because it forces anything written via `println()` to be sent immediately to the destination. This is especially important for file operations, where output is

often buffered. Buffering means data isn't written immediately—it waits until the system decides to flush the buffer. For logging, that delay is risky. Logs are often critical for diagnosing issues, so it's common practice to call `flush()` to ensure log messages are written instantly.

An appender becomes even more helpful if we automatically print the log level next to the message. Adding formatting improves readability—and `String.format()` makes this trivial. The `ConsoleAppender` changes slightly, and the `FileAppender` would look similar.

```
public void writeLog(LogLevel level, String message) {
    String logMessage = String.format("[%s]: %s", level, message)
    System.out.println(logMessage); #2
}
```

After we have created the Appender classes, we need to make them available to the Logger objects. A good place to maintain the various appender objects could be the configuration or any other object serving as a registry. I have chosen the configuration since it will keep this example simple. When we create the configuration, we can also create the appenders. In production-ready logging frameworks, appenders might only get created when they are used to save resources.

Listing 2.22 Initialization of the Appender objects

```
public class LoggingConfiguration {
    // Other code
    private ConsoleAppender consoleAppender;
    private FileAppender fileAppender;

    public LoggingConfiguration(LogLevel minLogLevel) {
        this.minLogLevel = minLogLevel;
        consoleAppender = new ConsoleAppender(); #1
        try { #2
            fileAppender = new FileAppender("log.txt"); #3
        } catch (IOException e) {
            throw new RuntimeException(e); #4
        }
    }

    // more code
}
```

We should add a helper method to return the right appender for a given name. That way, the configuration can return the proper appender object for a simple string that represents the appender. This lets the configuration return the right appender based on a simple string.

Listing 2.23 Method to receive an appender

```
public class LoggingConfiguration {
    private ConsoleAppender consoleAppender; #1
    private FileAppender fileAppender;

    public Appender receiveAppender(String appenderName) { #2
        if ("file".equals(appenderName)) { #3
            return fileAppender;
        }

        return consoleAppender; #4
    }

    // more code
}
```

Now we have everything in place. Let’s create three different logger objects, each with a unique name: "app," "audit," and "aircraft."

To connect each logger to the right appender, we store the mapping in a `Map` inside the `LoggingConfiguration`. The map uses the logger name as the `String` key and the `Appender` as the value.

```
public class LoggingConfiguration {
    private Map<String, Appender> appenderMapping = new HashMap<>
    // Other code
}
```

Finally, we need a method to configure the logger with the appender. Another method will map the logger name to the appender name. The new `map()` method lets you populate the `appenderMapping`. We can reuse the `receiveAppender()` method to find the right appender for a given name.

Listing 2.24 Configure the appender

```
public class LoggingConfiguration {
```

```
    private ConsoleAppender consoleAppender;
    private FileAppender fileAppender;

    public void map(String loggerName, String appenderName) { #1
        appenderMapping.put(loggerName,
        receiveAppender(appenderName)); #2
    }

    public Appender receiveAppender(
        String appenderName) { #3
        // ...
    }

    // more code
}
```

So far, we've talked a lot about how a `Logger` object gets a name, but we still need to implement it. To do that, we update the constructor so it accepts the configuration and the logger name.

Listing 2.25 Adding the name property

```
public class Logger {

    private String name; #1
    private LoggingConfiguration loggingConfiguration;

    public Logger(String name,
    LoggingConfiguration configuration) { #2
        this.name = name;
        this.configuration = configuration;
    }

    // ...
}
```

We can now create loggers with a name. The `LoggerManager` needs a small adjustment to support that.

```
public class LoggerManager {
    private static LoggingConfiguration configuration;

    public static Logger getLogger(String name) { #1
        if (configuration == null) {
            configure();
```

```
        }
        return new Logger(name, configuration); #2
    }
}
```

Of course, we could now start caching these loggers in a map to avoid creating new `Logger` objects every time. For our example, let’s skip this step — it’s optional.

Now, it’s time to bring all the pieces together. Earlier, we used the `LoggerManager` class to handle configuration. It feels natural to also configure the `Logger` to `Appender` mapping here.

Listing 2.26 The new way to configure our framework

```
public class LoggerManager {
    // Code omitted for brevity
    private static LoggingConfiguration configuration;

    public static void configure() {
        configuration = new LoggingConfiguration(LogLevel.DEBUG);
        configuration.map("app", "console"); #1
        configuration.map("audit", "file"); #2
        configuration.map("aircraft", "console"); #3
    }
}
```

Every `Logger` named "app" or "aircraft" will log directly to the console. Only the "audit" logger writes to a file. There’s plenty of room for improvement, but at this point, we have a working system.

Next, we need a small adjustment in the `Logger` and `LoggingConfiguration` classes. The `Logger` should use its name to ask the configuration for the correct appender whenever it writes a log statement.

To make this work, we need a method in the `LoggingConfiguration` class that returns the correct appender for a given logger name. Here’s the helper method that does exactly that:

```
public class LoggingConfiguration {
    // Other code
```

```
    public Appender receiveAppenderByLoggerName(
        String loggerName) { #1
        return appenderMapping.get(loggerName);
    }

    // more code
}
```

Finally, the `Logger` class must fetch the correct appender from the configuration. Thanks to the `Appender` interface, the `Logger` doesn’t care whether it’s writing to a file, the console, or anything else.

Listing 2.27 Logging with the appender retrieved from the configuration

```
public class Logger {

    private String name;
    private LoggingConfiguration configuration;

    // Constructor

    public void log(LogLevel logLevel, String message) {
        Appender appender =
            configuration.receiveAppenderByLoggerName(
                name); #1

        if (configuration.shouldLog(logLevel)) {
            LocalDateTime now = LocalDateTime.now();
            appender.writeLog(logLevel, now + " " +
            name + ": " + message); #2
        }
    }
}
```

2.2.11 Using the new system

With that, we can receive the correct configuration by using a logger name.

For the `Aircraft` system, we would need to use the term "aircraft" to receive the correct appender.

```
public class Aircraft {
    private static Logger logger = LoggerManager.
    getLogger("aircraft"); #1
```

```
    private static Logger auditLogger = LoggerManager.
    getLogger("audit"); #2

    public void takeOff() {
        logger.log(LogLevel.DEBUG, "Taking off");
        auditLogger.debug("Taking off - prove for the auditor.");
    }
}
```

With that, we've completed our custom logging system. Along the way, we introduced essential concepts like `LoggerManager`, `Configuration`, `Appender`, and `Logger`. If you care about good software design, you've probably already spotted several areas that could be improved. Reflecting on scalability and maintainability is always worth your time.

While "log4print" works, it's just the tip of the iceberg. Instead of inventing yet another logging framework, let's talk about why you should stop here—and what existing logging frameworks already solve for you.

2.3 Why not write a custom logging system?

Whenever you write code, you also write bugs. This rule of nature applies even more to low-level frameworks like logging, where performance and thread safety are critical. In this chapter, we've done things that are not best practice—and sometimes even dangerous. We might know and handle some issues, but the problems we don't know about should worry us.

Let's look at the limitations and pitfalls of "log4print"—and why relying on such a system in production would be risky.

2.3.1 Reconfiguration

If we need to reconfigure our system, we must change the code, recompile it, deploy it to production, and restart it. That's inconvenient—especially if we want to inspect something quickly. After finishing, we might need to roll back all those changes and restart the system.

A better approach would be to read the configuration from external files, like XML. Changing the file should trigger the system to pick up the new

configuration—ideally without a restart and without losing any log statements.

Reading a file is simple. Handling reconfiguration without dropping messages, on the other hand, is a hard problem.

Reconfiguration should never cause downtime. And losing log statements? Auditors hate that.

2.3.2 Dealing with files

We implemented a file appender, but it could be better. You may have noticed that we didn't close the resource properly, which could have caused trouble at the operating system level.

And that's just the start. If you've ever tried to open a 10 GB file with Notepad, you know what I mean. If you're not careful, log files can grow and even fill up your entire file system.

Proper log file management means thinking about things like log rotation or archiving. But with that, your system complexity grows. Ignore it, and you risk running out of disk space.

2.3.3 Multithreading issues

Multithreading in Java allows multiple threads to run in parallel within a single program. Think of threads as little programs running simultaneously, sharing resources like memory and files. It's standard in Java—and many frameworks use it heavily, even if you don't see it.

Our current code isn't ready for that. On startup, for example, we might accidentally create multiple configurations when we expected just one. Worse, two threads could try to write to the same file simultaneously.

That's where things get messy. Making code like ours bulletproof in multithreaded environments are tough—and not trivial.

2.3.4 Problems with System.out.println

`System.out.println()` can cause problems. We've used it often without thinking twice—but what's the issue?

If you look into the Java 17 source code, you'll find something interesting. The `println()` method contains a synchronized block. And if you check the `writeln()` method, you'll see the same.

Listing 2.28 The Java 17 `println()` method

```
public void println(String x) {
    if (getClass() == PrintStream.class) {
        writeln(String.valueOf(x));
    } else {
        synchronized (this) { #1
            print(x);
            newLine();
        }
    }
}
```

`System.out.println()` works fine for small-scale applications but quickly breaks down when generating high volumes of logs. You might have seen the `synchronized` keyword before. It's used to make code safer in multithreaded environments.

That might sound good at first—but in the case of logging, it's a problem. `synchronized` means only one thread can execute the synchronized block at a time. Other threads must wait until the first one finishes. For logging, this creates a bottleneck—exactly what you don't want when your system is under load.

When writing many logs by using `println()`, we could experience a significant slowdown since the various threads would have a long wait time.

Blocking access might come up not only with `println()` but also with file access. We need to identify that kind of issue and either document its behavior or fix our appenders in a way these issues don't happen.

2.3.5 More features

We've written a lot of code covering the basics, but plenty's still missing. Right now, the `LoggerManager` doesn't handle cases where someone asks for a `Logger` that doesn't exist. We also can't set log levels per `Logger`; it's system-wide only. And formatting log messages differently? That's not exactly easy, either.

2.3.6 It's not that easy

Writing your simple logging system looks easy at first—but it isn't. Sooner or later, feature requests pile up, and you'll spend more time writing boilerplate than solving real problems.

Homemade frameworks spread through your codebase. Years later, replacing them means rewriting large parts of your system—risking new bugs with every change.

Maintaining a logging framework might be fun if you enjoy learning about multithreading, file access, and the gritty details of systems programming. But chances are, your boss won't be thrilled watching you build infrastructure instead of features that generate revenue.

Most of us need logging, but it rarely feels exciting compared to machine learning, fraud detection, or other complex challenges. Maintaining your logging system is hard, lonely work. Funding and long-term support? Unlikely.

Luckily, there are mature, well-maintained logging frameworks—some refined by experts for over 20 years. Instead of reinventing the wheel, why not use one? If you're curious, you can join an open-source project and contribute. I explain how that works in the Appendix.

Good logging frameworks free you to focus on what matters. They offer simple APIs but handle the tricky parts behind the scenes—so you don't have to.

2.4 Summary

- Log messages record what the system is doing and are stored in log storage, such as log files.
- They should include variables and describe what happens in the system.
- Logging frameworks use factories to create `Logger` objects—we called ours `LoggerManager`.
- Log levels like `DEBUG` or `ERROR` categorize messages by importance.
- Appenders handle where log messages go—such as files or the console.
- Configuration objects store the settings for the logging system.

3 Basic logging patterns with System.Logger

This chapter covers

- Using the `System.Logger` for simple use cases
- Working with log levels in `System.Logger`
- Assessing the "pros" and "cons" of `System.Logger`

Java 9 introduced a new logging framework called `System.Logger`. Since we have covered all the basics in the previous chapter with "log4print", `System.Logger` will feel familiar to you.

`System.Logger` is a lightweight logging framework in the *Java Developer Toolkit* (JDK). It's easy to start with, and you can extend it later by integrating it with larger frameworks like Log4j if needed. It is a good choice for small projects or libraries such as:

- A utility that transforms JSON into Java objects
- A tool that estimates the CO2 emissions based on flight distance
- A service that verifies if a database is online
- A file compressor that zips files into archives

We'll also explore log levels in more depth. By the end, you'll know when and how to use a logging framework—and how to apply your knowledge to any project that only needs basic logging needs.

3.1 Receiving a System.Logger instance

Similar to the "log4print" framework from the last chapter, `System.Logger` uses a `LoggerManager` concept. As a reminder, a `LoggerManager` is a factory that gives you access to `Logger` instances used to create log statements.

In "log4print", we wrote the following code to get a `Logger` instance:

```
Logger logger = LoggerManager.getLogger();
```

With `System.Logger`, we replace that line with a call to `System.getLogger()`. Unlike our previous approach, the method requires a `String` argument: The name of the logger. Naming your logger is helpful, especially when integrating with other logging frameworks. For example, you can later use this name to turn off logging for specific application parts. Since we're not using any integration, we can use a simple name like "Aircraft." We obtain our logger instance like this:

```
System.Logger logger = System.getLogger("Aircraft");
```

At first glance, the type `System.Logger` might look a bit unusual, but it's just a regular Java inner class. In this case, `Logger` is a static inner interface inside the `System` class. You can use it like any other class or interface. Oracle's Java Tutorial—although a bit dated—is still a helpful reference if you want to learn more: https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html. With our logger instance in place, we can start logging messages.

Imports in Java

We don't need to import the `System` class. It's part of the `java.lang` package, which Java imports by default. This makes `System.Logger` feel even more lightweight: No third-party dependency is required, and not even an import is needed.

3.2 Log Levels for the System.Logger

In the previous chapter, we introduced the concept of log levels— categories that help classify the importance of log messages— using `DEBUG` and `ERROR` as examples. `System.Logger` provides three built-in log levels out of the box:

- `INFO`
- `WARNING`
- `ERROR`

We're already familiar with `ERROR`, but you may have noticed that `DEBUG` is

missing. Meanwhile, `DEBUG` and other log levels are only accessible when `System.Logger` is integrated with another logging framework. To keep things simple, we'll stick with the default levels provided.

- `INFO`: This level is for general application state updates that are not critical but helpful to understand what the system is doing. For example, "Service started," and "Connected to database." Without `DEBUG, INFO` often becomes the place for slightly more detailed messages.
- `WARNING`: Used for events that require attention but are not critical. These might indicate potential issues, like "Retrying failed connection" or "Slow database response detected." It's more urgent than `INFO,` but not yet an `ERROR.`
- `ERROR`: Reserved for serious problems that impact functionality. This could be "Failed to load configuration file" or "Service unavailable."

Be careful not to overuse `INFO` or `ERROR`. If everything is an error, nothing feels urgent. If everything is marked as `INFO,` meaningful messages get buried in noise.

3.3 Writing a simple log statement

Let's get to practice and log a message with the `INFO` level. By default, `System.Logger` is configured to log messages at the `INFO` level and above. That means `WARNING` and `ERROR` messages will also appear — just like with the "Atlantis Angle" from our `log4print` framework.

`System.Logger` default configuration is set to `INFO.` Following the Atlantis Angle we introduced earlier, it will also log messages from higher levels — `WARNING` and `ERROR.` In other words, if you set your logger to `INFO,` anything more severe will still appear.

Like with "log4print", `System.Logger` offers a `log()` method. Its signature is similar: it takes two arguments—one for the log level and one for the message. The `System.Logger.Level` enumeration defines the log levels.

I wonder if the author of that enum had a soft spot for long, literary sentences like those of Virginia Woolf. Either way, we have to deal with it.

```
System.Logger logger = System.getLogger("Aircraft");
logger.log(System.Logger.Level.INFO,
"Ready for take off"); #1
```

In the next section, we'll build the "AirportApp," a simple example demonstrating how logging is used in real-world scenarios. The use cases we'll cover are universal and can be adapted to any logging framework, not just `System.Logger`.

3.4 The AirportApp Skeleton

Our app needs a `main` method to start. As in many Java applications, it quickly initializes an instance of the `AirportApp` class so we can switch into the object context. From there, we immediately call the `execute` method, which contains the business logic.

```
public class AirportApp {
    public static void main(String[] args) {
        new AirportApp().execute(null); #1
    }

    public void execute(String lang) { #2
       // Business related code comes here
    }
}
```

Our skeleton app already has a flaw: we call the `execute` method with a `null` argument. This can lead to problems if the developer doesn't check for `null` inputs. For example, calling a method on a `null` value will trigger a `NullPointerException`.

```
public class AirportApp {
    public void execute(String lang) {
       String uppercaseLang = lang.toUpperCase(); #1
    }

    public static void main(String[] args) {
        new AirportApp().execute(null); #2
    }
}
```

Let's handle this situation by preventing the error and using logging to

inform the developer that something went wrong.

3.5 Naming the logger in an easy way

As we've seen, each logger needs a name. We could invent one—like "AirportApp"—but hardcoding strings like that is generally discouraged. Why? Because strings can't be refactored by IDE tooling, they're prone to typos.

Instead of guessing, we can ask Java directly by accessing the `.class` property of `AirportApp`. This property is part of Java's powerful reflection API and returns a `Class` object representing the class itself. From there, we call `getName()`, which returns the fully qualified name of the class—package and all. Something like `com.example.logging.AirportApp`.

This gives us a safe, typo-proof logger name that works well with future configuration tools.

The Reflection API

Java's Reflection API is a powerful tool—though sometimes considered slow — that allows programs to inspect and manipulate themselves at runtime.

Imagine code that can read the definition of a class, create an instance, and invoke a method marked with a specific annotation. All of that, and more, is possible with reflection.

While it sounds advanced, you already use parts of it regularly: the `getClass()` method and even the `instanceof` operator are built on top of reflection concepts.

```
String className = AirportApp.class.getName(); #1
System.out.println(className); #2
```

The output of this code would be something like `de.grobmeier.manning.logging.systemlogger.AirportApp`. We can use that fully qualified class name directly when creating a `System.Logger` instance:

```
public class AirportApp {
    System.Logger logger = System.getLogger(
    AirportApp.class.getName()); #1
    // rest of the code
}
```

This approach avoids hardcoding string names for loggers. Later, when we get into advanced configuration, you'll see how logger names— especially fully qualified ones—can control logging behavior more precisely. For example, you can filter logs by class or even by package.

Using the class name ensures consistency, avoids typos, and plays well with IDE refactoring tools. That alone makes it a solid choice early in your logging setup.

3.6 Warning: this shouldn't happen

One of the most common comments you'll find in source code is: **"This should never happen."**

Spoiler: it will.

Back when I was touring with my rock band, I might have said, **"That should never happen,"** if someone had told me I'd one day write a book about logging.

To avoid ending up like me (or worse—writing logging tools), you should ensure that your input values are what you expect them to be.

This isn't just good security advice—it's general best practice. In tech, you can't trust anything unquestioningly—especially not user input.

Since we know `null` is possible, let's decide how to deal with it. One option is to assign a default value. You could do this silently, but it's usually better to inform the developer if you're modifying inputs. After all, changing an input might change the behavior of the entire application.

And when something like that happens, a warning log is the right call.

Note

I always log parameters that don’t meet expected values or trigger errors. But when those parameters come from users, you must be especially careful.

User input can be malicious. Logging without validation could open up serious security issues.

If you’re wondering how bad that can get, check out section Log4Shell in the Appendix. It’s a textbook case of how logging unchecked input can go wrong.

The following code shows how we check if the `lang` variable is `null`. If so, we set a default value ("en") and log a warning to inform the developer.

```
public class AirportApp {
    System.Logger logger = System.getLogger(AirportApp.class.getN

    public void execute(String lang) {
        if (lang == null) { #1
            lang = "en"; #2
            logger.log( #3
                System.Logger.Level.WARNING,
                "Language was null, setting to default: " + lang)
        }
    }
 ...
}
```

Now that we’ve addressed the potential `NullPointerException`, we can make our application easier to debug by logging when entering and exiting methods.

3.7 Flow tracing

When the first line of a method logs an entry and the last logs an exit, you’re doing what’s called *flow tracing*.

Flow tracing helps you see the sequence of method calls and understand how your application executes. It’s basic but helps you follow the stack without a

debugger.

As you can imagine, flow tracing generates a lot of log statements — especially if you apply it across all methods.

That's why many developers use the `TRACE` level, which is specifically designed for capturing *everything*, from method entry to exit and beyond.

Unfortunately, `System.Logger` doesn't offer `TRACE` in its default configuration. So, for now, we'll use the `INFO` level to demonstrate the concept.

Typical flow tracing looks like this: one log statement at the start of the method and another at the end.

```
public class AirportApp {
 ...

    public void execute(String lang) {
        logger.log(
            System.Logger.Level.INFO,
            "Started Airport app"); #1

        // Business logic goes here #3

        logger.log(
            System.Logger.Level.INFO,
            "Ended Airport app"); #2
    }
 ...
}
```

A word of caution: flow tracing was once more popular than it is today. These days, many developers see it as a waste of resources—it generates a lot of log output, which can clutter your logs without adding much value. Modern tools make flow tracing largely obsolete. Instead of manually logging method entry and exit, developers use debuggers during development or tracing tools in production.

In production, such tools often rely on bytecode instrumentation—yes, that means they dynamically modify your bytecode at runtime to trace execution

paths automatically.

3.8 Dealing with unexpected problems

Now that we've built the basic structure of our method, we can add some business logic. Let's say we have an `AirportService` class with a single static method that returns an instance of itself. In theory, this should always work—but in practice? Who knows.

At this point, you have two options: You could throw an exception—something we'll cover in the next chapter. Or you could handle the problem immediately, right here, using the method. In the latter case, the best option might be to end the method early and log an error. You're changing the expected flow, which signals that something went wrong.

Listing 3.1 An example of business logic

```
AirportService instance = AirportService.getInstance(); #1
if (instance == null) { #2
    logger.log(
        System.Logger.Level.ERROR,
        "Did not receive airport service instance");
    return;
}

instance.performLogic(); #3
```

The combined code of our `AirportApp` already shows how much bloat and noise logging can add. This is a valid concern—and one that many developers raise. Unfortunately, there's no perfect solution. Either you live with the added verbosity or strip out logging and lose visibility into what's happening in your application.

Striking the right balance takes common sense—and a bit of experience. Logging isn't trivial. It affects the readability and maintainability of your code.

The complete example is shown below.

Listing 3.2 The full example

```
public class AirportApp {
    System.Logger logger = System.getLogger(
    AirportApp.class.getName()); #1

    public void execute(String lang) {
        logger.log(
            System.Logger.Level.INFO,
            "Started Airport app"); #2

        if (lang == null) { #3
            lang = "en";
            logger.log(
                System.Logger.Level.WARNING,
                "Language was null, setting to default (en)");
        }

        AirportService instance = AirportService
        .getInstance(); #4
        if (instance == null) { #5
            logger.log(
                System.Logger.Level.ERROR,
                "Did not receive airport service instance");
            return;
        }

        instance.performLogic();

        logger.log(
            System.Logger.Level.INFO,
            "Ended Airport app"); #2
    }

    public static void main(String[] args) {
        new AirportApp().execute(null);
    }
}
```

As you can see, logging takes up a lot of space—and creates many small `String` objects in the process. That's one reason developers often avoid it, especially in performance-critical code. You either accept the noise, or you fly blind. Logging is essential, but it needs a CPU and hard disk. Nothing's free.

3.9 Avoiding unnecessary work

Do you have a kid? If yes, you might know the feeling. If not, maybe you remember it: your dad asking you to clean your room—while your video games are calling. If you were like me, you might have tried to avoid it because honestly… a clean room didn't feel all that necessary.

Luckily, avoiding unnecessary work is not only possible in logging—it's considered good practice. Take the `log()` method: it's smart enough to check whether a message should be processed based on the log level. That's already helpful, but as we'll see in the following snippet, this alone isn't always enough—especially in the long run.

In the following code, the log level check happens three times:

Listing 3.3 This snippet performs three separate log-level checks at `INFO`

```
logger.log(System.Logger.Level.INFO, "A");
logger.log(System.Logger.Level.INFO, "B");
logger.log(System.Logger.Level.INFO, "C");
```

It's great that the `log()` method checks whether logging should occur. Doing this check multiple times in a row is still wasteful. Why check once initially if you expect certain log levels to be filtered?

`System.Logger` provides a method called `isLoggable()` for exactly this purpose. It checks whether a given log level is currently enabled, allowing you to avoid unnecessary comparisons and log method calls. With `isLoggable()`, you can decide early whether to execute a block of log statements.

The snippet below shows how to use `isLoggable()` in practice. Typically, we wrap multiple `log()` statements in an `if` block to avoid redundant checks. The condition is evaluated once; if logging is enabled, all messages inside the block will be executed.

Listing 3.4 This code checks if `INFO` is enabled before executing multiple `log()` statements.

```
if (logger.isLoggable(System.Logger.Level.INFO)) { #1
    logger.log(System.Logger.Level.INFO, "A"); #2
    logger.log(System.Logger.Level.INFO, "B");
    logger.log(System.Logger.Level.INFO, "C");
}
```

Although the performance gain can be ignored in small-scale systems, large systems can benefit significantly from avoiding unnecessary log-level checks. That said, there's a trade-off: if the log level is active (like `INFO`), then checking it manually with `isLoggable()` adds another comparison. So this pattern only makes sense when you're logging multiple messages or preparing expensive data for output.

Listing 3.5 Don't do this: unnecessary use of `isLoggable()` for a single log statement.

```
if (logger.isLoggable(System.Logger.Level.INFO)) {
    logger.log(System.Logger.Level.INFO, "A"); #1
}
```

If you're doing extra work to prepare log messages— like string concatenation or calculations—then `isLoggable()` becomes even more useful. Why pay the price for building a message no one will ever see?

In the following example, we prepare a `fullName` string using values from earlier in the business logic. This operation is cheap, but string formatting, data lookups, and JSON serialization could add costs to larger systems. Checking first with `isLoggable()` avoids wasted effort.

Listing 3.6 Only build log messages when the log level is active

```
String firstName = "Jane"; #1
String lastName = "Doe";

if (logger.isLoggable(System.Logger.Level.INFO)) {
    String fullName = firstName + " " + lastName
    logger.log(System.Logger.Level.INFO, fullName); #2
    logger.log(System.Logger.Level.INFO, "B");
    logger.log(System.Logger.Level.INFO, "C");
}
```

Watch out for pitfalls: using `isLoggable()` with larger blocks of code can

lead to subtle bugs. In the following example, can you spot the problem?

Listing 3.7 In large blocks, mistakes can go unnoticed

```
if (logger.isLoggable(System.Logger.Level.INFO)) { #1
    logger.log(System.Logger.Level.INFO, "A");
    logger.log(System.Logger.Level.INFO, "B");
    logger.log(System.Logger.Level.ERROR, "C"); #2
}
```

Have you spotted it?

In this snippet, an `ERROR '-level message is wrapped inside a check for the `INFO` level. This subtle mistake can lead to serious issues: if `INFO` is turned off, Even though it's critical, the `ERROR` message won't appear. The rest of the code may continue working, but you'll miss the most important problem.

Even worse, never include business logic inside an `isLoggable()` block unless it's directly related to logging. Codes like the following invite serious bugs: a database operation is hidden within an `isLoggable()` block. The code will only execute if the log level is set to `INFO`. That's a bad idea for production.

Listing 3.8 Nasty: the database service only runs when the log level is `INFO`

```
if (logger.isLoggable(System.Logger.Level.INFO)) { #1
    logger.log(System.Logger.Level.INFO, "A");
    logger.log(System.Logger.Level.INFO, "B");
    databaseService.save(); #2
    logger.log(System.Logger.Level.INFO, "C");
    logger.log(System.Logger.Level.INFO, "D");
}
```

Never write business code that depends on specific log levels. Be bold—add more `if` statements to ensure your code behaves as expected. Here's a better (and complete) example:

Listing 3.9 Having multiple `isLoggable()` blocks is perfectly fine

```
if (logger.isLoggable(System.Logger.Level.INFO)) { #1
```

```
    logger.log(System.Logger.Level.INFO, "A");
    logger.log(System.Logger.Level.INFO, "B");
}

databaseService.save(); #2

if (logger.isLoggable(System.Logger.Level.INFO)) { #3
    logger.log(System.Logger.Level.INFO, "C");
    logger.log(System.Logger.Level.INFO, "D");
}

logger.log(System.Logger.Level.ERROR, "C"); #4
```

3.10 Supplier saves Strings

If you enjoy modern Java syntax, you can also use Lambda expressions for logging. Lambdas are expressive and powerful, though some developers find them hard to read. Luckily, we won't be writing anything too complex here. Even if you're not a seasoned developer, using Lambdas with logging should feel straightforward.

Instead of using the `log()` method with a plain `String` that gets formatted right away, you can wrap the formatting logic in a Lambda expression. This way, the expensive `format()` call is only executed if the log level is active.

It might look a bit odd at first, especially with the extra brackets:

```
String userId = "12345";
logger.log(
    System.Logger.Level.INFO,
    () -> String.format("Processing request for user ID: %s", use
```

This Lambda uses the `Supplier` interface with a method called `get()`, which has been available since Java 8. A `Supplier'just returns a value. In our case, that value is a `String`. Perfect for logging.

The details of `Supplier` aren't important here. All that matters is that if the log level is enabled, the `log()` method will call the `Supplier` to get the message. Otherwise, the Lambda is ignored, and nothing is computed.

Calling `String.format()` may not be particularly expensive—it runs fast

enough in most cases. But imagine you're doing something more complex inside the Lambda—say, performance calculations.

The following code demonstrates how you could measure execution time between two points. First, we only record the current time if logging at the `INFO` level is enabled. Then, we log the duration using a Lambda that only runs when needed.

```
long start;
if (logger.isLoggable(System.Logger.Level.INFO)) { #1
    start = System.currentTimeMillis();
} else {
    start = 0;
}

// Do something else

logger.log(
    System.Logger.Level.INFO,
    () -> "Execution time: " + (System
    .currentTimeMillis() - start) + "ms"); #2
```

Lambdas can significantly improve performance. Especially when the operations required to build a log message are complex, and only needed at a certain log level.

Instead of always doing the work, you defer it. The Lambda will only execute if the logging framework decides the message should be logged.

3.11 Reading the output

When we run the application, the logging output is formatted similarly to what we saw in the previous chapter. Each log entry appears on two lines: The first includes the timestamp, class, and method name; the second contains the actual log message.

```
Mar 19, 2023 8:42:16 PM de.grobmeier.manning.logging.example1.Air
 execute
INFO: Started Airport App
Mar 19, 2023 8:42:16 PM de.grobmeier.manning.logging.example1.Air
 execute
```

```
WARNING: Language was null, setting to default (en)
Mar 19, 2023 8:42:16 PM de.grobmeier.manning.logging.example1.Air
 execute
SEVERE: Did not receive Airport service instance, returning
```

The formatting might not be beautiful, but it does its job: We can trace what happened and in which order.

Reading logs sucks—until it doesn't. Then, it becomes second nature. As you'll see in Chapter 5, some tools help you analyze logs.

3.12 Pros and Cons of System.Logger

`System.Logger` is a small but excellent library, specifically helpful for low-level libraries. Of course, it has its own set of pros and cons.

Here are some of the key benefits:

- It's part of the JDK — no third-party dependencies required
- It's easy to use — once you understand log levels, you're ready to go
- It requires no configuration — most setups work right out of the box
- It's lightweight - no bloat
- It's fast — in general, `System.Logger` performs well

But there are also some limitations:

- It's limited — only three log levels are available by default
- It lacks flexibility — custom configurations require additional integration work
- It's not extensible — no plugin system or advanced customization options
- It lacks a modern API — no placeholders or full lambda-style message formatting

You should consider the `System.Logger` if you want to write a library that developers can use for small and specific tasks. With `System.Logger`, users can decide whether to integrate with JUL or Log4j later.

You should consider using `System.Logger` if you're writing a library for small, focused tasks. It's a good fit for a lightweight project. Developers can later decide whether to integrate with JUL, Log4j, or another logging system.

However, `System.Logger` isn't suitable for large systems with advanced monitoring or logging requirements. While it can delegate to more powerful frameworks like JUL or Log4j, its native API is minimal. It doesn't let you configure where logs go, nor does it support plugins or advanced formatting features.

That said, `System.Logger` was never meant to be fancy. It's intentionally minimal. It just logs. And because it can integrate with other frameworks under the hood, it is a great start for logging with Java.

3.13 Summary

- `System.Logger` can serve as a simple starting point for smaller projects that don't need much power. JDK developers directly added it to the JDK.
- `System.Logger` offers only a straightforward, simplistic API.
- `System.Logger` is a simple framework that supports only three log levels out of the box.
- We can avoid expensive computations by using the `isLoggable()` method.
- `System.Logger` can integrate with other logging frameworks, such as JUL or Log4j, which enhances its configuration capabilities.

4 Exceptions: Try, Catch, Log

This chapter covers

- A deep dive into Exceptions
- Custom exceptions
- The difference between errors and exceptions

Imagine you're taking your usual Tuesday evening bath. You're singing at full volume when suddenly the door opens. Your partner enters with a guest. This was never supposed to happen. It's an exception in your personal program. If you don't handle it well, it leads to an awkward situation.

And the same is true in code: If you ignore exceptions, your program may crash or run in an unstable state.

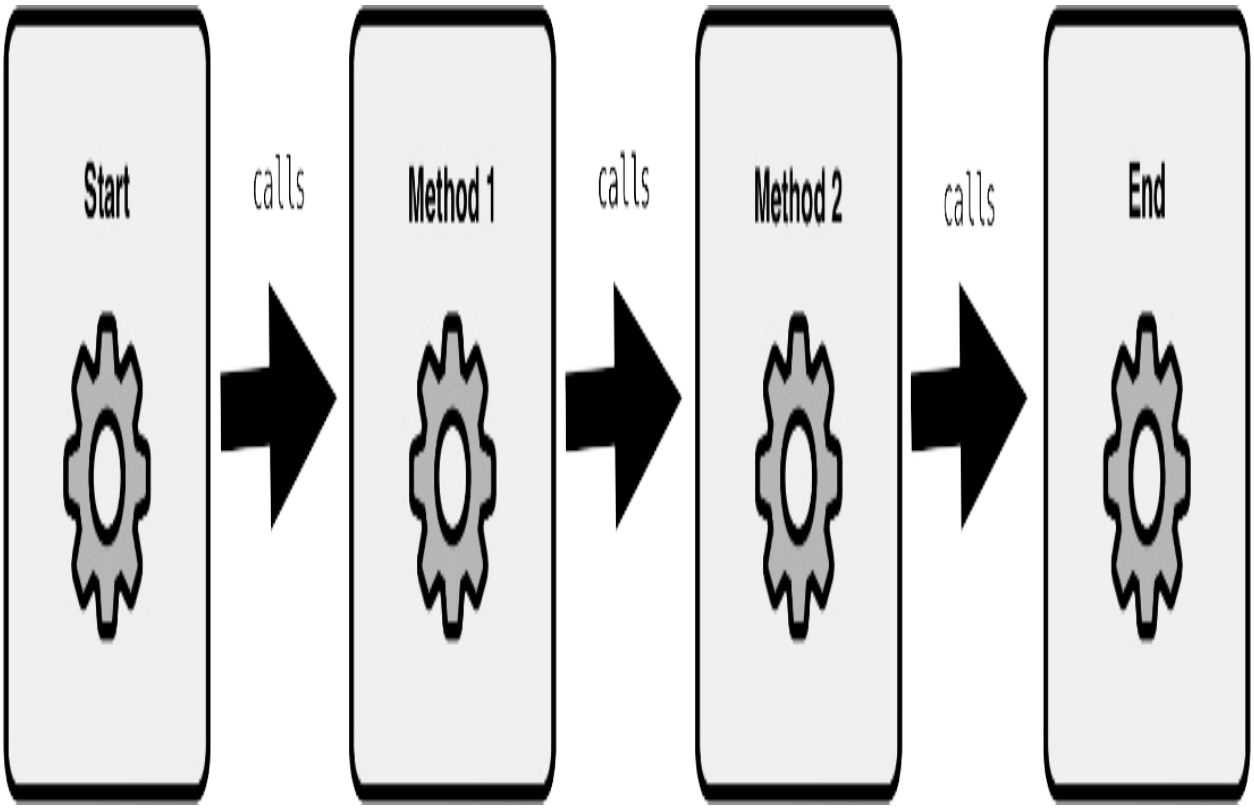
Even if you handle them without logging, you might swallow essential clues. To log exceptions well, you have to understand how they work. That's why this chapter covers not just logging, but also the fundamentals of exceptions.

4.1 What are exceptions?

While `if` and `else` offer control and predictability, exceptions can disrupt the normal flow of a program in unexpected ways. Beginners sometimes avoid them for that reason. Unfortunately, if handled poorly, it is easy to flood the logs with unnecessary information accidentally.

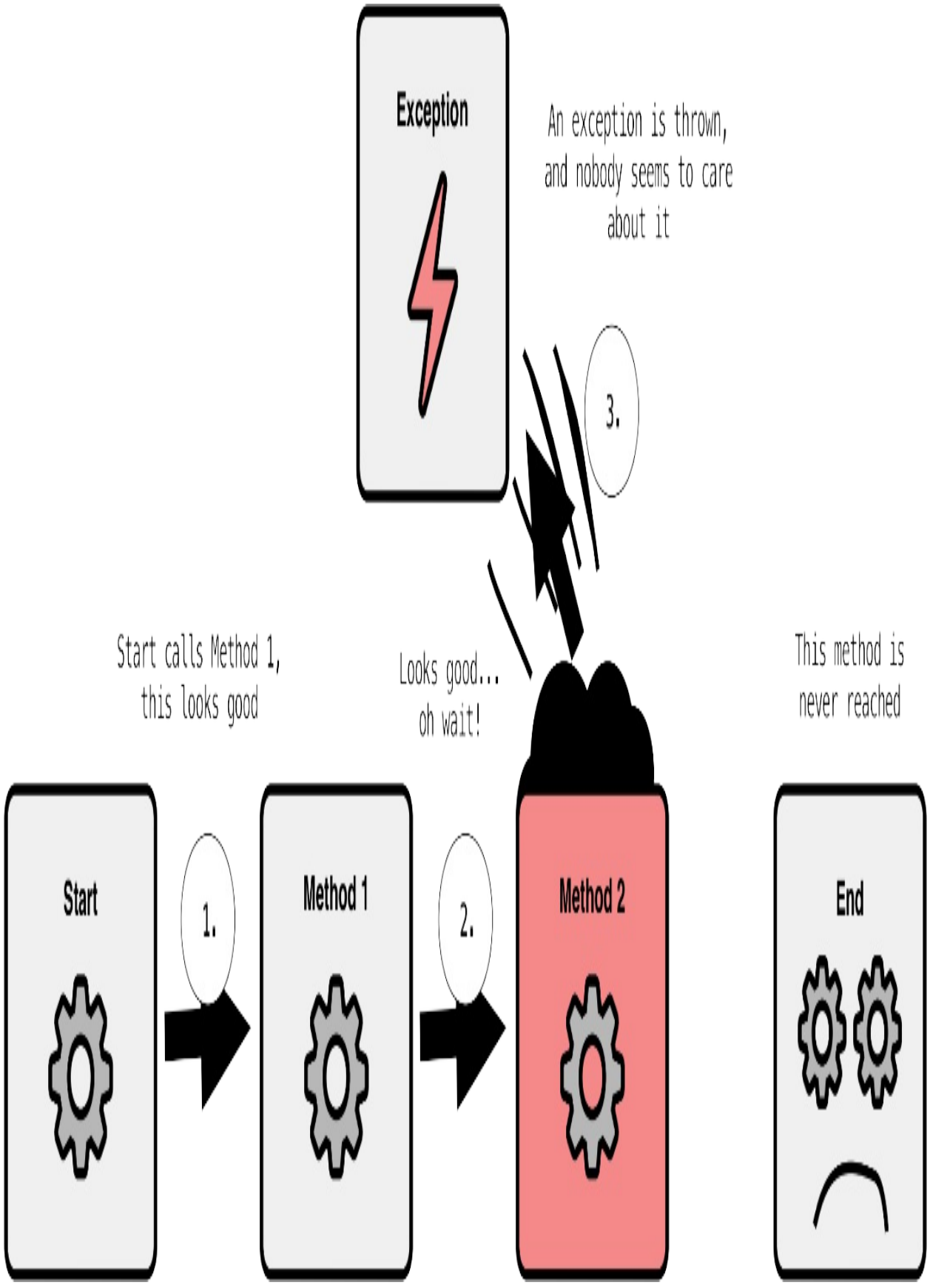
To understand exceptions, we should first take a look at the normal procedural flow of a program. It starts at one point, passes through several steps, and ends in a defined state.

Figure 4.1 Normal program flow



Exceptions are a concept in Java that represent situations that can occur, but are usually not expected. Joshua Bloch, the author of "Effective Java," refers to them as "exceptional conditions". A wrong password is not an exception; it happens often and hardly qualifies as exceptional. What could be exceptional?

Figure 4.2 Throwing an exception


Exception
An exception is thrown, and nobody seems to care about it
3.
Start calls Method 1, this looks good
Looks good... oh wait!
This method is never reached
Start
1.
Method 1
2.
Method 2
End

Choosing what constitutes exceptional is challenging. To determine what's normal, we need to consider what a method is intended to ensure.

Every method has *preconditions* and *postconditions*. A precondition is a condition that must be true before a method can run. If you write a method to calculate the new heading of an aircraft, the precondition is that the current heading is between 0 and 359 degrees.

A postcondition must be true after the code has run. After the calculation, the postcondition would be that the new heading is also between 0 and 359 degrees. That's the value we want to return. If we cannot fulfill the postcondition, we can consider this an exceptional condition, often commented on with "this should never happen" by developers.

```
public int calculateNewHeading(int currentHeading, int correction
    if (currentHeading < 0 || currentHeading > 359) { #1
        throw new IllegalArgumentException("Invalid current headi
    }

    int newHeading = (currentHeading + correction) % 360; #2

    if (newHeading < 0 || newHeading >= 360) { #3
        throw new IllegalStateException("Heading calculation out
    }

    return newHeading;
}
```

This example is simple, but identifying preconditions and postconditions in real systems is usually much harder. Imagine an online check-in system for an airline, which may need to check for seat availability, payment status, or even passenger age.

Postconditions can be equally complex. Returning a valid seat number might seem straightforward, but what if another customer grabs the seat just milliseconds earlier? In such cases, an exception might be the only way to signal that the expected outcome couldn't be fulfilled—a scenario that logging will eventually need to capture. Exceptions contain a message that describes the error. It also contains a stack trace. When printed, it reveals the chain of method calls that led to it. They often appear in log files.

In Java, there are two flavors of exceptions: checked and unchecked. Let’s take a closer look.

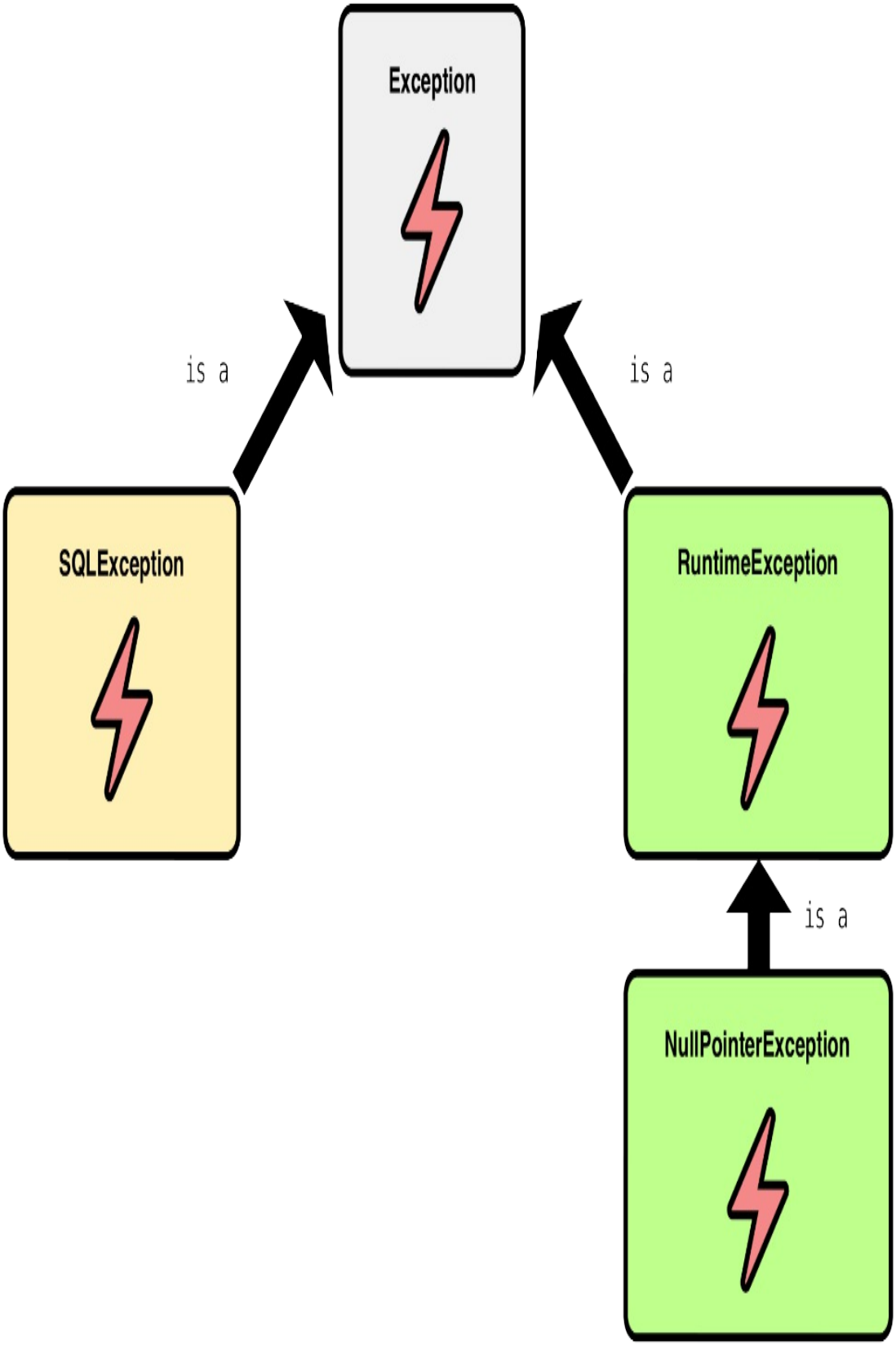
4.2 Checked vs. unchecked exceptions

In Java, there are two main categories of exceptions: checked and unchecked exceptions. All exceptions extend the class `java.lang.Exception`. This makes them a so-called *checked exception*. Except: any exception that inherits from `java.lang.RuntimeException` is considered an *unchecked exception* (also often just called a *runtime exception*). `RuntimeException` acts as a switch. If it’s there, it’s unchecked; if it’s not, it’s checked.

If you are not sure which type of exception you are dealing with, you can inspect the inheritance hierarchy of the exception. If your editor doesn’t show you directly, you can inspect the class hierarchy by Ctrl+clicking the exception name in most development environments (or Cmd+click on macOS).

In the following image, the exception hierarchy is shown. On the left, you’ll see `SQLException`. Since it extends `Exception` directly, it is a checked exception. On the right, you’ll see `NullPointerException`, which extends `RuntimeException` and is therefore an unchecked exception.

Figure 4.3 Hierarchy of common Java exceptions


Exception
is a
is a
SQLException
RuntimeException
is a
NullPointerException

In a log file, all exceptions look the same. However, how developers treat them differs. Let’s look at the checked exceptions first, as they have stricter rules.

4.2.1 Checked Exceptions

Developers use checked exceptions to signal potential problems and ensure the caller handles them. If the caller tries to ignore it, the compiler throws an error.

Here are the two rules for checked exceptions:

- If you call a method that might throw a checked exception, you must deal with it. This is called *handling* the exception. You do this with a try-catch block.
- If you don’t want to handle it right away, you can pass it on. In that case, you must *declare* the exception using the `throws` keyword in the method signature. It’s not handled yet.

In short: you must *handle or declare* checked exceptions. Let’s look at the simpler case first: declare.

Declaring an exception

Let’s use `SQLException`, a typical database exception included in the JDK. It often occurs when the database is not reachable or the database query is invalid. Here’s a simplified version of its definition, which clearly shows that it is a checked exception.

```
public class SQLException extends java.lang.Exception { #1
    // ...
}
```

Now, imagine we are calling a method from the `Connection` class that may throw a `SQLException`. As per the checked exception rules, I must either handle or declare it,

In the following examples, we assume a logger object exists, which is likely created with `System.getLogger()`. If we write a method that takes a `Connection` object and reads from a database, we have to deal with it. `createStatement()` and `executeQuery()` both might throw a `SQLException`. As it appears now, the compiler will refuse to compile it.

```
public ResultSet read(Connection connection) {
    logger.log(System.Logger.Level.INFO, "Start");
    Statement statement = connection.createStatement(); #1
    logger.log(System.Logger.Level.INFO, "End");
    return statement.executeQuery("SELECT * FROM my_table");
}
```

One solution is to declare the exception and add `throws SQLException` to the method signature. Now the `read()` method itself signals a possibility to throw an exception, even when this isn't guaranteed to happen.

```
public ResultSet read(Connection connection) throws SQLException
    logger.log(System.Logger.Level.INFO, "Start");
    Statement statement = connection.createStatement(); #2
    logger.log(System.Logger.Level.INFO, "End");
    return statement.executeQuery("SELECT * FROM my_table");
}
```

Whoever uses the `read()` method now has to deal with the potential `SQLException`. The caller must either handle it with a `try/catch` block or redeclare it. Thanks to this requirement to declare checked exceptions in the method signature, the potential exception is already visible.

Callers are informed of potential issues and can adjust their logging accordingly. For example, they might log the input parameters or context. Eventually, developers must handle the exception—otherwise, the application will fail.

Handling an exception: try/catch

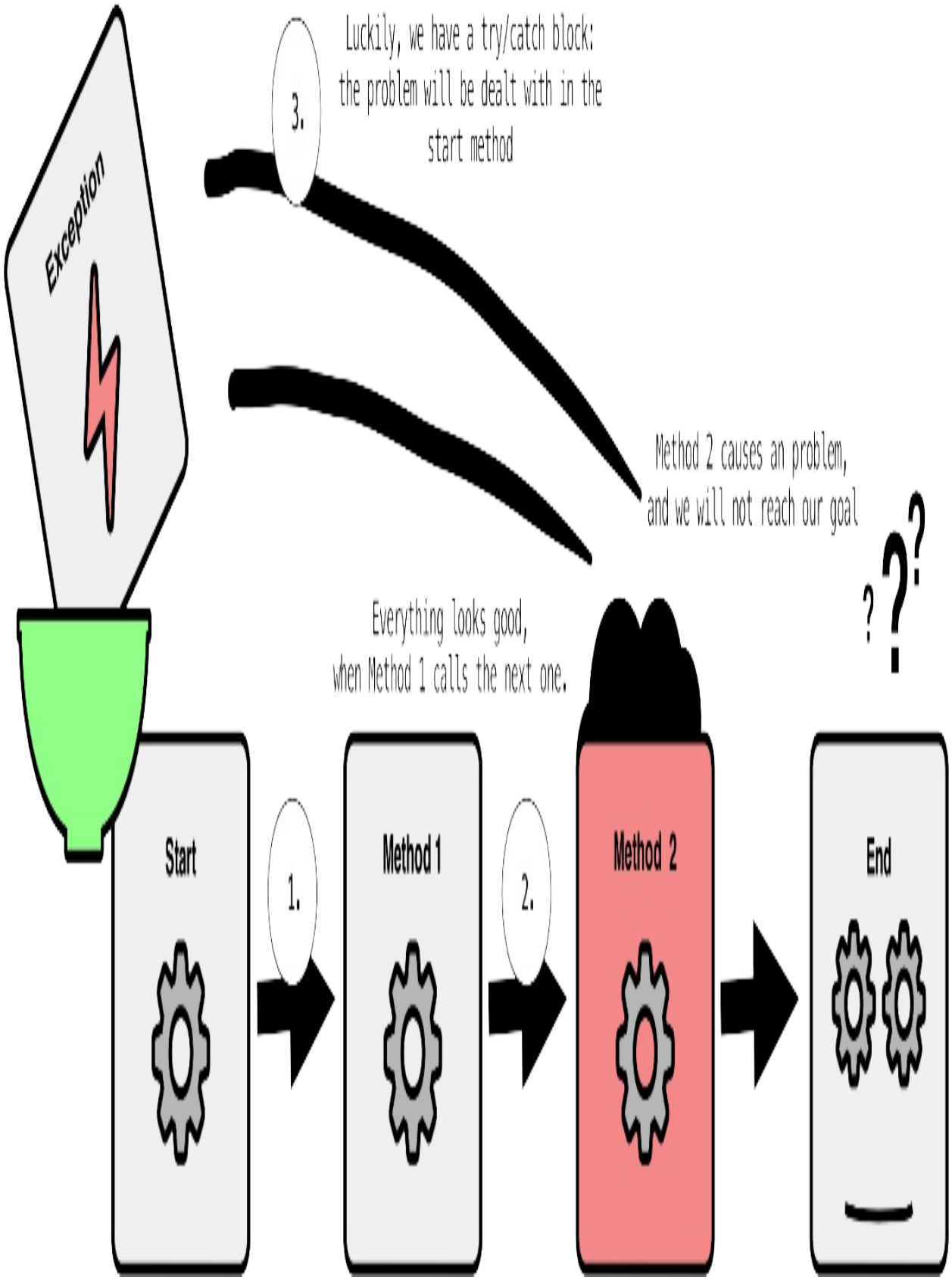
If we only passed exceptions to their caller, they would end up at the `main()` method. If they are even passed there, the program ends. To prevent that, we can handle the problem.

If you know what to do when an exception occurs, you can handle it. You might retry the request, reconnect to the database, or log the error and continue. When an exception is handled, the issue is resolved, and no further action is needed.

In Java, we use `try-catch` blocks for that. The `try` block contains the code that might throw an exception. The `catch` block contains the code that handles the exception. Most often, you'll log the exception with context information in the catch block.

In the image below, an exception is thrown in method 2, so the program never reaches its normal end. The class that initiated the process is prepared to catch the exception and determine the next course of action.

Figure 4.4 Catching an exception


Exception
3.
Luckily, we have a try/catch block:
the problem will be dealt with in the
start method
Method 2 causes an problem,
and we will not reach our goal
? ? ?
Everything looks good,
when Method 1 calls the next one.
Start
1.
Method 1
2.
Method 2
End

This example demonstrates how to use a `try/catch` construct, while omitting details to focus on the essentials. It's expected that `Start` and `End` will be logged; however, if `executeQuery()` throws an exception, `End` will be skipped, and only `Catch` will be logged.

```
public void read() {
    try { #1
        logger.log(System.Logger.Level.INFO, "Start");
        statement.executeQuery(); #2
        logger.log(System.Logger.Level.INFO, "End");
    } catch (SQLException e) { #3
        // handle the exception
        logger.log(System.Logger.Level.INFO, "Catch"); #4
    }
}
```

After the program catches the problem, it can continue as usual. For that reason, every catch block should include a log message. As a general rule, never leave a catch block empty. At a minimum, log what happened.

This was a very typical example of how Java exceptions and logging work together—except, of course, our log messages could be better. We will improve that now.

Logging an exception, the right way

In the previous `catch` block, we logged the word "Catch". Reflecting on the 6+2 key questions we discussed earlier, you might find it lacking.

```
try {
 ...
} catch (SQLException e) {
    logger.log(System.Logger.Level.INFO, "Catch"); #1
}
```

Just logging `Catch` isn't helpful. It tells us nothing about what happened, where it happened, or how serious the issue was. The problem was identified, and the program will continue to run; however, no meaningful information was recorded. Since the message was written at the `INFO` level, it doesn't even appear to be something unusual that happened.

A better approach is to write a proper message, log at the `ERROR` level, and include more context. This makes the log more useful for debugging and even for monitoring tools. Fortunately, the `log()` method can accept an exception as a parameter, which provides exactly what we need.

```
try {
 ...
} catch(SQLException ex) { #1
    logger.log(
        System.Logger.Level.ERROR,
        "We had an issue with the database",
        ex); #2
}
```

The `Exception` class defined a method called `printStackTrace()`. You have likely seen its output in the console. Logging frameworks do something similar, but instead of printing to `System.err`, they send the stack trace to your preferred log destination. We have already seen how this can be done with `Appender` in the previous chapter.

If you're not yet familiar with reading stack traces, don't worry. We'll get to them soon.

Many years ago, I thought this knowledge was already sufficient—until one day, I caught the wrong exception. That moment reminded me: even exceptions can extend other exceptions, and we need to be careful about which ones we catch.

Handling the right exceptions

Logging exceptions in a hierarchy can be tricky. If you catch an abstract or generic exception, it will also see all its subclasses. This becomes clearer when we define two exceptions. The first is a basic one that extends directly from `Exception`, making it a checked exception.

```
public class AircraftException extends Exception {
}
```

The second exception is specialized for military aircraft and implies a more critical situations. It deserves its own exception type, as we plan to handle it

differently.

```
public class MilitaryAircraftException extends AircraftException
}
```

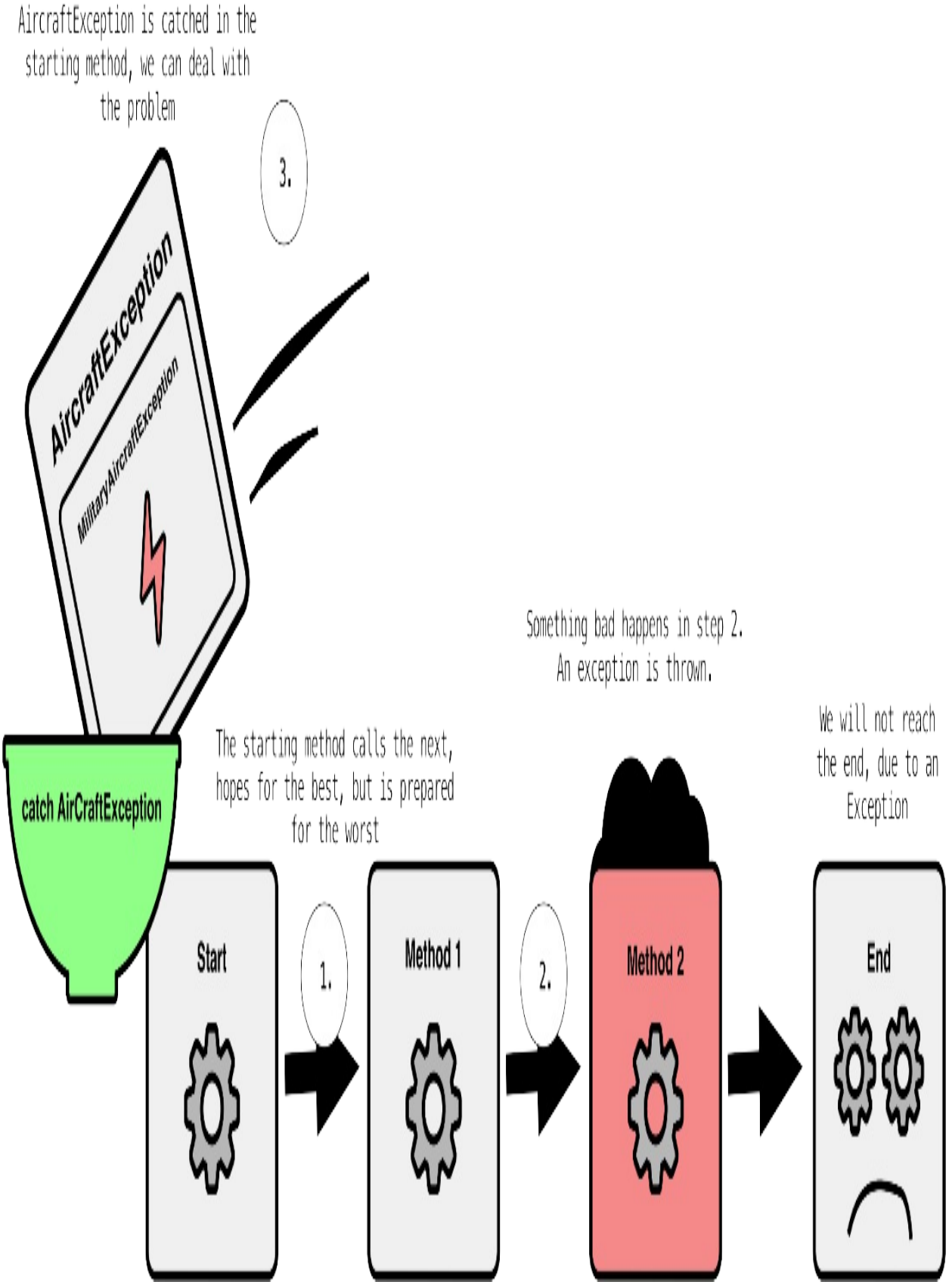
We need to decide whether we want to catch the first exception, the second one, or both. If we catch `AircraftException`, we will also see its subclass `MilitaryAircraftException`. The following code shows that.

```
public void startEngines() throws MilitaryAircraftException {} #1

public void takeOff() {
    try {
        startEngines(); #2
    } catch (AircraftException e) { #3
        logger.log(System.Logger.Level.ERROR, "Aircraft error", e
    }
}
```

Since `MilitaryAircraftException` **is a** subtype of `AircraftException`, it gets caught, even if we didn't intend that.

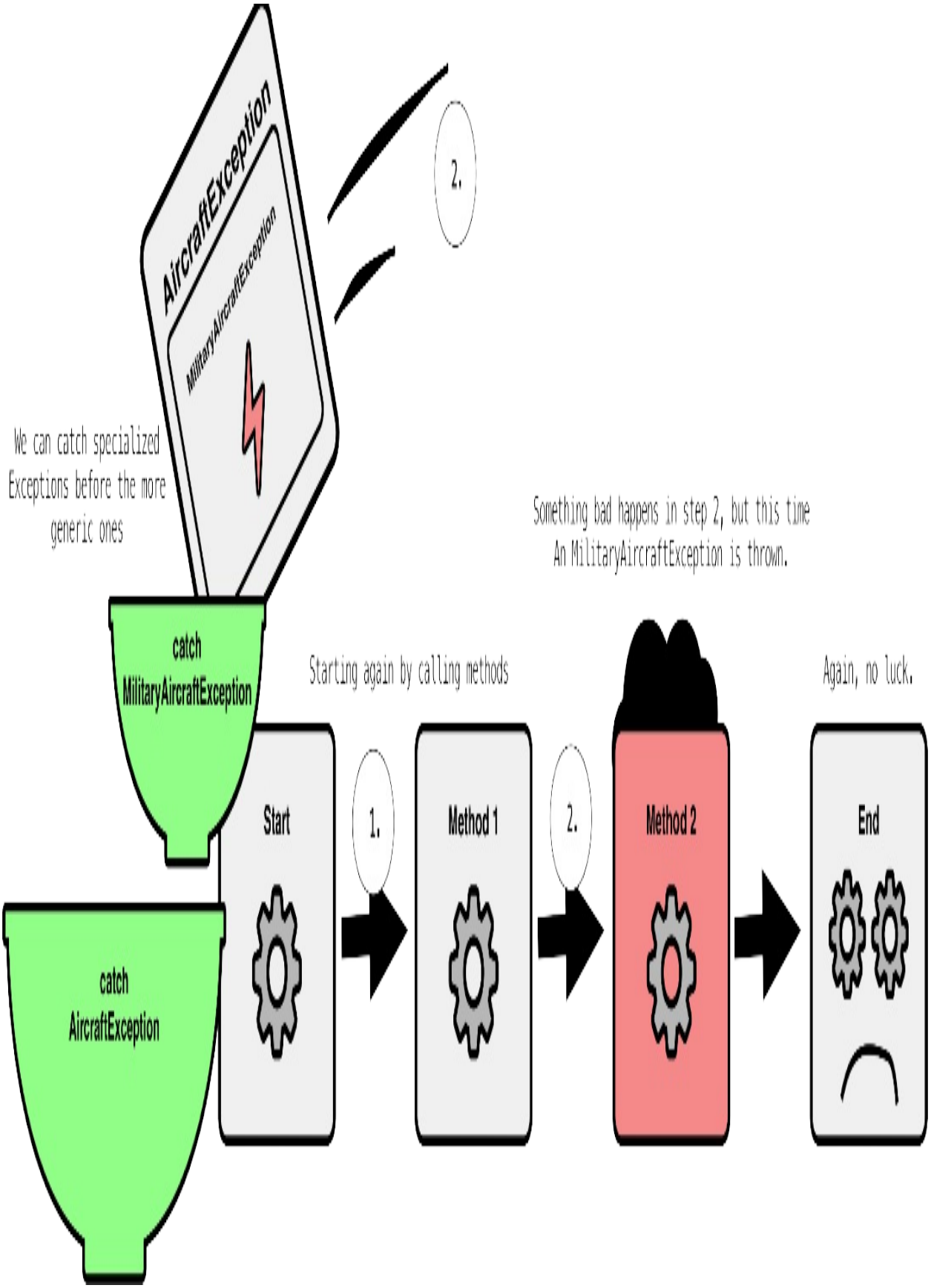
Figure 4.5 Catching a military aircraft exception


AircraftException is catched in the starting method, we can deal with the problem
3.
AircraftException
MilitaryAircraftException
catch AirCraftException
The starting method calls the next, hopes for the best, but is prepared for the worst
Something bad happens in step 2. An exception is thrown.
We will not reach the end, due to an Exception
Start
1.
Method 1
2.
Method 2
End

While the code is technically correct, the message is vague. If a `MilitaryAircraftException` occurs, it might be a critical issue that requires immediate attention. All we logged was `Aircraft error`, which doesn’t convey the severity of the situation.

If you need to handle both exceptions differently, you must use separate `catch` blocks. Think of it as a "catch before the catch". First, you handle the more specific one, then the general.

Figure 4.6 Catching multiple exceptions


AircraftException
MilitaryAircraftException
2.
We can catch specialized Exceptions before the more generic ones
Something bad happens in step 2, but this time An MilitaryAircraftException is thrown.
catch MilitaryAircraftException
Starting again by calling methods
Again, no luck.
Start
1.
Method 1
2.
Method 2
End
catch AircraftException

The following code shows how you can catch both exceptions separately. Additionally, we are logging two different messages at two distinct levels. `FATAL` should be used only for the most critical issues that might cause the system to crash.

```
public void takeOff() {
    try {
        startEngines(); #1
    } catch (MilitaryAircraftException e) { #2
        logger.log(System.Logger.Level.FATAL,
        "Critical military aircraft failure", e); #3
    } catch (AircraftException e) { #4
        logger.log(System.Logger.Level.ERROR, "Aircraft startup f
    }
}
```

Please note that the order of these catch blocks matters. If you place the generic one first, the specific one would never be reached—and the code wouldn’t compile.

For logging, we should always ask ourselves: Does this exception need special handling, visibility, or alerting? If this is the case, we should consider catching and logging it separately.

As we’ve now learned the most important rules for checked exceptions, It’s time to examine the rules—or lack thereof—for unchecked exceptions.

4.2.2 Unchecked Exceptions

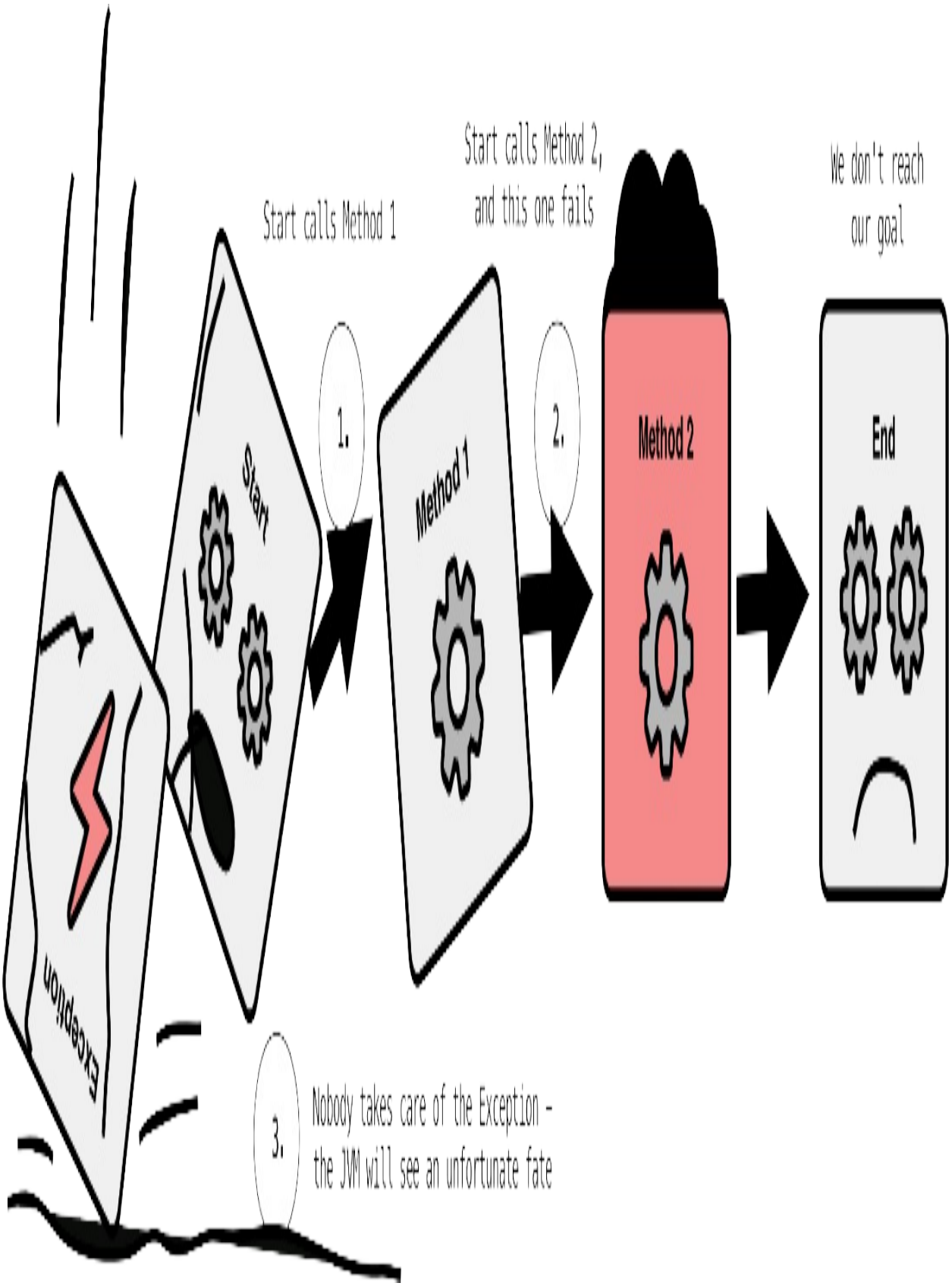
For checked exceptions, the rule is clear: handle or declare. For unchecked exceptions, a third option becomes available: ignoring them.

Unchecked exceptions include all exceptions that extend `RuntimeException` directly or indirectly. `RuntimeException` itself extends `Exception`, but it, and all its subclasses, have special rules: You don’t need to declare them in the method signature, and you’re not required to catch them. You can handle them, though, and you can declare them like checked exceptions.

They often occur unexpectedly, like `NullPointerException’s or`

`ArrayIndexOutOfBoundsException’s. And that makes them dangerous: if you don’t handle them, they will propagate up the call stack until they reach the `main()` method. If no one catches them, the program will terminate.

Figure 4.7 Ignoring an exception


Start calls Method 1
Start calls Method 2,
and this one fails
We don't reach
our goal
1.
2.
Start
Method 1
Method 2
End
Exception
3.
Nobody takes care of the Exception –
the JVM will see an unfortunate fate

Both exceptions usually indicate a programming error. Here's an example of an unchecked exception in action—one that is easily overlooked.

```
public static void main(String[] args) {
    String[] passengers = new String[] { #1
        "Christian",
        "Linda",
        "Olivier"
    };

    logger.log(
        System.Logger.Level.INFO,
        "Name of passenger: " + passengers[4] ); #2
    }
}
```

Arrays are zero-based, and the fourth element does not exist. The code will compile successfully, but it crashes at runtime with an `ArrayIndexOutOfBoundsException`, and no prior warning is given.

You could explicitly declare a `NullPointerException`. It is neither necessary nor helpful in most cases. If you were to implement a rare case like this, you could signal a possible `NullPointerException` to the caller. It might serve as a form of documentation.

```
public void readFromBlackBox() throws NullPointerException { #1
    String passengerName = null; #2
    int length = passengerName.length(); #3
}
```

If a caller sees `NullPointerException` in the method signature, they might be more cautious and add a `try/catch` block around the call.

```
public void read() throws NullPointerException {
    try {
        readFromBlackBox();
    } catch (NullPointerException npe) { #1
        logger.log(System.Logger.Level.ERROR, "NPE", npe); #2
    }
}
```

Declaring a `NullPointerException` explicitly is unusual, and so is catching them. Some other unchecked exceptions, though, might make more sense.

The Spring team makes a lot of use of unchecked exceptions in their framework to avoid excessive `try/catch` blocks.

While you can catch unchecked exceptions, it's often a better idea to cover them with unit tests instead of surrounding every line with `try/catch`. This approach helps keep your code clean and focused, without wrapping every call in exception handling.

Unit Testing

Unit tests not only help prevent runtime exceptions, but they also reduce noisy log files by catching issues before they reach production.

Unit tests are small pieces of code that check individual components of your program. They are typically written by developers and run automatically, often when a new change is pushed to a shared code repository, such as Git.

The most popular framework for writing unit tests in Java is JUnit.

There's an ongoing debate in the Java community about checked versus unchecked exceptions, especially around readability and clarity. There is no clear consensus, which means we will encounter both types in real-world code. Fortunately, throwing works the same.

4.3 Throwing exceptions

One of the most common reasons to throw an exception is when a user is misusing your class. Letting your code fail fast is usually considered a good practice and is often called "defensive programming".

Defensive programming

Defensive programming is a technique employed by developers, especially in security contexts. In short, it would mean your code fails early and fails fast when it runs into unexpected input. For programming, it means you would find issues early on. For security, it means you would not allow any unexpected input to enter your system.

We have already learned about the `throws` keyword. With `throws`, we signaled a potential problem, but we log when we handle the actual situation. With *`throw`*, we actually raise the problem.

The keywords are both tied tightly to a specific class called `Throwable`. `Throwable` is the superclass of all errors and exceptions in Java and defines important methods like `getMessage()` and `printStackTrace()`. You'll find more details on the throw keyword in the appendix if you're curious— and more about stack traces in the next chapter.

Let's look at an example. It is a good idea to reuse Exceptions that are already built into Java. Two exceptions stand out, `IllegalArgumentException` and `NullPointerException`. Both are `RuntimeException's.

Let us assume you are writing a method that takes a `String` as an argument. In the following example, the code checks if the input is empty or longer than 34 characters. If it is the case, we would throw an `IllegalArgumentException`.

```
public void writeBlackBoxEntry(String input) { #1
    if (input.isEmpty() || input.length() > 34) { #2
        throw new IllegalArgumentException("Bad input"); #3
    }
    // ...
}
```

This exception message can be improved by adding the actual length of the input. We can expect that a developer will log this message at some point.

```
public void writeBlackBoxEntry(String input) {
    if (input.isEmpty() || input.length() > 34) {
        throw new IllegalArgumentException(
            "Bad input, length: " + input.length()); #1
    }
    // ...
}
```

4.3.1 Antipattern: Log everything

Logging values is helpful—but only if you validate your input first. A

common antipattern is logging input values before checking them. At first glance, the following snippet appears harmless, but it could cause a serious problem.

```
public void writeBlackBoxEntry(String input) {
    logger.log(System.Logger.Level.INFO, "Input: " + input); #1
    if (input.isEmpty() || input.length() > 34) { #2
        throw new IllegalArgumentException(
            "Bad input: '" + input + "', length: " + input.length
    }
    // ...
}
```

You must never log input variables without first validating them. Doing otherwise may lead to log injection attacks, where an attacker injects special characters to manipulate your logs. Or consider the impact of logging an input string that’s 10 MB long.

The solution is simple: validate first. And if you throw an exception after validation, you don’t have to log them: The handler will take care of it.

`NullPointerException` versus `IllegalArgumentException`

You may have noticed that I didn’t check for null in the example above. That’s intentional. When receiving an unexpected null value, you should not throw `IllegalArgumentException`.

Instead, throw a `NullPointerException`, even if you consider it an illegal argument.

While this might feel unintuitive at first, it’s the Java-recommended approach. Joshua Bloch, author of "Effective Java", writes: "If a caller passes null in some parameter for which null values are prohibited, convention dictates that `NullPointerException` be thrown rather than `IllegalArgumentException`."

Please note that, before throwing the exception, I didn’t log anything. While it is tempting, you don’t need to log when you throw an exception: We would assume whoever handles the exception will write to the logs. The same is true for checked exceptions.

Resist the temptation to log too early. First, validate, and if you must throw an exception, don’t blindly add the input value to the exception message. After all, the handler might log it appropriately.

4.3.2 Antipattern: Log and throw

Another antipattern is logging an exception and then rethrowing it. In the following code, the developer logs the exception. Because no recovery action was taken, they chose to rethrow the same exception.

```
class Aircraft {
    public void readFromBlackBox() throws NullPointerException {
        // ...
    }

    public void read() throws NullPointerException {
        try {
            readFromBlackBox();
        } catch (NullPointerException npe) {
            log.error("NPE", npe); #2
            throw npe; #3
        }
    }
}
```

Logging should be done by the part of the code that handles the problem. It is tempting to catch, log, and throw again, especially during debugging. However, this approach risks duplicate log entries and misleading stack traces. You may end up logging the same exception multiple times, which can clutter your log files.

Too much information can be as harmful as too little. And just as we risk overlogging with this antipattern, some developers fear losing important details when exceptions are wrapped.

4.4 Wrapping exceptions

In recent years, it has become popular to wrap checked exceptions into unchecked ones to avoid boilerplate code. Modern frameworks, such as Spring, often do this when working with older libraries that still promote the

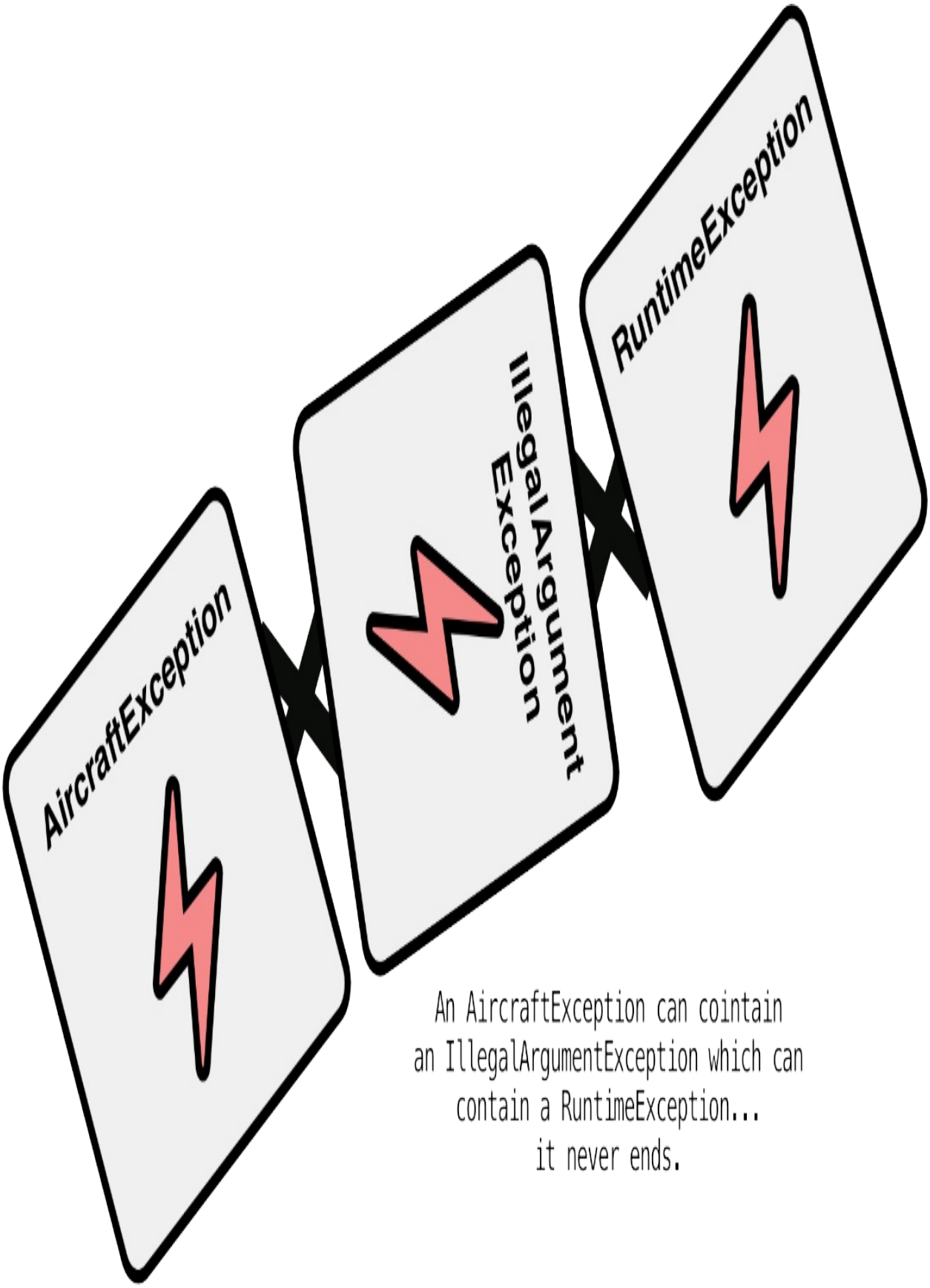
use of many checked exceptions.

In the following example, you'll see how an `SQLException` is wrapped inside a `RuntimeException`. This technique is supported by Java and known as *exception chaining*. Luckily, it doesn't lose any information because the original exception is passed into the new one as a cause. Again, we don't need to log the original exception if we wrap it correctly, assuming the handler will log it. Simple rule: who handles, logs.

```
public void createBlackBoxEntry(String input) {
    try {
        statement.executeQuery("SELECT * FROM blackbox"); #1
    } catch (SQLException e) {
        throw new IllegalArgumentException("Something is wrong",
    }
}
```

That way, exceptions can start looking like a Sushi roll: one exception wrapped in another, then another… Like with Sushi, nothing is lost, and you can easily access the wrapped information.

Figure 4.8 Chaining multiple exceptions



An AircraftException can cointain
an IllegalArgumentException which can
contain a RuntimeException...
it never ends.

Luckily, most logging frameworks can handle this. They walk the chain of causes using the `getCause()` method from `Throwable` and print all the information about each exception in the chain.

```
public void blackbox() {
    try {
        createBlackBoxEntry("data"); #1
    } catch (IllegalArgumentException e) {
        log.error(e); #2
    }
}
```

In the log file, you'll read something like this. First, our own message, Second, the one or many "caused by" sections, which show the original exceptions.

```
ERROR: Something is wrong
 at ...
Caused by: java.sql.SQLException: ...
 at ...
```

Exception chains and stack traces

Modern logging frameworks automatically walk the entire chain of exceptions. There's no need to log each layer manually.

The `Caused by` line shows the original exception. Thanks to this kind of wrapping, we can preserve context. Logging frameworks can print the entire context for us. However, when you wrap a checked exception in a RuntimeException, the caller may no longer be required to handle it and might miss it entirely.

There's one part of `try/catch` you can't ignore or bypass. It's the final bolt in the mechanism: `finally`.

4.5 Finally: finally

The `finally` keyword is also part of the `try/catch` construct. It will always be executed, regardless of whether an exception occurred or not. Only

`System.exit()` prevents the `finally` block from running. You can see the syntax in the following code example. First, a normal piece of code that might throw an exception. Second, a catch block to handle it. And finally, a `finally` block that always runs.

```
public void readFromDatabase() {
    Connection connection = null;
    try {
        connection = getConnection(); #1
        Statement statement = connection.createStatement();
        statement.executeQuery("SELECT * FROM blackbox");
    } catch (SQLException e) { #2
        logger.log(
            System.Logger.Level.ERROR,
            "Something went wrong", e);
    } finally {
        logger.log(
            System.Logger.Level.INFO,
            "Connection closed."); #3

        if (connection != null) { #4
            connection.close();
        }
    }
}
```

`finally` is often used to clean up resources. A resource is something that needs to be closed when you are done with it, like a database or files. Databases can only handle a limited number of connections, and operating systems only handle a limited number of open files.

4.5.1 Problems with finally

There are problems with the code above. First, we always log "Connection closed", even if it wasn’t closed. We may have encountered an exception in the past, and the connection was never established. Second, if `connection.close()` throws an exception, we will have misleading logs that pretend our connection was closed successfully, but then you see another exception in the log file. In fact, it could be the case that the connection was never successfully opened in the first place, and so you might end up debugging for hours to find a problem that never happened.

In the `finally` block, we should strive to minimize logging. If possible, we should avoid the finally block altogether.

4.5.2 Modern Java: try-with-resources

To avoid the problems above, Java offers a modern way to close resources: `try-with-resources`. It makes resource cleanup safer and helps keep your log files tidy.

The trick is to open the resource in the try block. Java will then automatically close it when the block ends.

With `try-with-resources`, our code becomes significantly simpler. We open the connection in the try block, and no `finally` block is needed since `close()` is always called automatically. The temptation to also log there is also gone. We don’t only clean up the code, we clean up the logs. No more false "Connection closed" messages. No need to double-check whether your logs are lying. This keeps your log files tidy.

```
public void readFromDatabase() {
    try (Connection connection = getConnection()) { #1
        Statement statement = connection.createStatement();
        statement.executeQuery("SELECT * FROM blackbox");
    } catch (SQLException e) {
        logger.log(
            System.Logger.Level.ERROR,
            "Something went wrong", e);
    }
}
```

Don’t log blindly in `finally` blocks. If you can avoid it, do it. We might lose some control over how the resource is closed, but in most cases, this is not a problem. Quite the opposite: the code is much smaller and easier to read. Additionally, we avoid lying logs when something failed before it was correctly initialized. If we must use `finally`, we should also check whether the operation actually succeeded, or better, let `try-with-resources` handle it. With this, we have almost completed our tour of exceptions. Only one thing remains: their dark twin, errors.

4.6 Errors

An Error is the most dangerous kind of failure in Java. When this occurs, the *JVM* (Java Virtual Machine) is in an unrecoverable state. Like exceptions, errors can be thrown, but you should never throw or catch one yourself: this is the JVM's domain. Like exceptions, errors extend the `Throwable` class.

You might have seen `StackOverflowError` or `OutOfMemoryError`. The first can indicate an infinite recursion (a method that repeatedly calls itself). The second one tells you that you have run out of memory.

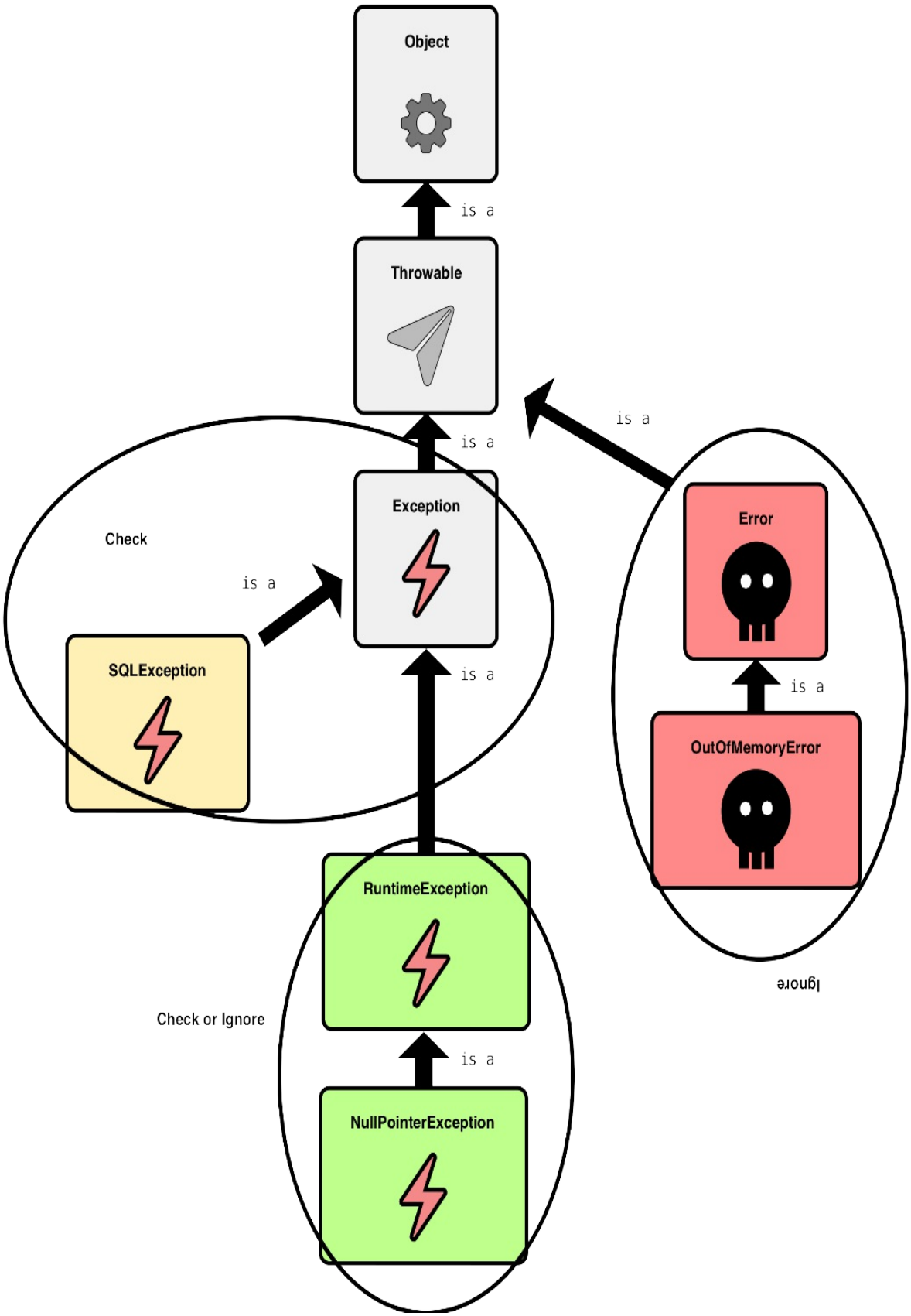
As Java developers, we can't manage memory directly. And we can't break out of recursion once it's running. By the time these errors happen, the JVM is already on its way down. All you can do is fix your code.

Even if you try to catch an error, there are no guarantees your `catch` block will run. The JVM might terminate before it gets there.

Because of this, there is no point in logging any error: even if your logger is called, it might never write anything.

You should not write code that throws them, declares them in your method signature, or catches them at all. For the sake of completeness, I added the errors to my image.

Figure 4.9 Complete exception and error hierarchy


Object
is a
Throwable
is a
is a
Exception
Check
is a
SQLException
is a
Error
is a
OutOfMemoryError
Ignore
RuntimeException
Check or Ignore
is a
NullPointerException

However, don’t bother logging or catching them. There is no point. An error means the end of your program. All you can do now is debug your code.

4.7 Why debugging is not enough

You’ve made it this far and might be wondering: What’s the difference between debugging and reading stack traces in log files?

Both tools let you follow the execution of your program, but they work in different ways. Debugging is what you do in real-time. You can stop at breakpoints, step through the code, and inspect or even change variable values while the program is running. It is like the present tense in a language. Log files, by contrast, are a snapshot of the past. You can’t change anything —past tense.

When you actively work on your code with full access to the source code and a local environment, debugging is the best tool. On a production system, you can’t attach your IDE, stop the program, or step through it. Then logs are your only option. Both are different tools for different situations during development.

Now that we understand how exceptions bubble, get wrapped, and show up in stack traces, let’s look at some tools that help us read our log files more effectively.

4.8 Summary

In this chapter…

- Checked and unchecked exceptions serve different purposes
- Handling, declaring, or ignoring exceptions are the three options we have
- Avoid the "catch-log-throw" and "log-everything" antipatterns
- Wrapping exceptions is no problem for logging
- `try-with-resources` often beats `finally`

5 How to read log files

This chapter covers

- How to read stack traces effectively
- Why the command line is a log file's best friend
- Tools to read log files efficiently

Previously, we learned how to create log files using `System.Logger` and briefly mentioned stack traces. Now it's time to explore the other side of logging: reading and interpreting what we've written.

Imagine if you opened the logs of a production system with Notepad. These files could easily grow to hundreds of megabytes, sometimes even gigabytes. Notepad loads the whole file into memory, which will likely fail. Even if it worked, you'd have to scroll through thousands of lines before finding a trace of the problem. If you find it, it comes in the form of a stack trace. Stack traces are chains of method calls that led to an exception. They are generated from Java and often printed into a log file.

If you want to look for these stack traces in big files, you need more powerful tools. Most of them are command-line tools that can help you navigate to the right place. Before we look at these tools, let's start with the one thing every developer must be able to read: the stack trace in a log.

5.1 Reading the bubbles: stack traces

In the last chapter, we learned that exceptions interrupt the normal flow of a program. They are thrown from one method to another, and when we see them in our log file, we call them stack traces.

Exceptions are like a dolphin releasing an exceptional bubble deep in the ocean. One that rises steadily to the surface.

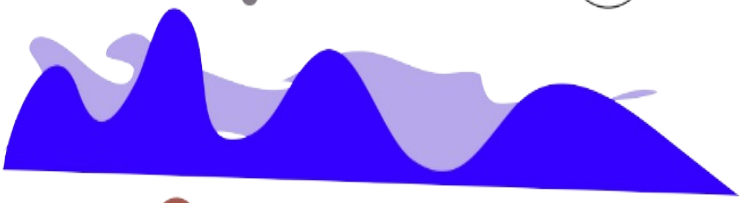
First, they touch seaweeds. The seaweeds don't care, and neither do the

corals. The bubbles keep rising. Scientists are waiting to collect and analyze them. Their composition reveals which way they came from deep below.

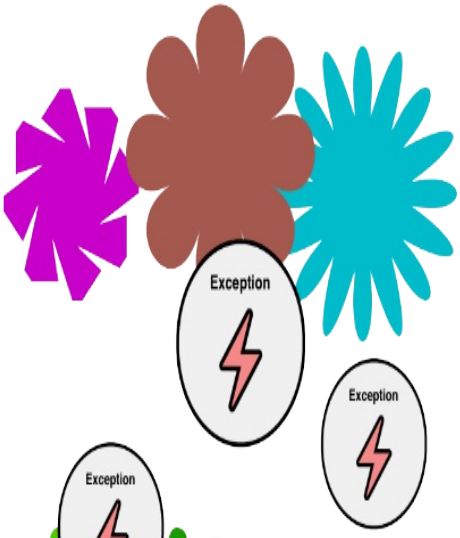
Figure 5.1 Collecting information on their way up

When they are on the surface, in the catch clause, developers can investigate like scientists

4.

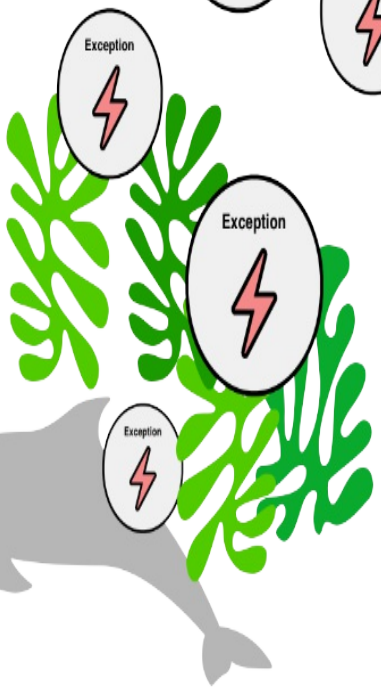


Even additional info from other objects can be added



3.

On their way up, Exceptions can tell where they have been



2.

The dolphin causes Exceptions, with little info

1.

Note

The behavior described in this section applies to single-threaded applications. For a beginner's book, that's perfectly fine. However, if you've been programming Java for a while, you may have heard about multi-threaded applications.

In a multi-threaded application, exceptions don't necessarily bubble up to the main method. Each thread runs independently and may have its own uncaught exception handler, isolated from other threads.

If you are curious about multi-threading, a classic reference is "Java Concurrency in Practice" by Brian Goetz (ISBN: 0321349601). Although it covers older versions of Java, its concepts remain relevant.

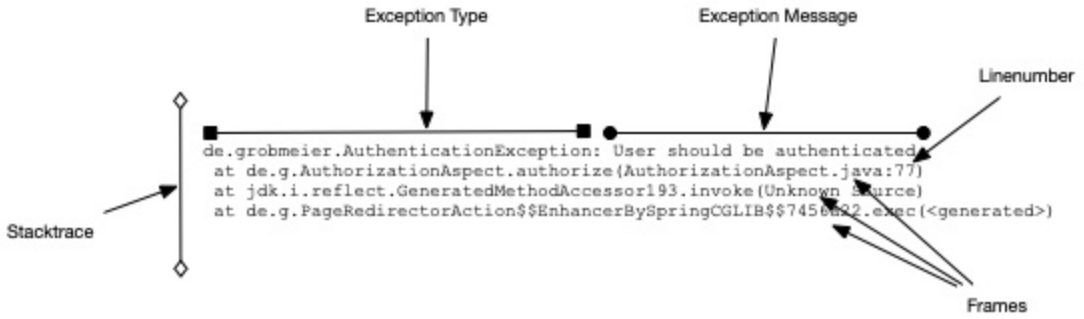
A more modern resource is "The well-grounded Java Developer" (2nd Edition) by Benjamin Evans, Jason Clark and Martijn Verburg (ISBN 9781617298875) https://www.manning.com/books/the-well-grounded-java-developer-second-edition

It explains concurrency, class loading, JVM tuning, and the Java Platform Module System (JPMS).

Exceptions are like bubbles, and developers are the scientists studying them. When we log exceptions, the stack trace reveals their composition—and the path they took on their way up.

A stack trace consists of several parts. The exception type is the class name, such as `NullPointerException` or `IOException`. Each exception usually comes with a message that describes what went wrong, followed by a long list of method calls called stack frames. There are also often numbers at the end of a frame that indicate the line number.

Figure 5.2 Anatomy of a stack trace



5.1.1 A real life stack trace

The actual stack traces you'll see in your log files are usually much longer and can be intimidating at first. Some have dozens or hundreds of lines. With a few simple tricks, you can easily break them down into digestible parts. The following stack trace is a real-world example from the Spring framework—slightly shortened to fit on the page. These kind of stack traces are common, and it is perfectly to feel overwhelmed. There is no way you can identify all the classes immediately. A lot of the information we see in stack traces feel useless to use, and we have to break it down into smaller pieces so it makes sense. Stack traces can be ugly, and the following snippet proves the point.

```
**de.grobmeier.AuthenticationException: User should be authentica
 **at de.g.AuthorizationAspect.authorize(AuthorizationAspect.java
 at jdk.i.reflect.GeneratedMethodAccessor193.invoke(Unknown Sourc
 at jdk.i.reflect.DelegatingMethodAccessorImpl.invoke(
 DelegatingMethodAccessorImpl.java:43)
 at java.l.reflect.Method.invoke(Method.java:566)
 at org.s.aop.aspectj.AbstractAspectJAdvice.invokeAdviceMethodWit
 GivenArgs(AbstractAspectJAdvice.java:634)
 [...] #3
 at org.s.aop.framework.ReflectiveMethodInvocation.proceed(
 ReflectiveMethodInvocation.java:186)
 at org.s.aop.framework.CglibAopProxy$CglibMethodInvocation.proce
 CglibAopProxy.java:753)
 at org.s.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.i
 CglibAopProxy.java:698)
 **at de.g.PageRedirectorAction$$EnhancerBySpringCGLIB$$7456a22.e
 <generated>) #4
 at org.s.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.i
 CglibAopProxy.java:543)
```

Stack traces contain methods that call other methods, and they include the

line numbers where the calls happened. If you've never worked with Spring before, you might notice a lot of unfamiliar class names. That makes them hard to read, even when they are reduced like in my example. Instead of focusing on what you don't recognize, focus on what you do.

5.1.2 Breaking down stack traces

Like with a book, start with the first line. The exception type and its message often point you in the right direction immediately. A `NullPointerException` can be easily identified that way, especially when a line number is shown. Likewise, a message such as "The denominator should not be zero" clearly hints at a division by zero.

If that's not enough, scroll to the bottom and move upward until you find the first class you wrote yourself. In our ocean metaphor, that's the bottom of the sea. The following line is the first I can recognize. It shows where the problem first appeared, the *lowest occurrence* of our own class.

```
at de.g.PageRedirectorAction$$EnhancerBySpringCGLIB$$7456a22.exec
```

The line indicates a problem with page redirection. That alone isn't enough to fix the problem, but it tells us where to start looking.

Next, scroll up to the highest occurrence of a class you wrote. That way, we can see where the problem was raised. In this case, it's in the `authorize()` method of the `AuthorizationAspect` class (line 77).

```
at de.g.AuthorizationAspect.authorize(AuthorizationAspect.java:77
```

Now we can draw our first conclusion: a redirect happened, but the target page required authentication that wasn't provided.

One could say:

- The lowest occurrence is the entry point: where the problem originated or first appeared.
- The highest occurrence is the exit point: where the problem was raised.

You can often skim the frames between these two points. Many are unrelated

calls from the application framework. That’s why learning to scan for your package names is a valuable skill.

5.1.3 Context matters: finding the why

We now know where the failure began, but we also need to understand why. When you find a stack trace, read the surrounding log lines (before and after). Look for the URL, user ID, request ID, parameters, feature flags, or state changes, or anything in general, that explains why this path was taken. The example shows the actual page that was called before the exception occurred.

```
2025-10-10 12:34:56.789 INFO Connected to database /aircraftdb
2025-10-10 12:34:56.889 INFO Redirection to /secure/page
de.grobmeier.AuthenticationException: User should be authenticate
 at de.g.AuthorizationAspect.authorize(AuthorizationAspect.java:7
 at jdk.i.reflect.GeneratedMethodAccessor193.invoke(Unknown Sourc
 ...
```

Later on, we’ll learn about thread names and request IDs, which can help us finding the related lines for the same request. Make it a habit to read more than just the stack trace. Scroll a little before and after to see the big picture of the problem.

5.1.4 Interpreting "Caused by"

Stack traces become more complex when exceptions are wrapped in other exceptions. In the following example, the `SQLException` appears to be the main problem. It’s followed by "caused by". Never ignore these lines. They often reveal the actual root cause that triggered the more visible exception.

```
java.lang.SQLException: Could not select user data for user id 42
 at de.g.RequestProcessor.process(RequestProcessor.java:45)
 at de.g.WebController.handleRequest(WebController.java:88)
Caused by: java.io.IOException: Database not reachable #2
 at de.g.Database.connect(Database.java:22)
 at de.g.RequestProcessor.process(RequestProcessor.java:40)
 ... 2 more
```

There can be many "caused by" sections, which might be overwhelming at first. Fortunately, you can use the same strategy as before. Start by reading

the message, even if it is often too generic to be helpful. Then scroll down to the lowest occurrence of "caused by", read its message, and find the lowest and highest occurrence of our own classes. That isolates the problem at its deepest level.

5.1.5 Suppressed exceptions

Some exceptions are *suppressed*. That can be confusing at first, but the idea is similar to "caused by." Suppressed exceptions usually occur while handling another problem. For example, when closing a file, in a `try-with-resources` block, another exception might be thrown. If an exception already occurred while processing the file, the closing exception is suppressed. In short, Java keeps both exceptions but shows only the first as the main cause. The stack trace looks familiar.

```
java.lang.Exception: Main exception #1
 at de.g.Main.process(Main.java:50)
 at de.g.Main.main(Main.java:20)
Suppressed: java.io.IOException: Closing file failed #2
 at de.g.Main.process(Main.java:45)
 ... 1 more
```

5.1.6 More tips for reading stack traces

A quick note on linenumbers: they’re not always available. If the code was compiled without debug information, line numbers are missing. Also, if the code was obfuscated (for example, by using ProGuard or similar tools to make reverse engineering harder), line numbers are usually removed. In that case, you might see "Unknown Source" or similar messages. When that happens, you’ll need to rely on the order of frames in the stack trace to understand the flow.

A few useful keywords are:

- `… N more`: indicates N frames are omitted, usually because they were duplicates
- `Native Method`: points to code that runs outside the JVM (for example, native libraries or JNI).
- `<generated>`, `$Proxy`, `EnhancerByCGLIB`: runtime-generated classes,

often created by frameworks like Spring or Hibernate. Usually, you can skim those.

With a bit of practice, reading stack traces becomes much easier. In a nutshell, it’s all about these steps:

1. Don’t panic: break it down
2. Read the message
3. If there are any "caused by" sections, start with the last one
4. Find the lowest occurrence of your own class
5. Scroll up to the highest occurrence

If you follow these steps, you should be able to understand most stack traces. Now that we know how to read stack traces, let’s explore some tools that make opening and navigating log files easier.

5.2 Command line: Opening the black box

As already mentioned, log files can grow very large, and editors like Notepad aren’t built to handle them. In most cases, your applications run on remote servers rather than your local machine. Downloading huge log files just to read them is slow, inconvenient, and often unnecessary.

There is a better way: the command line. The command line is also known as "Shell", "Terminal", or "Bash." On Windows, you might hear people refer to it as "CMD." Technically, these are different programs, but they work similarly. In this chapter, we will treat them as the same thing.

If you didn’t use it yourself before, you’ve probably seen a colleague writing commands in a (often) black window. That’s the command line: a tool for running programs without clicking. All major operating systems come with it.

The command line offers powerful tools to read log files. It might not feel as friendly as a graphical interface at first, but with a bit of practice, you’ll see why experienced developers rely on it. It’s a skill worth learning.

For a long time, the command line was mainly used on Linux and macOS. Microsoft eventually recognized this as a weakness. They created a modern command line experience and made it even possible to run Linux applications directly on Windows through the Windows Subsystem for Linux (WSL).

Quick Start WSL

A bit of a background: The first version of WSL was a compatibility layer, translating Linux commands into Windows equivalents. That worked surprisingly well. Starting with WSL 2, Windows includes a full Linux kernel, brining near-native-compaitbility. Although I prefer using Linux directly, WSL has very few drawbacks for most developers.

If you don't have WSL installed yet, follow Microsoft's official installation guide: https://learn.microsoft.com/en-us/windows/wsl/install

All examples in the following sections use Ubuntu Linux or WSL. If you're on macOS, you can use the same—or very similar—commands. Let's get started by trying to navigate to a folder.

5.2.1 Navigating on the command line

When you open your shell, you start in your user folder. That's usually not where your log files are stored, so you'll need to navigate to the correct directory first.

Print your current location

To check your current location, use the `pwd` command. It stands for "**P**rint **W**orking **D**irectory." In the examples, you'll see commands starting with `$>`. It's the command prompt, showing it's your turn to type. Your own prompt might look different, but the commands work the same way.

Typing `pwd` and pressing enter will show your current directory. In my case, it will show that I am in the `/home/christian` folder.

```
$> pwd #1
/home/christian #2
```

Creating folders

Let’s create a subfolder to store and analyze our log files. The `mkdir` command does precisely that. It’s short for "**M**a**K**e **DIR**ectory." Running the following command creates a new folder inside your current directory.

```
$> mkdir aircraftproject #1
```

Listing folder contents

To confirm that it worked, list the contents of your current folder with the `ls` command. It stands for "**L**i**S**t". The output below shows both the newly created folder and an existing folder.

```
$> ls
aircraftproject #1
Music #2
```

Changing directories

To move from your user folder into the new directory, use the `cd` command, short for "**C**hange **D**irectory." It’s simple to use.

```
$> cd aircraftproject
```

After changing directories, you can verify your location again with `pwd`. I often run this command while working in the terminal, just to be sure I’m where I think I am.

```
$> pwd
/home/myfolder/aircraftproject
```

If you want to move "up" one level to your user folder, use the `cd` command again. This time, the target is" `..`". The double dots mean "up one directory." In practice, it looks like this.

```
$> cd .. #1
$> pwd
/home/myfolder
```

The `cd` command works both with relative and absolute paths. On Windows, you could use `cd C:\home\myfolder\aircraftproject`. In WSL, your home directory is usually located under a path like `/mnt/c/Users/christian/`.

Using autocompletion

Another helpful navigation tip is to use the Tab key for autocompletion. For example, start typing a path like `C:/home` and press Tab. The shell will show you all matching folders.

```
$> cd /home/myfolder #1
aircraftproject Music #2
```

If there's only one match, `tab` autocompletes the path for you. If there are multiple options, type a few more characters and press `tab` again. The results will be narrowed down.

With these commands, you're now ready to reach the folder containing your log files. Next, we'll look at how to read them.

5.2.2 Reading logs efficiently

In this section, we'll look at three essential command-line tools for reading log files: `less`, `tail`, and `grep`. Each one of them has special superpowers that make it easier to analyze logs.

Scrolling logs with `less`

`less` is a lightweight tool for viewing files without loading them fully into memory. Even large log files can be opened, and you can scroll through them easily. You can't edit them in `less`, but that's rarely needed anyway. To open a log file with less, type the command followed by the file path.

```
$> less /var/log/apache2/error.log #1
[Sun Apr 09 10:37:20 2023] [notice] workerEnv.init() ok /etc/http
workers2.properties
[Sun Apr 09 10:37:21 2023] [error] mod_jk child workerEnv in erro
[Sun Apr 09 10:49:18 2023] [notice] jk2_init() Found child 5725 i
 scoreboard slot 10
```

```
[Sun Apr 09 10:41:05 2023] [notice] jk2_init() Found child 6327 i
 scoreboard slot 6
[Sun Apr 09 10:41:17 2023] [notice] workerEnv.init() ok /etc/http
workers2.properties
[Sun Apr 09 10:41:18 2023] [notice] workerEnv.init() ok /etc/http
workers2.properties
[Sun Apr 09 10:41:19 2023] [notice] workerEnv.init() ok /etc/http
workers2.properties
```

Use `b` to scroll backward and the `space` bar to move forward. Press `q` to quit and return to the command line.

To go to a specific line, press `:` followed by the line number and Enter. This will bring you directly to that line without scrolling.

Showing line numbers in `less`

Line numbers can be helpful when inspecting log files, and `less` lets you display them. They're disabled by default, since adding them can slow down loading large files. To enable line numbers, use the `-N` option.

```
less -N aircraft.log #1
```

To open a log file at a specific line, add the line number to the command. It might take a moment to jump to the correct position, but it's generally faster than showing line numbers. Starting at line 100, you'd type something like this:

```
less +100 -N aircraft.log #1
```

less to the end of the file

The most important part of a log file is often at the end. When viewing a file with `less`, you don't need to scroll manually—press `G` to jump straight to the bottom (G stands for "go to the end").

If you already know you want to start there, add the `-G` option to your command to open the file directly at the end.

```
less -G aircraft.log #1
```

Searching in a file

You can search within a file while using `less`. Press the `/`, type your search term, and press Enter to start the search. With `n`, you can jump to the next match, and the uppercase `N` will bring you back to the previous match.

Figure 5.3 Using search with less

less sql-wasm.js

```
var initSqlJs = function (moduleConfig) {

  if (initSqlJsPromise){
    return initSqlJsPromise;
  }
  // If we're here, we've never called this function before
  initSqlJsPromise = new Promise(function (resolveModule, reject) {

    // We are modularizing this manually because the current modularize setting in Emscripten has some issues:
    // https://github.com/kripken/emscripten/issues/5820

    // The way to affect the loading of emcc compiled modules is to create a variable called `Module` and add
    // properties to it, like `preRun`, `postRun`, etc
    // We are using that to get notified when the WASM has finished loading.
    // Only then will we return our promise

    // If they passed in a moduleConfig object, use that
    // Otherwise, initialize Module to the empty object
    var Module = typeof moduleConfig !== 'undefined' ? moduleConfig : {};

    // EMCC only allows for a single onAbort function (not an array of functions)
    // So if the user defined their own onAbort function, we remember it and call it
    var originalOnAbortFunction = Module['onAbort'];
    Module['onAbort'] = function (errorThatCausedAbort) {
      reject(new Error(errorThatCausedAbort));
      if (originalOnAbortFunction){
        originalOnAbortFunction(errorThatCausedAbort);
      }
    };
;
```

If you are at the end of a file, you can search backward by pressing `?` instead of `/`.

There are even more options with `less`, and it's certainly a tool worth getting fluent with. `less` was written in 1984. It won't go away any time soon, so it's a valuable investment in your toolbox. It's one of my favorite tools for many years.

5.2.3 Just read the latest with tail

`less` won't refresh once a file is open, but log files might grow in the background. Sometimes you want to watch new lines as they appear.

`tail` lets you read the last lines of a file and even follow new ones in real time. By default, it only shows the final ten lines of a log file.

```
tail /home/myfolder/aircraftproject/logfile.log #1
```

Ten lines are often not enough, so you can display more using the `-n` option. It takes a number as an argument, letting you choose how many lines to show.

```
tail -n 100 /home/myfolder/aircraftproject/logfile.log #1
```

To watch new lines as they're added, use the `-f` option (short for follow). I don't recommend following logs on production systems, unlike you are a high-speed reader. However, the following is very useful during development. Think of it as an auto-refresh for your log files.

```
tail -f /home/myfolder/aircraftproject/logfile.log #1
```

The little sister of `tail` is `head`, which displays the first few lines of a file. You rarely need it for log files, but since both commands, `tail` and `head`, are so similar, it's worth mentioning. Reading and scrolling through logs is easy with `less` and `tail`, but sometimes you need extended search capabilities. That's where `grep` comes into play.

5.2.4 Advanced searching with grep

`grep` is a potent tool that can be difficult to master beyond the basics. It searches for text patterns within files or across entire directories, using regular expressions for advanced matching. In this book, we'll look at its core features, but I encourage you to explore it further on your own.

Basic usage of `grep`

Here's the simplest way to use grep: this command searches for the word "Exception" in a log file and prints all matching lines.

```
grep' Exception' aircraft.log
```

The output is only the lines that contain the word "Exception." stack traces aren't included, since they follow on the next lines. The result might look like this with two different exceptions inside the `aircraft.log`.

```
[Request failed: java.lang.NullPointerException: a is null]
java.lang.IllegalArgumentException: Wrong to use zero as input
```

Showing context with `-A`, `-B`, and `-C`

We've found the exceptions, but this output isn't very helpful yet. The `-A` option (for "after") tells `grep` to include a certain number of lines following each match. For example, `-A 10` shows ten lines after every matched line.

```
grep -A 10 'Exception' aircraft.log #1
```

Now the output is much more helpful, since we can see part of the stack trace. If you need more context, simply increase the number.

```
java.lang.NullPointerException: a is null
 at de.grobmeier.example.aircraft.AircraftController.npe(
AircraftController.java:66)
 at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invok
(Native Method)
 at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invok
NativeMethodAccessorImpl.java:77)
 at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.i
DelegatingMethodAccessorImpl.java:43)
 at java.base/java.lang.reflect.Method.invoke(Method.java:568)
```

```
 at org.springframework.web.method.support.InvocableHandlerMethod
InvocableHandlerMethod.java:207)
 at org.springframework.web.method.support.InvocableHandlerMethod
invokeForRequest(InvocableHandlerMethod.java:152)
 at org.springframework.web.servlet.mvc.method.annotation.
ServletInvocableHandlerMethod.invokeAndHandle(
ServletInvocableHandlerMethod.java:117)
 at org.springframework.web.servlet.mvc.method.annotation.
RequestMappingHandlerAdapter.invokeHandlerMethod(
RequestMappingHandlerAdapter.java:884)
 at org.springframework.web.servlet.mvc.method.annotation.
RequestMappingHandlerAdapter.handleInternal(
RequestMappingHandlerAdapter.java:797)
```

Instead of `-A`, you can use the `-B` option (before), which shows a specific number of lines preceding each match. If you want the full context—both before and after—you can use the `-C` option, which combines the two.

Searching for multiple words

`grep` can also search for multiple words at once using the logical OR operator (|). You'll need to escape it with a backslash (`\|`), otherwise the shell will interpret it as a pipe—a special operator used to pass output between commands.

```
grep 'Exception\|Error' aircraft.log #1
```

Showing line numbers with `-n`

`grep` can also display line numbers. It's fantastic to combine `grep's powerful search with the ability to jump to a specific line in a file in `less`. Just add the `-n` option to show line numbers.

```
grep -n 'Exception' aircraft.log
```

The output shows the line number before each matching line. This option can make `grep` slightly slower, since it needs to count the lines first.

```
63:java.lang.NullPointerException: a is null
```

Now you can open the file with `less` and jump to line 63. Exactly, like we

have seen in the previous section.

```
less +63 aircraft.log
```

In production environments where logs rotate daily (like app.log, app.log.1, etc.), `grep` becomes indispensable. You can even search across all logs recursively with `grep -r 'Exception'/var/logs` or specify to search in multiple files in one folder with `grep 'Exception' *.log`.

`grep` is more potent than we've covered here. You could use it with regular expressions, do invert searches, and much more. The man pages are an excellent place to learn more about it.

Man pages

The internet or AI chatbots can help you with command-line questions, but sometimes it's worth learning directly from the source. Most commands include their own documentation in the form of man pages (short for "manual").

For example, type `man grep` to open the complete documentation for grep. To quit the page, press `Q`; to scroll down, press the space bar.

You can also use `grep --help` for a shorter overview, but for detailed explanations, the man pages are usually more helpful.

Now that we know how to write logs and read logs, it's time to move on to something bigger. Apache Log4j is a popular logging framework that offers many features beyond what we've seen so far.

5.3 Summary

In this chapter, we:

- stack traces are read bottom up, looking for the lowest and highest occurrences of your own classes
- caused by and suppressed exceptions should not be ignored
- Messages containing native methods or generated classes can often be

ignored
- The command line provides tools to navigate to logs efficiently
- Powerful tools to read and search log files are `less`, `tail`, and `grep`

Welcome

Thank you for purchasing *Java Logging with log4j2*.

Many years ago, I was a junior developer. My senior developer introduced the concept of logging to me. A few months later, I heard him explain the same ideas to another junior developer. Eventually, it was my turn to explain. What if everyone explaining logging had a book like this one to share? I imagine my mentor giving me a worn copy. He'd say, "Read it, we'll talk about it tomorrow."

It was also my mentor who sparked my interest in open source. I wanted to contribute code too, and what could be more natural than working on logging itself? I became a member of the Apache Log4j core team in 2009, and I am still there.

When Log4shell, a major security incident, hit the software industry in 2021, I didn't leave the project— quite the opposite. The incident opened my eyes to the importance of open source and logging, and I was reminded how essential it is to truly understand the tools we rely on.

Those challenges inspired this book—a guide that seniors can pass down to juniors. If you're reading this as a senior developer, I hope it prompts you to think: "Yes, that's what juniors need to know!" And if you're a junior developer, I hope you find it engaging and straightforward to read.

Programming is complicated enough. This book's goal is not to add to that complexity but to make logging easy to apply and accessible.

There are numerous options available for *Java Logging with log4j2*, and this book aims to explain the basics first, introduce Apache Log4j, and then compare it with other popular choices. At the end of this book, developers will be able to decide between them and—one day, hopefully—pass this book on to their juniors developers. Until then, enjoy the book.

Please share your thoughts in the liveBook's Discussion Forum. Your

feedback will help shape the final book—and I’m always grateful for questions and for hearing about your experiences.

—Christian Grobmeier

In this book