# Java 8 to 21

*Explore and work with the cutting-edge features of Java 21*

**Shai Almog**

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-55513-922

www.bpbonline.com

# Dedicated to

*Tao & Tara*

# About the Author

**Shai** is an author, entrepreneur, blogger, open-source hacker, speaker, Java rockstar, developer advocate and more. He is a former Sun/Oracle engineer with 30+ years of professional experience. Shai built **Java Virtual Machines (JVMs)**, development tools, mobile phone environments, banking systems, startup/enterprise backends, user interfaces, development frameworks and much more.

Shai is on the advisory board for multiple organizations including dzone, dev network, and so on. Shai speaks at conferences all over the world and shared the stage with luminaries such as James Gosling (father of Java).

# About the Reviewer

**Ravi Rajpurohit** is a highly accomplished Software Engineer, currently holding the position of Manager at Morgan Stanley in Bangalore. With a career spanning various prestigious organizations, Ravi has amassed a wealth of experience and expertise in the field. His journey began as an intern at ISRO RRSC-W in Jodhpur, where he laid the foundation for his career in software engineering. He further solidified his skills while working as a Software Engineer at Confluxsys Private Limited in Pune. Ravi's talent and dedication led him to excel as a Senior Software Engineer at Publicis Sapient in Bangalore, where he contributed to the development of cutting-edge software solutions. Proficient in Python, Java, PHP, and other programming languages, Ravi possesses a versatile skill set that allows him to tackle complex technical challenges. Outside of his professional pursuits, Ravi finds joy in sharing his knowledge through technical teaching during his spare time. Originally from Jodhpur, Ravi brings a unique perspective and strong work ethic to his endeavors, making him a valuable asset in the software engineering realm.

# Preface

Java has a big problem. It is often perceived as old, but Python is older, and JavaScript is its contemporary. Why is that?

Java has had continuous popularity and success for nearly three decades. That means that most of us worked with Java 1.4, which was released two decades ago. Unlike any other platform out there, Java is still compatible with that release. That is fantastic but also creates a sense of disconnect. Developers compare that highly outdated version of Java to modern incarnation of other languages or platforms. That is an unfair comparison, and Java is a victim of its own success.

There are several sources for information about new Java features but there is a lack in a comprehensive introductory guide to modern Java, that carries us from that old version to the modern world. With this book, we hope to fill that gap, to answer the question of "is this still the right way to solve this problem?".

This book is divided into 10 chapters. They cover Java basics, OOP basics and continue to cover everything we need to know, in order to build a modern Java backend. Here is a brief description for each chapter:

**Chapter 1: Hello Java –** In this chapter, we will review the basics of the language and tooling, to make sure we are all on the same page. We focus on Java 8 syntax and as we move forward in the book, we will discuss the newer features of Java, all the way to version 21.

**Chapter 2: OOP Patterns –** In this chapter, we explore the best ways to leverage OOP, what we would consider as good design and what type of OOP principles are expressed in the Java API.

**Chapter 3: 8 to 21 to GraalVM –** Java went through many pivotal releases. Java 1.1 introduced inner classes. Java 2 introduced Swing and collections. Java 5 introduced Generics. However, one of the most pivotal releases was Java 8, which introduced a slew of wide sweeping changes. Up until Java 8, the releases of Java versions were irregular although they tended to follow a two-year cadence. With the release of Java 9, this changed. We now have a 6-month release cycle. This pace made the period between Java 8 and Java 21 relatively short. We review the changes between these releases in this chapter.

**Chapter 4: Modern Threading –** In this chapter, we will discuss the evolution of Java threading from the beginning, all the way to the future. We will explain the trade-offs and challenges and we will also discuss the best ways to write highly concurrent code moving forward.

**Chapter 5: It's Springtime in Java –** Spring brings together libraries and tools from multiple different projects, to create a unified experience. Spring is also highly configurable; it is not limited to backend or database driven applications.

**Chapter 6: Testing and CI –** In this chapter, we will cover some common tools in the world of testing such as JUnit and Mockito. We will cover GitHub Actions when discussing CI due to its wide availability and free quota. We will also discuss some high-level concepts such as **Test-Driven Development (TDD)**.

**Chapter 7: Docker, Kubernetes and Spring Native –** The original thought behind this chapter was around cloud native development. However, that acronym is somewhat vague. In the old days, we would take the resulting jar files we would build and place them on an application server connected to the internet. These days are long gone, containers and orchestration have completely changed the way applications are deployed in production.

**Chapter 8: Microservices –** The rise of Kubernetes has made the Microservices approach far more commonplace. The cost of spinning up and managing multiple smaller servers became manageable and as a result, drove a massive move in that direction. Microservices bring a great deal to the table, they are easier to build and often easier to scale.

**Chapter 9: Serverless –** The complexities of Monoliths and Microservices pushed some developers towards an easier route. Serverless is a problematic branding term indicating that we no longer need to manage our server, only the application. The serverless provider takes over the complexity of spinning up container instances for us as long as we follow some guidelines in application development.

**Chapter 10: Monitoring and Observability –** Monitoring and observability are essential aspects of software development that help developers and operations teams ensure the reliability, stability, and performance of their applications. As

systems become increasingly complex, the need for effective monitoring and observability tools grows in tandem, providing a comprehensive understanding of the behavior and health of applications during development, deployment, and production. The Java ecosystem is no exception, offering a rich set of libraries and tools designed to facilitate these critical tasks. This chapter will delve into the concepts of monitoring and observability in the context of Java applications, exploring the underlying principles, methodologies, and best practices that will enable developers to build robust, scalable, and resilient systems.

# Code Bundle and Coloured Images

Please follow the link to download the
***Code Bundle*** and the ***Coloured Images*** of the book:

## https://rebrand.ly/boau4e5

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Java-8-to-21**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

# Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

## Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

## If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

# Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**

# Table of Contents

# Chapter 1
# Hello Java

## Introduction

Java is one of the most ubiquitous programming languages on earth. One of its biggest selling points is its simplicity. In fact, the father of Java, *James Gosling*, used to describe his approach to the design of Java with the following metaphor.

When James had to move to a new residence, he would pack all his belongings into well-labeled boxes. Then, he would only unpack the stuff he needed. Six months later, he would throw away all the other boxes without looking at the content. If he looked at the content, he would probably keep it. Java was designed according to the principle of minimalism. Over the years, it grew, but it is still a small language with a massive API and ecosystem.

In this chapter, we will review the basics of the language and tooling to make sure we are all on the same page. We will focus on Java 8 syntax, and as we move forward in the book, we will discuss the newer features of Java, all the way to version 20.

# Structure

In this chapter, we will discuss the following topics:

- Requirements
- Setting up a project
- Hello Java
- Principals of OOP
  - Encapsulation
  - Inheritance
  - Polymorphism
- Built-in types
  - Arrays
  - Generics and Erasure
- Debugging
  - Debugging Java

# Objectives

By the end of this chapter, the reader will learn the basics of working with Java. This book assumes some basic prior familiarity with Java. Regardless we chose to approach this chapter as a blank slate since your familiarity might be significantly out of date. By the end of this chapter, you will be able to read simple Java code; as the book progresses, we will build on top of that knowledge. As such, this chapter is the foundation for the rest of the book.

# Requirements

To get started, we need to install the Java JDK, and in our case, OpenJDK is recommended. OpenJDK is the open-source version of the Java virtual machine, which we need in order to run Java applications. There are commercial packages of the JDK as well as the Oracle official bundle. However, OpenJDK is free, portable, and supported indefinitely, so a variant of OpenJDK is highly recommended. There is a comprehensive tool to locate the OpenJDK distribution that fits your requirements on the foojay.io site[1]. You can check out SDKMAN[2] if using a Mac, Linux, or the Linux subsystem on Windows.

1 https://foojay.io/almanac/jdk-20/
2 https://sdkman.io/

Other than that, you will need to install IntelliJ/IDEA[3], which is the leading Java IDE. The community edition is sufficient for most novices, but the Ultimate edition is recommended, especially for working with Spring and Web.

# Setting up a project

Java is simple. This facilitated an enormous ecosystem around it and, as a result, added complexity to Java as a whole. We can compile a single Java source file using the `javac` command line. But this is not practical for significant applications. A real-world application is comprised of multiple files, with interaction among the various pieces, and is often packaged in a novel way. There are many complexities in the build process, including the management of dependencies. That is where build tools come in. At the time of this writing, there are three common build tools for Java as follows:

- **Maven**: This is the tool we will discuss in this book. It is the market leader, although a bit older. It uses XML syntax by default and is somewhat clunky. However, it has a huge ecosystem and is very mature.
- **Gradle**: Uses a syntax based on Groovy or Kotlin for the build scripts. Works with Mavens dependency system, making the migration from Maven easy.
- **Ant**: A legacy system based on XML and a precursor to Maven. It is listed for it its completeness, as we still run into ant scripts occasionally.

All of these tools can be used effectively from the command line, but we do not need to do that. Java is particularly powerful when working within the confines of the IDE, where we can leverage its automation to instantly spot problems and investigate our code. To get started, we launch IntelliJ/IDEA and create a new project[4], as seen in *figures 1.1* and *1.2*:

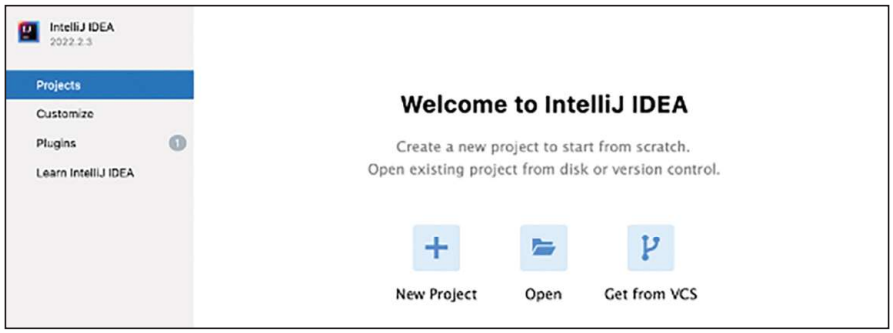


***Figure 1.1**: IntelliJ/IDEA new project window*

3 https://www.jetbrains.com/idea/download/
4 The complete projects are available here: https://github.com/shai-almog/java-book

Refer to *figure 1.2* to see the steps for the creation of a new project:

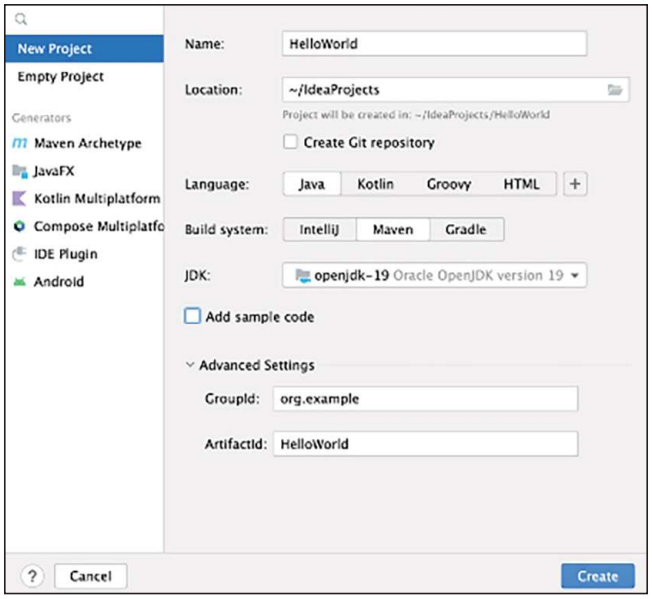


***Figure 1.2**: IntelliJ/IDEA creating a new project*

When creating a new project, make sure to select **`Maven`** as the build system and a new version of the JDK. Notice that you can download a version of the JDK directly from the combo box to pick a JDK, as shown in *figure 1.3*.

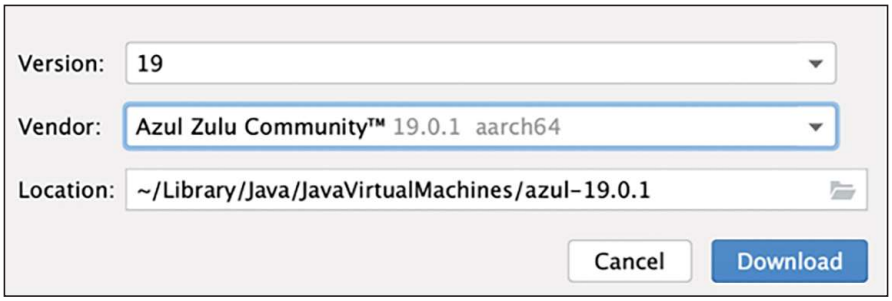


***Figure 1.3**: Download and install a new JDK from within IntelliJ/IDEA*

Once we create a new project, we are faced with the UI shown in *figure 1.4*. We have expanded the tree on the left-hand side to show the Java directory. This is where our Java source code should be placed. On the right side, we can see the maven pom file, which is not important at this point. This gives us the environment where we can start writing code.

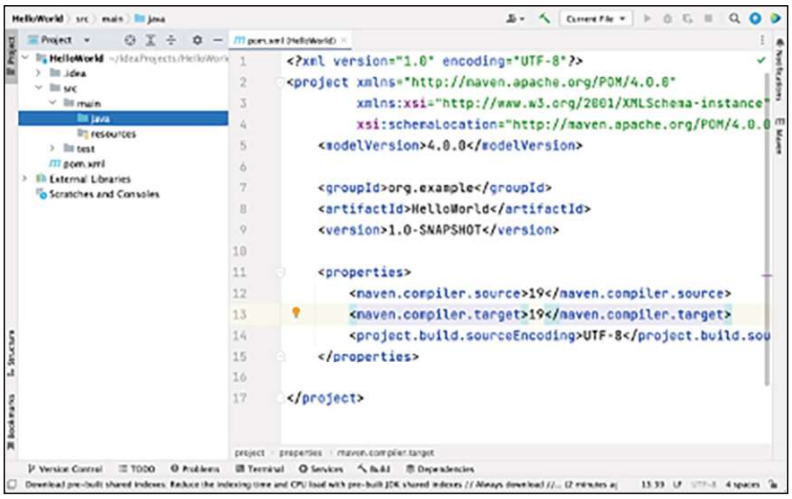


***Figure 1.4**: New Maven project in IntelliJ/IDEA*

# Hello Java

Let us start with the most basic "hello world" example we can make. Right-click the Java directory in the project, as shown in *figure 1.5,* and select **`New | Java Class`**. When prompted, we can enter the name of the new class and press enter. We went with **`HelloWorld`**. Names have a capital first letter by convention (but it is not required). They followed the convention of capitalizing every word. Names cannot include spaces and various other characters. They cannot start with a number either but can be Unicode characters from arbitrary languages (although this is uncommon). The name cannot be one of the reserved Java keywords[5].

*Figure 1.5* features the New Java Class Context Action:

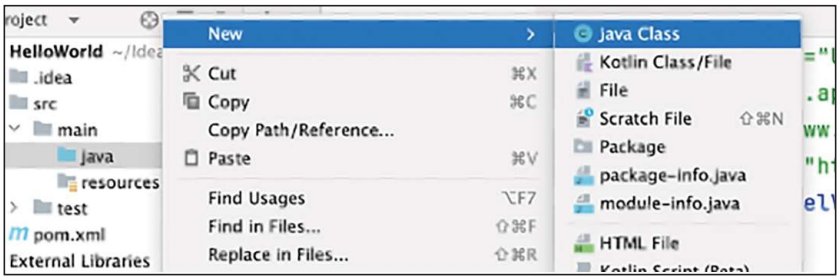


***Figure 1.5**: New Java Class Context Action*

5 https://en.wikipedia.org/wiki/List_of_Java_keywords

*Figure 1.6* features the New Java Class Context Action:


***Figure 1.6**: New Java Class Prompt dialog*

This creates a file named **`HelloWorld.java`** in Java. In order for a class to be public, it must reside in a source file with the same name. Notice that since the language is case-sensitive, the public class **`HelloWorld`** cannot reside in the file **`helloworld.java`**. A public class is exposed to usage outside of its package. We will discuss packages soon enough. Now that we understand that let us create our first Java application.

```
1. public class HelloWorld {
2.     public static void main(String[] argv) {
3.         System.out.println("Hello World");
4.     }
5. }
```

Let us go over the lines in the code one by one. In Line 1, we start with the **`public`** keyword. This indicates that we wish to export this class to external packages. In this case, since we want to run the class, it must be public. The next keyword is **`class`**, which is a basic building block of objects in Java. In Java, almost everything is part of a class in one way or another. Classes let us package code and data (methods and fields) together and work as a single cohesive element. This is a big subject that we will discuss in the following section.

In Line 2, we start with the public again. Elements within the class can have different visibility levels. Within the class, you have full access to everything, and visibility applies to other classes. If we remove the public keyword from the class, it can still be used by other classes in the same package but cannot be used out of the package. The same is true for the elements we write in the class. In this case, it is a method that is an operation we can perform. This method can be accessed by everyone because it is public. Methods (and fields) can have the following visibility levels:

- **`public`**: Full access by anyone with access to the class. Notice that if the class is not public and the method is public, it would still not be visible to everyone. Only those who can access the class.

- **`protected`**: This is like the default access but also allows access to subclasses, even if they are outside of the current package. We will discuss subclassing soon.
- **`[default]`**: Unlike the others, this is not a keyword. This is the default mode when we do not specify visibility and just leave out one of the other keywords. The default visibility is package private. That means only classes within the package have access to the method or field.
- **`private`**: This is the strictest visibility level. Elements marked as private are only visible from within the class.

The next keyword is **`void`**, which means that the method does not return a value. After that, we have the name of the method, which is "main" and the arguments passed to this method. The arguments are an array of strings named **`argv`**.

The body of the method references the **`System`** class, which has a field named **`out`**. We invoke the **`println`** public method on that object and pass the string **`Hello World`** as the argument. This prints **`Hello World`**, as can be seen in *figure 1.7*.

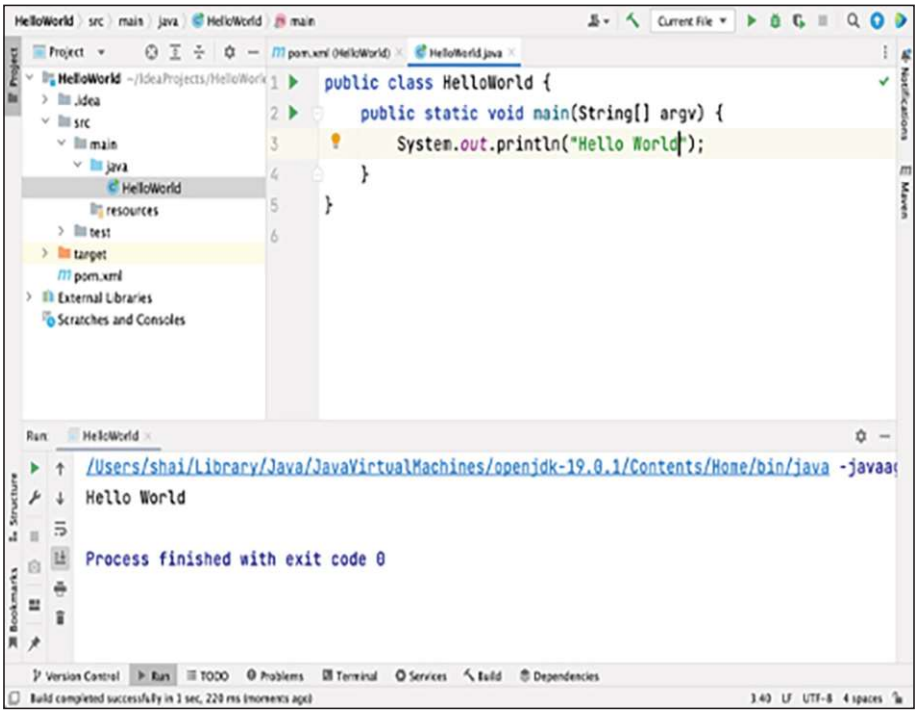


***Figure 1.7**: After pressing the green play button on the top, the application runs; we can see the output in the Console section at the bottom of the screenshot*

# Principals of OOP

Let us pause for a second and take a step back. Why is it important to place methods in classes? Why are we making such a big deal about visibility attributes such as the public? This all fits into the three principles of **Object Oriented Programming** (**OOP**).

**Encapsulation** is the first and arguably most important principle of the three. It means that the data of the object and the operations on that data are packaged together in one class. But it has another important aspect: hiding. When we encapsulate data, we hide implementation details. We will start with a simple example.

# Encapsulation

Let us consider, for example, that your daughter is learning simple fractions at school. You want to help her practice that. You want to create an application that will help her practice simple fractions. In a simple fraction, we have two numbers: a numerator and a denominator. The numerator is the number on top of the fraction, and the denominator is the number on the bottom[6]. We can represent a fraction like the following:

```
1. public class Fraction {
2.     public int numerator;
3.     public int denominator;
4. }
```

This code does not include any encapsulation whatsoever, and we left both fields public. Let us see how we can use it to implement a simplistic math equation. Notice that we are taking a very simple approach here because this is a demo. We will package the logic into a method so that we can add two fractions easily:

```
1. public static Fraction addFractions(Fraction first, Fraction second)
   {
2.     var newFraction = new Fraction();
3.     newFraction.numerator =
4.             first.numerator * second.denominator +
5.             second.numerator * first.denominator;
6.     newFraction.denominator =
```

6 This is for non-native English speakers; I did not learn the English terms in school and had to look them up

```
7.              first.denominator * second.denominator;
8.      return newFraction;
9.  }
```

The code is relatively simple, albeit a bit verbose. We access the fields and multiply them, add them, and assign the result to a newly created object. We can now make simple usage of this API:

```
1.  var first = new Fraction();
2.  first.numerator = 1;
3.  first.denominator = 2;
4.
5.  var second = new Fraction();
6.  second.numerator = 2;
7.  second.denominator = 3;
8.
9.  var result = addFractions(first, second);
10. System.out.println(
11.         first.numerator + "/" + first.denominator + " + " +
12.         second.numerator + "/" + second.denominator + " = " +
13.         result.numerator + "/" + result.denominator);
```

The code is very simple; we create two objects. Assign the values representing 1/2 and 2/3, respectively. We then invoke the **`addFractions`** method that we defined before. Finally, we print the full equation. This is a bit verbose, but ultimately works. It can be made more efficient with additional methods, but it has some failings that cannot be fixed. Let us continue with the code.

```
1.  var third = new Fraction();
2.  third.numerator = 1;
3.  third.denominator = 2;
4.
5.  var forth = new Fraction();
6.  forth.numerator = 2;
7.
8.  // bug forgot to change that to forth...
9.  second.denominator = 3;
```

```
10.
11. var secondResult = addFractions(third, forth);
12. System.out.println(
13.         third.numerator + "/" + third.denominator + " + " +
14.         forth.numerator + "/" + forth.denominator + " = " +
15.         secondResult.numerator + "/" + secondResult.denominator);
```

The fact that this code is duplicated and verbose is a problem. But the bigger problem is the 9th line. It assigns the value to the wrong variable, resulting in a division by zero. If we run the code, we see the following:

```
1/2 + 2/3 = 7/6
1/2 + 2/0 = 4/0
```

Notice that the second line is wrong because the code was not meant to deal with division by zero. Since there is no encapsulation, we could not catch the illegal value of the field before usage. Let us take a second stab at this with encapsulation:

```
1.  public class Fraction {
2.      private final int numerator;
3.      private final int denominator;
4.
5.      public Fraction(int numerator, int denominator) {
6.          this.numerator = numerator;
7.          this.denominator = denominator;
8.          if(denominator <= 0) {
9.              throw new IllegalArgumentException("Invalid denominator:
                " + denominator);
10.         }
11.     }
12.
13.     public Fraction add(Fraction other) {
14.         int numerator = this.numerator * other.denominator +
15.                 other.numerator * denominator;
16.         int denominator = this.denominator * other.denominator;
17.         return new Fraction(numerator, denominator);
```

```
18.     }
19.
20.     @Override
21.     public String toString() {
22.         return numerator + "/" + denominator;
23.     }
24. }
```

Let us review specific lines of code to understand what is different about this version of the class. In Lines 2 and 3, we define the same variables. They are private, which means that they are fully encapsulated and can only be accessed from within the same class. They are both final. That means they cannot be modified after assignment; they must be assigned in the constructor at the latest. This effectively makes this class immutable; its content cannot be modified. Immutability is an important design principle as it promotes safer, more reliable code.

In Line 5, we define a constructor for the class that initializes both variables. Notice that we use the same name for the constructor arguments as the fields. This is completely optional but is a very common convention in Java. To distinguish between the arguments and the class fields, we prefix the fields with the keyword **`this`**. In Line 8, we explicitly throw an exception if the denominator is illegal. This prevents users from creating invalid objects intentionally or accidentally.

The add method on Line 13 includes many encapsulation benefits. It is no longer static and can be named **`add`** instead of **`addFractions`** since it is now directly associated with a fraction. It no longer needs a second argument since it uses the fields of this class.

Line 21 overrides the **`toString`** method of Java **`Object`**. This brings us to an inheritance which is the second principle of OOP. All objects in Java inherit from a class called **`Object`**, which defines a few important methods, including **`toString`**. This means that when we try to print the object, it will appear correctly. Notice that Line 20 includes the **`@Override`** annotation.

Annotations let us declare things about elements in the Java code; in this case, we indicate that we are replacing a method that is defined in the base class (**`Object`**), but we do not need to do that. It will work fine without the override annotation. The reason it is recommended to add that annotation is that if the method in the base class is removed or missing, we will get a compiler error. We will discuss inheritance in more detail soon enough. Let us look at the usage of this new class:

```
1. var first = new Fraction(1, 2);
2. var second = new Fraction(2, 3);
3. var result = first.add(second);
4. System.out.println(first + " + " + second+ " = " + result);
5.
6. var third = new Fraction(1, 2);
7.
8. // will throw an exception...
9. var forth = new Fraction(2, 0);
10. var secondResult = third.add(forth);
11. System.out.println(third + " + " + forth+ " = " + secondResult);
```

This is the full usage, including the "buggy" second block. Notice how much more concise it is. Line 3 is particularly satisfying in its simplicity. Notice that Lines 4 and 11 become trivial compared to the previous code. Since **`toString()`** is built into Java, the code is the equivalent of writing:

```
1. first.toString() + " + " + second.toString() + " = " + result.
   toString()
```

When we run this version, the bug in Line 9 becomes even more obvious as we get a clear exception:

```
1/2 + 2/3 = 7/6

Exception in thread "main" java.lang.IllegalArgumentException: Invalid
denominator: 0

    at com.debugagent.ch01.encapsulation.Fraction.<init>(Fraction.
    java:11)

    at com.debugagent.ch01.encapsulation.SampleUsage.main(SampleUsage.
    java:15)
```

Notice that the exception points us to the file where the error occurred, that is, the class name and the line number. This makes it very easy to locate the code that triggered the problem and make a fix.

Java 14 introduced a new concept: **Records**. A Java record is a final class that has final fields. It is immutable. This seems like the ideal option for our fractions. Let us port our code to use records:

```
1. public record Fraction(int numerator, int denominator) {
2.     public Fraction add(Fraction other) {
3.         int numerator = this.numerator * other.denominator +
4.                 other.numerator * this.denominator;
5.         int denominator = this.denominator * other.denominator;
6.         return new Fraction(numerator, denominator);
7.     }
8. }
```

This is the record equivalent of our fraction class or at least a close approximation. The usage code is identical if we use a record. However, there are two things missing here. The **`toString()`** method and the verification code. If we run this, you will see the following output:

```
Fraction[numerator=1, denominator=2] + Fraction[numerator=2,
denominator=3] = Fraction[numerator=7, denominator=6]

Fraction[numerator=1, denominator=2] + Fraction[numerator=2,
denominator=0] = Fraction[numerator=4, denominator=0]
```

This is due to the default implementation of **`toString()`** in records and the fact that we did not explicitly create a constructor. We can solve both problems by creating a more verbose record:

```
1. public record Fraction(int numerator, int denominator) {
2.     public Fraction(int numerator, int denominator) {
3.         this.numerator = numerator;
4.         this.denominator = denominator;
5.         if(denominator <= 0) {
6.             throw new IllegalArgumentException("Invalid denominator:
               " + denominator);
7.         }
8.     }
9.
10.    public Fraction add(Fraction other) {
11.        int numerator = this.numerator * other.denominator +
12.                other.numerator * this.denominator;
13.        int denominator = this.denominator * other.denominator;
```

```
14.         return new Fraction(numerator, denominator);
15.     }
16.
17.     public String toString() {
18.         return numerator + "/" + denominator;
19.     }
20. }
```

It is still slightly smaller than the class and replaces the default implementations of the constructor and **`toString()`**. It is still worth it since it implements other methods, specifically **`equals()`** and **`hashcode()`**.

One final subject we should cover is packages. In the sample code for this chapter, you will find all the samples shown to you. They are all in a single project file, and all have the same names. This might seem odd. How can the **`Fraction`** class avoid collision with the **`Fraction`** record?

The answer is that they reside in different packages. The best practice in Java is to place all classes within packages representing their roles. The name of the package uses a reverse domain notation, followed by the name of the package. In a similar way to classes residing in files bearing the same name, we expect packages to reside in directories matching the package name. For example, in the following package:

```
1. package com.debugagent.ch01.records;
```

The IDE created a directory hierarchy matching **`com/debugagent/ch01/records`** under the Java directory. Notice the name of the package. The author of the book owns the domain debugagent.com. By using the name that one owns in reverse, we make sure it will not collide with code that another developer might write. The following parts of the package name are up to you to decide. There is another abstraction of modules that we will discuss later.

# Inheritance

The second principle of OOP is inheritance. We discussed it briefly in the encapsulation section but let us take a step back and discuss the basics both in OOP and in Java. Inheritance lets us base a new class on an existing one, where we can expose common functionality. Java includes the following two types of inheritance:

1. Implementation
2. Interface