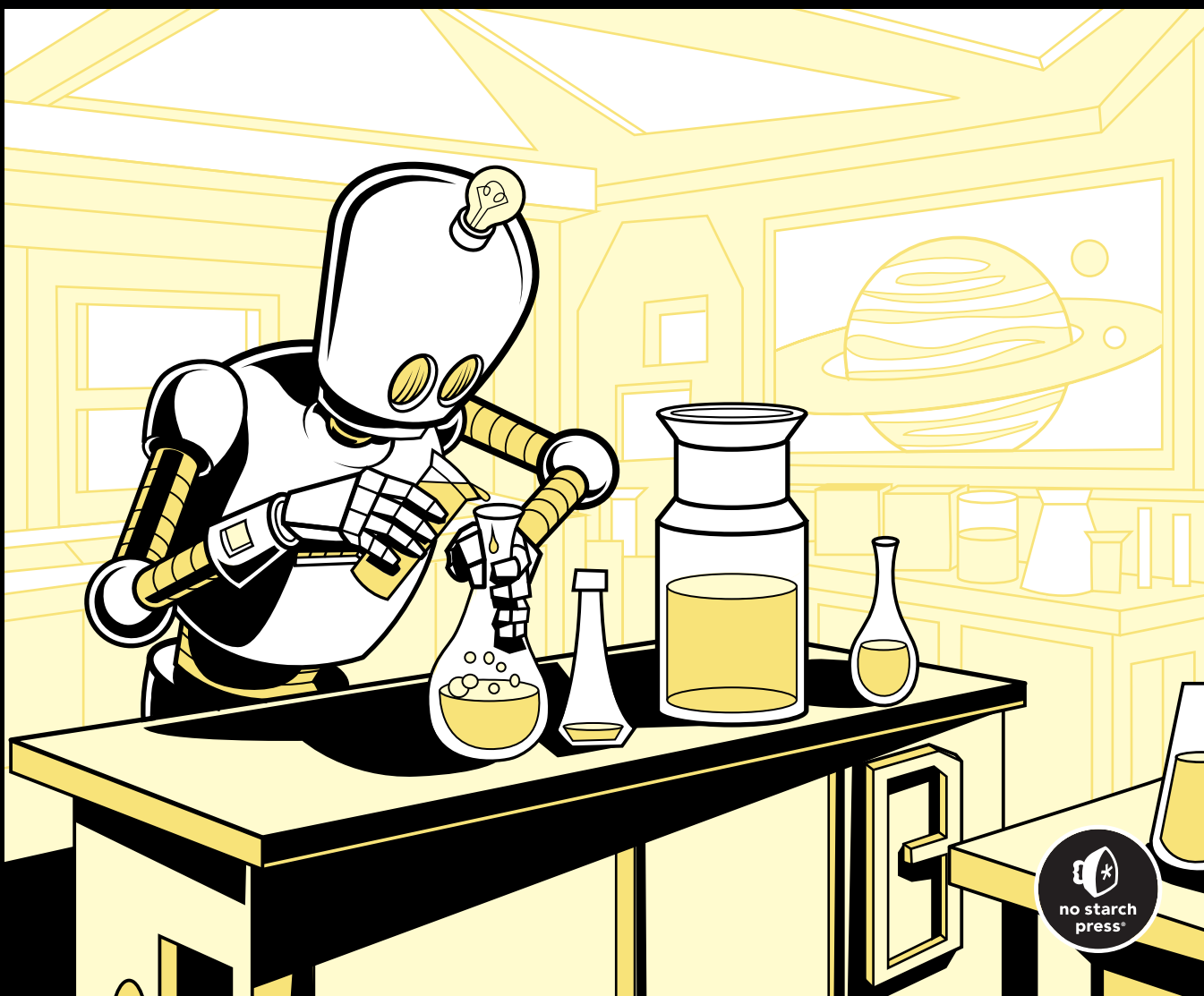
KOTLIN FROM SCRATCH

A PROJECT-BASED INTRODUCTION FOR THE INTREPID PROGRAMMER

FAISAL ISLAM



no starch press®

KOTLIN FROM SCRATCH

KOTLIN FROM SCRATCH

A Project-Based Introduction for the Intrepid Programmer

by Faisal Islam

no starch press®

San Francisco

First printing

ISBN-13: 978-1-7185-0352-6 (print)
ISBN-13: 978-1-7185-0353-3 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Jennifer Kepler
Developmental Editors: Nathan Heidelberger and Abigail Schott-Rosenfield
Cover Illustrator: Josh Kemble
Interior Design: Octopod Studios
Technical Reviewer: Anton Arhipov
Copyeditor: James Brook
Proofreader: Audrey Doyle
Indexer: BIM Creatives, LLC

Figure 9-3, created by Aiyaan Faisal, has been reproduced with permission.

Library of Congress Cataloging-in-Publication Data

```
Names: Islam, Faisal (Author of books on computer programming), author.
Title: Kotlin from scratch : A project-based introduction for the intrepid programmer /
  by Faisal Islam.
Description: San Francisco : No Starch Press, [2025] | Includes index.
Identifiers: LCCN 2024023794 (print) | LCCN 2024023795 (ebook) |
  ISBN 9781718503526 (paperback) | ISBN 9781718503533 (ebook)
Subjects: LCSH: Kotlin (Computer program language) | Functional programming
  (Computer science)
Classification: LCC QA76.73.K68 I85 2025  (print) | LCC QA76.73.K68
  (ebook) | DDC 005.13/3--dc23/eng20240805
LC record available at https://lccn.loc.gov/2024023794
LC ebook record available at https://lccn.loc.gov/2024023795
```

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

[E]

To Saila and Aiyaan, for their unwavering patience and steadfast support,
and to all the curious minds

About the Author

Faisal Islam is a manager, educator, developer, and author of both fiction and nonfiction. With over two decades of coding experience in languages such as C, Java, Python, and Kotlin, he thrives on solving complex real-world challenges. Faisal's background in engineering and economics equips him with a unique perspective, allowing him to apply computational thinking, modeling, simulation, and optimization techniques effectively.

Beyond his technical pursuits, Faisal is an advocate for STEM education, particularly among young learners. His passion lies in inspiring the next generation of coders. In his spare time, Faisal enjoys photography, sci-fi novels, and traveling with his family.

About the Technical Reviewer

Anton Arhipov is a developer advocate for the Kotlin team at JetBrains. With a professional background in server-side development, Anton has been building tools for developers for over a decade. Recognized as a Java Champion since 2014, he often presents as a speaker at large software development conferences.

BRIEF CONTENTS

CONTENTS IN DETAIL

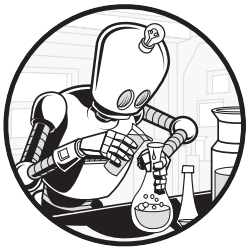
ACKNOWLEDGMENTS

My heartfelt gratitude goes out to the remarkable team at No Starch, including Bill Pollock, Jill Franklin, Nathan Heidelberger, and Abigail Schott-Rosenfield. Without their support, I could not have transformed my fledgling idea into the very book you now hold. Nathan's and Abigail's meticulous editing and diligence have made this book eminently readable. No Starch's commitment to excellence in publishing was evident throughout. During the production stage, I had the pleasure of working with Sabrina Plomitallo-González, Jennifer Kepler, and James Brook—they ensured that the book met the highest publishing standards.

I extend my sincere appreciation to my technical reviewer, Anton Arhipov. His thorough technical review and insightful suggestions helped address coding issues I might have overlooked. His guidance allowed me to embrace idiomatic Kotlin styles and best practices.

Finally, I am profoundly thankful to my wife, Saila, and my son, Aiyaan. Their unconditional support and encouragement during the book-writing process were invaluable. Aiyaan, as an early reader of some of the book chapters, contributed a beautiful illustration for the section on ant colony optimization. I hope that witnessing the final product will motivate him to explore the entire book—he possesses a genuinely curious mind.

INTRODUCTION



This book will teach you to code from scratch using Kotlin, one of the most exciting programming languages used today. Along the way, you'll learn to think like a problem solver and programmer. Through 37 hands-on projects, you'll develop applications that can tackle a wide range of fun and challenging problems, from basic math and science to advanced applications in algorithms and optimization.

You'll learn the most from this book if you have a curious mind. You embrace thinking systematically and are not hesitant to ask why things behave the way they do, challenge commonly held assumptions, and explore unique perspectives. This book will help you use those skills to uncover the inherent complexities of real-world problems and build simplified mathematical models to explore what-if scenarios. Kotlin, a simple, elegant, and powerful general-purpose language, is an excellent tool for these purposes.

Prepare for a journey where coding isn't just a skill but a way to unlock your imagination and problem-solving capabilities. With code, you can tackle problems that would be intractable without a computer, conduct in-depth analysis, and generate fascinating and useful insights.

The Power of Coding

Coding is the process of creating instructions that computers can understand and execute. It has revolutionized the way we live and work. Coding powers almost every modern device, from software, websites, mobile apps, smartwatches, and smartphones to the Mars rover exploring a distant world 140 million miles away.

Here are some reasons you may want to learn to code:

Build valuable skills In today's digital age, coding is a highly sought-after ability that opens up numerous career opportunities. It also helps you improve your critical-thinking and problem-solving abilities.

Automate tasks You can code programs that perform repetitive tasks faster and more accurately than humans. For example, many manufacturing processes are now done by robots that follow coded instructions.

Quickly analyze large amounts of data Coding skills are essential for analysts working with increasingly large quantities of data. You can use coding to extract insights from large datasets and to help make data-driven decisions.

Build your own projects Programming your own software, websites, or mobile apps can be a fun and rewarding way to bring your ideas to life, express your creativity, and earn some extra money.

Have fun Coding can be a fun and rewarding hobby that allows you to create something from scratch and see it come to life.

Coding is for everyone, regardless of age, background, or interest. All you need is a computer, an internet connection, and a willingness to learn. Why not give it a try?

Why Kotlin?

Kotlin is a modern, general-purpose programming language first officially released in 2016 by JetBrains, its parent company. Kotlin was named after Kotlin Island, near Saint Petersburg in the Baltic Sea. This follows the tradition of Kotlin's parent language, Java, which, according to some sources, was named after an island in Indonesia.

Kotlin runs on the same platform as Java, the Java Virtual Machine, and is fully compatible with Java. However, Kotlin is much more concise and expressive, with its own unique features. As a relatively new programming language, it has had the benefit of 20/20 hindsight, meaning it incorporates some of the best features from other popular languages that predate it. It provides many ways to reuse code between multiple platforms,

and its clean language design and powerful features offer a pleasant developer experience.

Here are some of the key advantages of Kotlin:

Easy to learn Because Kotlin's syntax is concise and easy to read, you can learn it more quickly than many other programming languages.

Versatile Kotlin is widely used for mobile applications (especially Android apps), but you can also use it to develop full-stack web and cross-platform applications that run on Windows, Mac, or Linux devices.

Safe Kotlin's type system helps you avoid null pointer exceptions and common errors that are difficult to handle in other programming languages such as Python, C, and C++.

Supports multiple programming styles You can choose your preferred way to code in Kotlin, whether procedural, structured, object oriented, or functional. You can also combine different styles as needed. (Projects in this book use elements of all these styles.)

Interoperable with Java You can use Kotlin alongside Java, meaning you can use Java code and libraries in Kotlin programs, which makes it easier to migrate from Java to Kotlin. Many enterprise-level applications were written in Java and are now being migrated to Kotlin for easy maintenance. As an additional benefit, Java libraries are more mature, and Kotlin can use them while new Kotlin-specific libraries are being developed.

Popular Kotlin is one of the fastest-growing programming languages and has become very popular since Google recognized it as the preferred language for Android app development. Kotlin developers are in high demand in the software industry.

In short, Kotlin is a versatile programming language that can be used for both commercial and noncommercial applications. It's beginner-friendly and suitable for first-time learners who want to explore its features and build their programming skills while solving real-world problems.

Who Is This Book For?

This book is for anyone who wants to learn how to use a fun, modern programming language to tackle real-world problems that cannot be solved manually. It does not assume any prior coding knowledge, though familiarity with another programming language will help you move through the covered topics more quickly.

This book is not a traditional textbook on coding, nor is it a comprehensive reference manual on Kotlin. Instead, it teaches you to think systematically and strategically, helping you develop problem-solving strategies before diving into code development.

This book takes a hands-on approach. Throughout the book, you'll learn by working through numerous examples, full-length projects, and

exercises. You'll use fundamental principles from math and science to construct simplified mathematical descriptions of the underlying problems (we call them models), and then develop Kotlin applications to solve those problems efficiently and quickly. In addition, the book's projects demonstrate core concepts from numerical analysis and computer science, such as convergence and stability, time and space complexity, recursion, sorting, searching, and several nature-inspired stochastic optimization techniques.

This book does not cover advanced Kotlin topics such as generics, extensions, type aliases, or coroutines, which are more suitable for complex applications. It also does not cover computer science concepts such as advanced data structures and design patterns, or how to develop Android apps with Kotlin. It's impossible to cover all that ground in a single book. Instead, this book will help you master essential Kotlin features and skills that will prepare you for further learning and exploration.

What's in This Book?

This book is organized into four parts and nine chapters. Each chapter includes core concepts, examples, one or more full-length projects, and exercises.

Part I covers essential coding skills in the first three chapters. Read these chapters sequentially to build a foundational understanding of Kotlin and its core features.

Chapter 1: Kotlin Basics Introduces Kotlin basics like comments, variables, operators, flow control, functions, lambdas, and input/output. Teaches you to write clear, concise, and reusable code via many examples and exercises, and walks you through a console-based calculator project.

Chapter 2: Arrays, Collections, and Classes Introduces data manipulation and object-oriented programming. Explains how to use arrays, lists, mutable lists, classes, encapsulation, inheritance, polymorphism, abstract classes, interfaces, and enum classes. Discusses the benefits and trade-offs of each concept and how to apply them in your programs. The project in this chapter involves developing a simple console-based task management app.

Chapter 3: Visualizing with JavaFX Covers the basics of JavaFX, a toolkit for creating graphical user interfaces and data visualization. Explores JavaFX components such as `Stage`, `Scene`, layout containers, and nodes. Teaches you to use various types of charts, add dynamic simulation in your code using JavaFX's animation feature, and use the Canvas tool for free-form drawing through a series of projects.

Part II, "Applications in Math and Science," contains a series of short projects that introduce new concepts and allow you to apply your newly acquired coding skills to real-world problems.

Chapter 4: Solving Mathematical Problems with Code Demonstrates how to use Kotlin code and algorithms to solve mathematical problems with real-world applications. Covers various concepts and tools from mathematics, such as arithmetic, trigonometry, the Pythagorean theorem, the Fibonacci sequence, the haversine formula, modulo operations, and linear algebra. Explores stochastic processes and random number generation and leverages many core features of Kotlin, such as variables, data classes, conditional and iterative structures, and library functions.

Chapter 5: Modeling and Simulation Shows how to model and simulate the motion of various physical systems, such as cannonballs, water fountains, pendulums, thermal heating and cooling, and binary stars. Uses math, visualization, and animation to help you understand the dynamics and behavior of these systems over time.

Part III, "Recursion, Sorting, and Searching," presents more advanced concepts and tools that will require significant effort to master, resulting in significant rewards. All key concepts are developed and explored as independent projects.

Chapter 6: Recursive Functions and Fractals Introduces the fantastic world of fractals, self-similar patterns that repeat at different scales. Teaches you how to use recursive functions, L-systems, and the Mandelbrot set to generate fractals, then render the fractals and explore their properties with JavaFX.

Chapter 7: Sorting and Searching Focuses on sorting and searching, two fundamental topics in computer science and data science. Shows you how to implement and compare three common sorting algorithms (insertion sort, merge sort, and quick sort) and explore three graph traversal algorithms (depth-first search, breadth-first search, and A*). Explains how to use stacks, queues, lists, and maps to store and manipulate data for sorting and searching. Also discusses the time and space complexities of these algorithms using big O notation.

Finally, Part IV, "Optimization with Nature-Inspired Algorithms," introduces cutting-edge ideas learned from the natural world that will help you solve difficult optimization problems. These methods will also introduce you to the world of stochastic (probabilistic) and heuristic algorithms.

Chapter 8: The Genetic Algorithm Discusses the advantages and challenges of using stochastic algorithms, which exploit randomness to deal with intractable or complex problems. Explores genetic algorithms, a class of process-based nature-inspired algorithms (NIAs) that have been used to solve problems involving text matching, combinatorial optimization (the knapsack problem), and finding global optima for a multivariate function by developing three separate applications.

Chapter 9: Agent-Based Algorithms Covers two agent-based NIAs: particle swarm optimization and ant colony systems. Shows how to

harness the power of these algorithms in Kotlin by creating applications for function optimization and solving a traveling salesman problem.

Afterword A short section that tells you where to go next to continue your coding journey by enhancing your coding skills and learning other advanced features of Kotlin.

The Projects

The 37 projects in this book are designed to help you flex your coding and problem-solving skills. Each project poses an interesting challenge, systematically outlines a strategy for solving it, and walks you through implementing that strategy as a Kotlin app. All code and background information relevant to the problem are thoroughly explained. Here's a look at each of the projects and everything you'll accomplish as you work through them:

Project 1: Build a Console-Based Calculator Learn to think through a problem systematically and develop a well-organized program to solve it. Utilize core Kotlin language features such as conditionals and functions to design an interactive calculator and practice validating user input.

Project 2: Build a Versatile Task Manager Design and implement a console-based task manager application that enables users to add and delete tasks, display the task list, update task statuses, and more. Structure the app and its data using lists, classes, and data classes and incorporate robust error handling as well.

Project 3: Build "Hello, World!" in JavaFX Discover the classes and layout containers that form the fundamental building blocks of any JavaFX visualization. This project serves as a basic template for subsequent projects requiring more elaborate visualization schemes.

Project 4: Visualize Data as a Bar Chart Learn about the core charting tools of JavaFX by creating a bar chart using the monthly sales data from a fictitious company. Plot text data on one axis and numeric data on the other.

Project 5: Create a Multiseries Line Chart Visualize multiple datasets at once by developing a line chart that compares the average heights of men and women of different ages. Plot numeric data to both the x- and y-axes.

Project 6: Draw a Spiral Seashell Use Kotlin's built-in trigonometric functions along with the JavaFX canvas to draw a spiral seashell. This is the first of many projects where you'll leverage the powerful free-form drawing features of JavaFX to create a complex two-dimensional image.

Project 7: Animate a Square Define the start and end state of a solid red square and let JavaFX transitions do their magic by smoothly moving the square back and forth inside the graphics window.

Project 8: Animate a Bouncing Ball Harness JavaFX's powerful timeline and keyframe features and manipulate object properties to create captivating motion. Animate a ball gracefully gliding inside a rectangular box.

Project 9: Find the Square Root with the Babylonian Algorithm Develop a simple app that converges on the square root of any positive number by iteratively refining an approximation of its value. Try to figure out what technique Kotlin's built-in square root function employs to do the same.

Project 10: Create Pythagorean Triples with Euclid's Formula Implement Euclid's method for generating Pythagorean triples (sets of integers where the sum of the squares of the first two numbers equals the square of the third number).

Project 11: Identify Prime Numbers with the Sieve of Eratosthenes Take a peek into the world of prime numbers and utilize an ingenious method developed by Eratosthenes to reveal them up to a given integer.

Project 12: Calculate Earth's Circumference the Ancient Way Time travel back to ancient Egypt to re-create an experiment that uses simple trigonometry to estimate Earth's circumference and radius with remarkable accuracy.

Project 13: Code the Fibonacci Sequence Generate a list of Fibonacci numbers from scratch and then draw the Fibonacci spiral. Use advanced JavaFX features to explore how these concepts are embedded in both natural and man-made systems.

Project 14: Find the Shortest Distance Between Two Locations on Earth Utilize the haversine formula and an understanding of latitude and longitude to calculate the shortest distance between any two locations on Earth's surface while accounting for Earth's spherical shape.

Project 15: Do Encryption with the Hill Cipher Implement a classic encryption algorithm that's based on the linear transformations of texts. Convert plaintext into ciphertext and vice versa using vectors, matrices, and modular arithmetic while employing multidimensional array operations.

Project 16: Simulate a One-Dimensional Random Walk Learn to simulate random processes by investigating how far an entity will travel after taking n random steps. Use the root-mean-squared (RMS) distance to measure how far the walker goes, and run the simulation thousands of times to compare the simulated RMS values with theoretical ones.

Project 17: Predict the Flight of a Cannonball Employ the bisection numerical method to find the root of a nonlinear function in order to tackle the age-old problem of determining the correct firing angle for a cannonball.

Project 18: Design a Fountain with Water Jets Continue exploring the physics of projectiles by designing a multitier water fountain. Combine core Kotlin skills with advanced JavaFX techniques to generate and display the fountain's structure and the water jets' trajectories.

Project 19: Track a Pendulum's Motion and Phase Leverage Newtonian laws of motion and gravity to calculate and plot a pendulum's angular displacement and velocity along with its phase-space over time. Incorporate the impact of air drag to make the pendulum's motion more realistic.

Project 20: The Physics of Coffee Cooling Create a Kotlin app that tackles a quintessential dilemma: Should you add cream to your coffee when you buy it at the coffee shop or when you get back to the office? Apply principles of Newtonian cooling to keep your coffee as warm as possible.

Project 21: Simulate a Binary Star System Animate the intricate dance of two stars orbiting each other. As time progresses, the stars gracefully trace their paths, allowing a delightful exploration of stellar dynamics.

Project 22: The "Hello, World!" of Fractals Learn the features of fractal geometry by drawing a series of concentric squares. Write your first recursive function and set its stopping condition.

Project 23: Draw the Sierpiński Triangle Realize a classic fractal pattern named after mathematician Wacław Sierpiński. Translate the pattern's geometry into a codable strategy and make multiple recursive calls from within the same function.

Project 24: Create a Fractal Tree Develop a Kotlin program that draws a fractal tree, starting with a trunk and recursively generating smaller branches in opposite directions.

Project 25: Design an L-System Simulator Code an interactive L-system simulator from scratch, including a Kotlin implementation of Turtle Graphics. L-systems generate fractals using an alphabet of symbols, a starting axiom, and a set of rules defining how symbols are iteratively replaced.

Project 26: Code and Visualize the Mandelbrot Set Calculate and draw the Mandelbrot set, a captivating two-dimensional fractal defined in the complex plane. This mesmerizing image emerges from simple iterative rules and reveals intricate structures when magnified ad infinitum.

Project 27: Space-Efficient Sorting with Insertion Sort Explore the world of sorting algorithms and big O notation. Though it's slower than other popular sorting methods, insertion sort operates in place, resulting in a low-space complexity.

Project 28: Faster Sorting with Merge Sort Implement merge sort, a speedy sorting algorithm that's stable but less memory efficient than

insertion sort. Recursively divide an array of values into smaller subarrays, sort them, and merge them back together.

Project 29: High-Efficiency Sorting with Quick Sort Code the quick sort algorithm, a fast and memory-efficient algorithm perfect for large datasets. Recursively select a pivot element in an array and divide the other elements into subarrays with values less than and greater than the pivot.

Project 30: Stack-Based Searching with Depth-First Search Implement depth-first search (DFS), one of several algorithms for traversing or searching a graph data structure. Use a stack to explore each path to its fullest, and backtrack when a dead end is reached.

Project 31: Queue-Based Searching with Breadth-First Search Use a queue to realize the breadth-first search (BFS) algorithm. BFS explores all nodes at the current depth before moving on to nodes at the next level of depth.

Project 32: Heuristic Searching with A* Implement A*, an informed search algorithm that uses a heuristic function to guide its search. Given a weighted graph, a source node, and a goal node, A* finds the shortest path from the source to the goal.

Project 33: Evolve Gibberish into Shakespeare Use principles of heuristics and natural evolution to go from random strings to a famous line from Shakespeare. Develop modular, reusable code for the genetic algorithm, modeling genetic operators like elitism, selection, crossover, and mutation.

Project 34: Solve the Knapsack Problem Unleash your genetic algorithm code on the knapsack problem, one of a class of combinatorial optimization problems where the global optimal solution is notoriously hard to achieve.

Project 35: Optimize a Multivariate Function with the Genetic Algorithm Adapt the genetic algorithm to solve an optimization problem with a complex, multivariate, nondifferentiable mathematical function. Employ real-valued genes to represent the decision variables and locate the global optimal solution within a given decision space.

Project 36: Optimize a Multivariate Function with a Particle Swarm Implement the particle swarm optimization (PSO) algorithm, a nature-inspired, agent-based method that seeks optimal solutions to complex problems by emulating swarm behavior. Write code that actively pursues continuous improvement through individual and collective learning.

Project 37: Solve the Traveling Salesman Problem with an Ant Colony System Use an ant colony system, another nature-inspired algorithm, to tackle Berlin52, a notoriously challenging optimization problem. Discover the optimal route through 52 stations in Berlin from among the astronomical 1.551119×10^{66} distinct ways to arrange the tour.

Getting Started

To get the most out of this book, read the text carefully and run the code examples yourself. You can run the code on any Windows, Mac, or Linux operating system. You can also run the code online if it does not require any graphics elements.

If you're completely new to coding, you'll need to familiarize yourself with some key terms before you start, such as *text editor, integrated development environment (IDE), library, compiler, debugging,* and *executing* or *running a program.*

To help you get started, this book's appendix includes instructions on how to run Kotlin programs, as well as definitions of the aforementioned key terms. If you're already familiar with these concepts and know how to install and set up an IDE, you can skip this section. Otherwise, make sure you've completed all the steps listed there before you begin your coding journey in Kotlin.

You can find the code examples discussed in this book on GitHub at *https://github.com/imfaisalgit/KotlinFromScratch.*

Resources

Kotlin. "Kotlin Docs." (The official Kotlin documentation.) Accessed June 15, 2024. *https://kotlinlang.org/docs/home.html.*

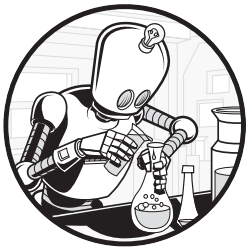
Kotlin Playground. Online Kotlin editor and compiler. Accessed June 15, 2024. *https://tinyurl.com/59hauntd.*

PART I

PROGRAMMING FUNDAMENTALS

1

KOTLIN BASICS



This chapter will guide you through the fundamental building blocks of the Kotlin programming language. We'll explore essential language features such as comments, variables, operators, flow control structures, functions (including lambda expressions), and basic input and output techniques.

Together, these elements allow you to manage and manipulate data within your code, govern the program's behavior so that it responds dynamically to different scenarios, and keep your code well organized and easy to maintain. Understanding these basic elements will also pave the way for the more sophisticated language features discussed in Chapter 2, such as arrays, collections, and custom data structures (including classes).

The topics covered here are core elements in any programming language, serving as the foundational elements for application development. If you've already worked with another language, these elements will look familiar in Kotlin. Conversely, if Kotlin is your first language, what you

learn here will be readily transferable to other languages. We'll explore these components in a practical way, with short code snippets illustrating each new idea. At the end of the chapter, we'll tie everything together with a simple project.

I'm assuming you're using the free version of IntelliJ IDEA as your integrated development environment (IDE) for developing and running code. See the appendix for instructions on installing, setting up, and using this tool with a basic "Hello, world!" program.

Using Comments

A *comment* is a line (or several lines) of explanatory text in a code file that the compiler will ignore when it runs the code. The text is meant to provide helpful hints, such as what the next code segment does, why a particular method was chosen, or how to properly use a language feature in a code segment. When you're coding, you should insert comments to remind yourself what each piece of code is about. Experienced coders know that documenting code is essential to guard against forgetting important details down the road. Also, when others use or build on your code, your comments can be a lifesaver.

In Kotlin, you have two main ways to add comments. The first is to use // to start a single-line comment. The compiler will ignore anything on the line after the two slashes. The other way is to use /* and */ to start and end a comment that spans multiple lines. Here's an example of how these two commenting styles work:

```
// This is a single-line comment.

/*
   This code block will be ignored by the compiler
   as it is inside a multiline comment block.
*/
```

Kotlin also provides a third type of comment, which is used for automatically generating documentation. This sort of comment begins with /** and ends with */. Documentation comments are used to describe variables, functions, and classes more formally, and they often include tags such as `@param`, `@return`, and `@throws` to explain standard aspects of a piece of code. Here's an example that compares multiline and documentation comments:

```
/* This is a multiline comment
   used for providing useful tips or reminders. */

/**
 * This is a documentation comment.
 *
 * @param name The name of the person
 * @return The greeting string
 */
```

```
fun greet(name: String): String {
    return "Hello, $name!"
}
```

While these two types of comments use a similar syntax, they serve different purposes. Multiline comments are intended to be read within the code file itself. Documentation comments, on the other hand, are intended to be exported from the code file to generate official documentation for production-ready code that other developers can consult.

Variables

In coding, a *variable* is a name given to a data element. For simplicity, we can think of variables as containers that hold various data types in a computer's memory. Once assigned, the variable name can be used as a stand-in for the value it represents. In this way, variables allow us to store and manage data, enabling the persistence of information within a program.

Every variable should have a meaningful name that clearly describes its purpose or function or otherwise reflects the nature of the data assigned to it. For example, a variable holding a person's name could be called `name`, and a variable holding a person's age could be called `age`. By convention, variable names should consist of a lowercase word or use *camelCase* to join multiple words. In the latter case, no spaces appear between words, and every word after the first starts with a capital letter, as in `lastName` or `ageInYears`.

In Kotlin, you create a new variable by *declaring* its name with a keyword such as `val` or `var` and *initializing* it (assigning it to a value). (A *keyword* is a reserved word that has a special meaning in a programming language. A keyword can't be used as an identifier—for example, as a variable or function name.) Which keyword you use depends on whether you want the variable's value to stay the same or change during program execution. A variable declared with `val` is *read-only*, meaning its value can't change after it's been initialized. A variable declared with `var` is *mutable*, meaning the variable can be assigned a different value after it's been initialized. You can change the value of a mutable variable as many times as needed.

Consider this example, where we use two variables to create a message:

```
fun main() {
    val name = "John Sinclair"
    val age = 30
    println("$name is $age years old")
}
```

We declare two variables, `name` and `age`, and assign them the values `"John Sinclair"` and `30`, respectively. Both variables are declared with the `val` keyword, so they can't be reassigned to different values later. We then include both variables in a message to be printed to the console by adding a dollar sign (`$`) before each variable name. (We'll discuss how this syntax works in

more detail in "Working with Strings" on page 14.) If you run this code (using CTRL-SHIFT-F10 in IntelliJ IDEA), the output should look like this:

```
John Sinclair is 30 years old
```

Notice how the output shows the values assigned to the `name` and `age` variables rather than the variable names themselves. But what if we want to assign new values to these variables over the course of the program? For that, we have to use the `var` keyword instead of `val`, as shown here:

```
fun main() {
    var name = "John Sinclair"
    var age = 30
    println("$name is $age years old")
  ❶ name = "John Sinclair Jr."
    age = 12
    println("$name is $age years old")
}
```

Here we declare the `name` and `age` variables with the `var` keyword, giving them the same initial values as before. Then we assign them new values ❶. Notice that the second time around, we no longer need the `var` (or `val`) keyword when setting the values of the variables. Once we've declared and initialized a variable for the first time, we can work with the variable using only its name.

If you now run the program, this is what you should see:

```
John Sinclair is 30 years old
John Sinclair Jr. is 12 years old
```

We've successfully reassigned the variable names because they were declared with the `var` keyword. Try changing the `var` keyword back to `val` for one or both variables and running the code again. The IDE will instantly generate an error message about how you can't assign a new value to a variable declared with `val`, and it won't let you run the program until you fix the error.

Constants

Kotlin also provides the `const` keyword (short for *constant*) for setting the immutable value of a variable at the beginning of a file. The value must be known during compilation of the code. Declaring a variable with `const` is allowed only for primitive types or strings. (We discuss the common data types in Kotlin in the next section.) The judicious use of constants has two important benefits: it improves program efficiency in accessing fixed values, and it improves code clarity by avoiding hardcoded "magic numbers" deep inside the code without clear context. Here's an example of creating a variable with the `const` keyword:

```
const val PI = 3.14159265359
```

In this case, we know the value of the mathematical constant pi, and we know that this value won't change over the course of a program, so it makes sense to declare it using the `const` keyword at the start of the program. In Kotlin, it's customary to use all caps for top-level constant names, as we've done here for `PI`. Multiple words can be joined using an underscore.

Kotlin has many other naming conventions for various code constructs. Table 1-1 summarizes the most common ones.

Table 1-1: Naming Conventions in Kotlin

Name	Convention	Example
Package name	Use lowercase letters with no underscores. Join multiple words or use camelCase. Use reverse domain notation (auto-generated by the IDE).	`org.example.myProject`
Class name	Use PascalCase for class and inheritance names. Choose words that are nouns or noun phrases.	`FlightSimulation`
Function name	Use camelCase for function and method names. Use verbs or verb phrases.	`calculateShortestPath()`
Variable name	Use a single word or camelCase to join multiple words. Choose a word that describes the purpose, function, or property of the variable (make it meaningful).	`username`
Constant and final variable name	Use uppercase letters with underscores to separate words.	`MAX_VALUE`

These naming conventions are based on the recommendations in the official Kotlin documentation at *https://kotlinlang.org*. We'll revisit them as we discuss the code constructs they relate to.

Common Data Types

A value in code can be of various *data types*. For example, a value might represent a number, some text, or a logical value (`true` or `false`). In Kotlin, each variable is associated with a specific data type, and once a variable's data type has been set, it can't hold values of other types. A variable with a numeric value can be associated with different types, such as `Int` for whole numbers only or `Double` or `Float` for numbers with decimal components. A variable holding text values can be of type `Char` for a single character or type `String` for multiple characters. A logical value will have a `Boolean` type. Table 1-2 lists the common data types in Kotlin and their key characteristics.

Table 1-2: Common Kotlin Data Types

Data type	Description	Size (in bits)	Range of values
`Byte`	Signed integer	8	–128 to 127
`Short`	Signed integer	16	–32,768 to 32,767
`Int`	Signed integer	32	–2,147,483,648 to 2,147,483,647
`Long`	Signed integer	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`Float`	Floating point (single precision)	32	–3.4028235E+38 to 3.4028235E+38
`Double`	Floating point (double precision)	64	–1.7976931348623157E+308 to 1.7976931348623157E+308
`Char`	16-bit Unicode character	16	0 to 65,535 (in decimal)
`Boolean`	Represents true or false	1	`true` or `false`
`String`	A sequence of characters	Varies	N/A

We choose the data type for a variable based on problem-specific needs, focusing on factors such as the data type's ability to accommodate values, its level of precision, and its memory utilization. If you know a numeric variable will hold only integer values, for example, `Int` would be a better choice than `Float`. If those values need to be exceptionally large, `Long` would be more appropriate than `Int`.

Type Inference

When declaring a variable in Kotlin, it isn't mandatory to explicitly specify its data type. The Kotlin compiler excels at *inferring* the variable's data type based on the assigned value. Here, for example, Kotlin infers that `name` is of type `String`, since its value is a series of characters enclosed in quotation marks:

```
val name = "John Sinclair"
```

You also have the option to explicitly declare a variable's data type. Here's how to explicitly declare `name` as a `String`:

```
val name: String = "John Sinclair"
```

To declare the data type, we place a colon after the variable name, followed by the desired type. This can be any acceptable data type, including custom data types you might have created to represent complex structures with specific properties and behaviors (for example, classes or data classes, which we'll discuss in Chapter 2).

For numeric values, Kotlin infers the `Int` type if a variable is first assigned a whole-number value or the `Double` type if it's assigned a value

with a decimal component. If you need a different numeric type, you can explicitly indicate it by using a type suffix at the end of the value itself, such as `L` for `Long` and `f` for `Float`. For example:

```
val regularInt = 42
val floatNumber = 3.14f
val longNumber = 123456789L
```

In this code, `regularInt` is inferred as `Int` by default, while `floatNumber` and `longNumber` are explicitly declared to be of types `Float` and `Long`, respectively, using the `f` and `L` suffixes.

Type Casting

Type casting, also known as *type conversion*, is the process of changing a variable's or expression's data type to another compatible data type. (An *expression* is a code snippet that evaluates to a particular data type.) This process is primarily used to resolve type mismatches. In general, Kotlin enforces strict type checking to prevent common sources of runtime errors, such as unintended type conversions. Implicit type casting may be allowed only when no risk of data loss or unexpected behavior exists. For example, smaller numeric types can be implicitly promoted to larger numeric types since no risk of data loss exists, as shown here:

```
val intNumber = 22  // type inferred as Int
val longNumber: Long = intNumber  // implicit type casting
```

We take the value of the `intNumber` variable and assign it to the `longNumber` variable, implicitly converting the value from type `Int` to `Long`. This may work, but the default setting of most IDEs is to prevent the use of implicit type casting altogether. Instead, you're encouraged to use explicit type-casting methods to achieve type conversion. Some of the common type-casting methods supported in Kotlin include `toByte()`, `toInt()`, `toLong()`, `toShort()`, `toDouble()`, `toFloat()`, `toChar()`, and `toString()`. Here is an example of explicit type casting:

```
val intNumber = 44  // type inferred as Int
val doubleNumber: Double = intNumber.toDouble()
```

We first create the `intNumber` variable and assign it a value of `44`. The compiler will infer `intNumber` to be of the `Int` type. Then we explicitly cast it to type `Double` using the `toDouble()` method before assigning it to `doubleNumber`.

Since type casting is allowed only when the associated data types are compatible, not all conversions are possible. For example, you can't always convert a text type into a numeric or logical type. To illustrate, try running the following lines of code:

```
val message: String = "Hello, world!"
val intValue: Int = message.toInt()
```

Here we try to take the string in the `message` variable and cast it to an integer with the `toInt()` method. This will throw a `NumberFormatException` error at runtime, indicating the conversion isn't possible due to the incompatible data types. This error seems reasonable: How should Kotlin know the numeric equivalent of the `"Hello, world!"` string?

> **FUNCTIONS AND METHODS**
>
> In Kotlin, a *function* is a block of code that performs a specific task, such as adding two numbers and returning their sum. It's the basic unit of code organization. A *method* is a specific type of function that's declared as part of a class or data type. Type-casting functions like `toInt()` and `toString()` are technically methods because they're associated with values of a particular data type.
>
> Whereas a regular function is invoked simply by using the function name, as when we call the `println()` function, a method is invoked using *dot notation*. With this syntax, you provide a value, followed by a period, followed by the name of a method associated with that value. For example, `age.toString()` calls the `toString()` method on the value stored in the `age` variable (presumably an integer), creating a string version of that value.
>
> We'll discuss functions in detail later in this chapter, and we'll discuss methods in Chapter 2, when we study classes.

Operators

Operators are special symbols for manipulating variables and other values in your code. Each operator performs a specific mathematical, logical, or text-based operation. In this section, we'll review the most common categories of operators in Kotlin.

Arithmetic

Arithmetic operators are for performing basic mathematical operations such as addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Here are some examples of how to use arithmetic operators in Kotlin:

```
val a = 20
val b = 7
val sum = a + b            // addition, yields 27
val difference = a - b     // subtraction, yields 13
val product = a * b        // multiplication, yields 140
val quotient = a / b       // division, yields 2
```

Here we apply the four main arithmetic operations to the `a` and `b` variables. Notice that when you divide two integers using the division operator `/`,

the result is also an integer, with any fractional part discarded. In this case, `20 / 7` gives a result of `2`, not `2.857143`. If you need to retain the fractional part, you must convert one of the numbers into a floating-point number, as shown here:

```
val a = 20
val b = 7
val quotientInt = a / b                // integer division
val quotientFloat = a.toFloat() / b    // real division
```

Here, `quotientInt` will have a value of `2`, but `quotientFloat` will have a value of `2.857143` since we use `toFloat()` to convert `a` from an integer to a floating-point number.

The *remainder* or *modulo* operator is another mathematical operator we'll use many times in this book. It's designated by the `%` symbol. This operator returns only the remainder from the integer division of two numbers. Here's an example:

```
val a = 20
val b = 7
val result = a % b // The result is 6.
```

In this example, `a % b` returns the remainder when `a` is divided by `b`. Since 20 divided by 7 has a remainder of 6, the value of `result` is `6`. Can you guess what the result would be if we flipped the numbers around—that is, if we calculated `7 % 20`? Additionally, what would be the result of the integer division `7 / 20`? These questions may sound trivial, but I encourage you to write a few lines of code to verify your guesses.

Kotlin uses the same order of operations for arithmetic operators as regular mathematics: division and multiplication take precedence over addition and subtraction. To avoid confusion about the order of operations, it's good practice to use parentheses to clearly isolate different operational blocks. For example:

```
fun main() {
    // example without parentheses
    val resultWithoutParentheses = 5 + 3 * 2
    println("Result without parentheses: $resultWithoutParentheses")

    // example with parentheses
    val resultWithParentheses = (5 + 3) * 2
    println("Result with parentheses: $resultWithParentheses")
}
```

In the first calculation, `5 + 3 * 2`, multiplication takes precedence over addition, so it evaluates to `5 + (3 * 2)`, resulting in `11`. In the second calculation, `(5 + 3) * 2`, the addition inside the parentheses is performed first, and then the multiplication, resulting in `16`. This demonstrates how using parentheses can clarify and control the order of operations in mathematical expressions.

Assignment

Assignment operators are used to assign values to variables. We've already been using the main assignment operator (=) throughout this chapter's examples to set a variable's value from scratch. Other assignment operators, like += and -=, take a variable's existing value and modify it. Here are some examples:

```
var a = 10
a += 5          // equivalent to a = a + 5 (a becomes 15)
a -= 5          // equivalent to a = a - 5 (a becomes 5)
a *= 5          // equivalent to a = a * 5 (a becomes 50)
a /= 5          // equivalent to a = a / 5 (a becomes 2)
```

The assignment `a += 5` is equivalent to saying, "Take the value of `a`, add `5` to it, and put the result back in the `a` variable." Similar assignment operators exist for the other three arithmetic operations.

You can try using `+=` with a string variable too, if it was declared with `var`. For example:

```
var s = "John Smith"
s += " Jr."           // The s becomes "John Smith Jr."
```

Be mindful that this operation essentially creates a new string and assigns it to the previously used variable name, rather than directly modifying the old string (which is discarded). For strings, other assignment operators (for example, `-=`) will generate errors.

Unary

Whereas most operators have two operands, unary operators have just one. The *increment* (`++`) and *decrement* (`--`) unary operators increase or decrease a variable's value by `1`, respectively. Here's how to use these operators in Kotlin:

```
var a = 10
a++         // equivalent to a = a + 1 (a becomes 11)
a--         // equivalent to a = a - 1 (a becomes 10 again)
```

Essentially, `a++` is a shorter way of writing `a += 1`, itself a shorter way of writing `a = a + 1`. Likewise, `a--` is equivalent to `a -= 1`.

Relational

Relational operators compare two values and return a Boolean value (`true` or `false`) based on the comparison. These operators include `==` and `!=` for equality and inequality, and `>` and `<` for greater than and less than. Here are some examples of these operators in action:

```
val a = 10
val b = 5
val isEqual = (a == b)      // equality check
```

```
val isNotEqual = (a != b)   // inequality check
val isGreater = (a > b)     // greater than check
val isLesser = (a < b)      // less than check
```

In this code segment, `isEqual` will be `false` because `a` and `b` aren't equal, and `isNotEqual` will be `true`. Meanwhile, `isGreater` will be `true` because `a` is greater than `b`, and `isLesser` will be `false`. Notice how we put each comparison in parentheses. This isn't strictly necessary, but it helps visually separate the comparison from the assignment operation it's a part of.

The preceding examples used numeric values, but relational operators can also be used for comparing strings:

```
val text1 = "Hello"
val text2 = "World"
val isNotEqual = (text1 != text2) // true
val isGreater = (text1 > text2)   // false
```

In Kotlin, strings are compared lexicographically, character by character, based on their Unicode values. The comparison starts from the first character of each string and continues until a difference is found or one of the strings ends. The string with the smaller Unicode value at the first differing character is considered lesser. This implies that earlier letters in the alphabet are considered lesser than later letters and that uppercase letters are considered lesser than lowercase letters.

Logical

Logical operators are used to perform logical operations such as AND (`&&`), OR (`||`), and NOT (`!`) on Boolean values. Here are some examples of how to use logical operators in Kotlin:

```
val x = true
val y = false

val andResult = (x && y)   // logical AND operation (returns false)
val orResult = (x || y)    // logical OR operation (returns true)
val notResult = !x         // logical NOT operation (returns false)
```

The result of a logical operation involving two Boolean values can be summarized in a *truth table* like the one shown in Table 1-3. A truth table shows the output corresponding to every possible combination of input values.

Table 1-3: Truth Table for Two Logical Values

Value 1	Value 2	AND	OR
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

In this table, the operands are Boolean values, which can be either `true` or `false`. For example, in the first row, both Value 1 and Value 2 are `true`. The result of an AND operation on these operands is `true`, and the result of an OR operation is also `true`. Unlike AND and OR, a NOT operation has just one Boolean operand, which gets inverted. For instance, the NOT operator turns `true` into `false`.

Working with Strings

In Kotlin, a *string* is a sequence of characters represented by the `String` data type. Strings are helpful for storing and manipulating text data in a program. They're commonly used to represent words, sentences, and other textual information. User input data is also initially read as strings and then converted as needed into other types using functions such as `toInt()`, `toDouble()`, and `toBoolean()`.

The individual characters in a string are numbered, or *indexed*, starting from zero. You can access a specific character in a string by placing its index in square brackets after the variable name representing that string. For example, to get the second character of a string in the `msg` variable, use `msg[1]`. Alternatively, you can use the `get()` method of the `String` class to retrieve the character at a specific index. For example, `msg.get(1)` returns the same second character as `msg[1]`.

In this section, we'll discuss some common techniques for working with strings. Keep in mind as you read that strings are immutable objects in Kotlin, so once a string is created, its content can't be changed. Any operation that appears to modify a string creates a new string, and the original value is discarded.

Concatenation

Concatenation is the process of combining two or more strings into a single string. In Kotlin, you can do this in multiple ways. For example, you can use the `+` operator, or you can use a string's `plus()` method. Both techniques are demonstrated here:

```
val a = "Hello,"
val b = "world!"

// Use the plus ( + ) operator.
var c = a + " " + b
println(c) // output: Hello, world!

// Use the plus() method of the String class.
c = a.plus(" ").plus(b)
println(c) // output: Hello, world!
```

In this code segment, we first create two string variables, `a` and `b`, with values `"Hello,"` and `"world!"`, respectively. Then we use the `+` operator to concatenate (join) the two strings and add a space (the string `" "`) in

between, assigning the resulting string to the `c` variable. The output of `println(c)` is:

```
Hello, world!
```

Notice how we call the `plus()` method twice in the same expression to concatenate string `a` with a space and string `b`. This technique is known as *method chaining*; the second method call is applied to the result of the first.

Another way to join multiple strings is to use the `buildString` function, as in this example:

```
val c = buildString {
           append("Hello,")
           append(" ")
           append("world!")
       }
println(c) // output: Hello, world!
```

We create a variable `c` to hold the concatenated string and append all the string segments through a single call to the `buildString` function.

String Templates

Most characters that appear between the opening and closing quotation marks of a string are interpreted literally as ordinary text. *String templates*, however, are a powerful feature in Kotlin that allow you to embed code within a string. They're a concise and expressive way to combine static text with dynamic values resulting from variables, expressions, or even function calls. As hinted at when we first discussed variables, string templates use dollar signs (`$`) to signal that what follows should be treated as code rather than as literal text. For example, consider the following code snippet:

```
val name = "John"
val age = 30
val message = "My name is $name and I'm $age years old."
println(message)
```

Here we embed the variables `name` and `age` within the `message` string template using the `$` symbol. The values of these variables are automatically substituted into the resulting string when it's evaluated. When you run this code snippet, the output should therefore look like this:

```
My name is John and I'm 30 years old.
```

Notice how `John` and `30` have been inserted into the string in place of `$name` and `$age`. Compare the string template syntax with how we might generate the same message using string concatenation:

```
val message = "My name is " + name + " and I'm " + age + " years old."
```

The string template version is much more readable and spares us from the tedium of including all the + operators and remembering to add spaces to the strings before and after each variable. This isn't to say that string concatenation is never useful, but if your goal is to inject values from your code into a string, a string template is likely the better choice.

String templates can also handle more complex expressions by enclosing them within braces (the { and } symbols) after the dollar sign. This allows you to perform calculations, access object properties, or invoke functions directly within the template. Here's an example:

```
val x = 20
val y = 15
val result = "$x + $y = ${x + y}"
println(result) // output: 20 + 15 = 35
```

The `${x + y}` in this string template tells Kotlin to add the values in the `x` and `y` variables and insert the result into the string.

Escape Sequences

Escape sequences are special character combinations used in strings to represent characters such as whitespace that are difficult to input directly. They're prefixed with a backslash. For example, the escape sequence `\n` represents a newline character, and `\t` represents a tab character. Escape sequences are often used within string templates to format the output. Here's an example:

```
fun main() {
    val name = "John"
    val age = 30

    // using escape characters in string template
    val message = "Name: $name\nAge: $age"

    println(message)
}
```

In this example, the `\n` in the middle of the `message` string adds a line break between the name and age, improving the formatting of the output. If you run the code, the output should break across two lines, like this:

```
Name: John
Age: 30
```

Other common escape sequences include `\\` for a backslash and `\$` for a dollar sign. These are needed because otherwise a backslash in a string is interpreted as the start of an escape sequence and a dollar sign as the start of some Kotlin code to be inserted into a string template.

Null and Nullable Types

Null represents the absence of a value. By default, Kotlin's type system assumes that a variable can't hold `null`. Consider this example:

```
var str: String = "Hello, world!"  // valid initialization
str = null  // invalid, will result in compilation error
```

The `str` variable is declared to be of the `String` type, so its value must be a string. Trying to set the variable to `null` will therefore trigger an error. If you want to allow a variable to be `null`, you must explicitly declare the variable to be of a *nullable type* by appending `?` to the type declaration, as shown here:

```
var str: String? = "hello world" // valid initialization
str = null // reassigned to null, no compilation error
```

The type declaration `str: String?` indicates that the `str` variable can hold either a string or `null`. Because the variable is of the `String?` nullable type, setting it to `null` is now valid and won't cause an error.

Kotlin's not-nullable-by-default type system is designed to prevent *null pointer exceptions*, which are common runtime errors in languages that freely allow variables to have null values. A null pointer exception occurs when a program tries to access or manipulate data using a null reference, which doesn't point to a valid memory location or object. Proper handling of null values is essential to prevent these exceptions and the code from crashing.

You can ensure null safety in Kotlin in multiple ways. One way is to explicitly check whether a nullable variable holds `null`, before accessing its properties or methods. For instance:

```
val str: String? = "Hello, world!"
val len = if (str != null) str.length else -1
```

In this example, we check whether the variable `str` is `null` by using an `if... else` conditional statement. If `str` is not `null`, its `length` property is accessed; otherwise, `-1` is assigned to `len`. (We'll discuss conditional statements in the next section.)

Another mechanism is Kotlin's *safe call operator* (`?.`), which allows us to call a method on a nullable object without risking an error. If the object contains `null`, the result is `null` as well. Otherwise, the method is called as usual. For example:

```
val len = str?.length
```

In this case, if `str` is `null`, `len` will be assigned `null`; otherwise, `len` will be assigned the length (number of characters) of `str`.

Kotlin also provides the *Elvis operator* (`?:`). Used in conjunction with the safe call operator, it allows you to give a default value (instead of `null`) for an expression involving a nullable object. If the object isn't `null`, the

expression will be evaluated as normal; otherwise, the default value will be used. For example:

```
val len = str?.length ?: -1
```

In this case, if `str` isn't `null`, its length will be assigned to the `len` variable; otherwise, the value after the `?:` operator (`-1`) will be assigned.

NOTE *The examples we've discussed so far have focused on the `String?` nullable type, but note that the `?` operator can be applied to other data types, such as `Int`, `Double`, or even `Boolean`. This flexibility in using nullable types is particularly helpful when working with user input.*

Finally, Kotlin has another related operator called the *null assertion operator* (also called the *double-bang operator*), denoted by two exclamation points (`!!`). It can be used to assert that a nullable variable doesn't contain `null` even though the compiler can't guarantee this to be true. Using `!!` is a way of saying that you're sure a particular nullable variable isn't `null` so that the compiler will skip checking for nullability. Here's an example:

```
val name: String? = "John"
val length = name!!.length
```

We declare `name` as a nullable string, but by using `!!` while accessing the string's length, we're asserting that it's not `null`. If `name` actually were `null`, this snippet would result in a `NullPointerException`. As such, it's best to avoid or limit the use of the null assertion operator and favor safer constructs like safe calls (`?.`) and null checks (`?:`). Alternatively, aim to design your code to reduce nullable types, for increased reliability and predictability.

Flow Control

Flow control is an essential aspect of programming, providing mechanisms to regulate when and how code is executed. Kotlin's flow control constructs enable developers to create flexible and dynamic programs by effectively managing the sequence of statements and controlling the program's behavior. In this section we'll discuss two important types of control structures: conditionals and loops. These concepts are foundational to any programming language.

Conditional Statements

Conditional statements allow you to determine what a program should do based on certain tests. Kotlin has two primary conditional statements: `if` and `when`. They both introduce branching into code, the idea that a program can take different forks depending on the circumstances. In general, `if` statements are appropriate for straightforward binary decisions or when facing a limited number of conditions. On the other hand, if you need to

manage several conditions, handle various cases, or strive for cleaner and more structured code when implementing branching logic, `when` could be your preferred tool.

if Statements

An `if` statement runs a block of code when a given condition evaluates to `true`. Here's a simple example:

```
val x = 10
if (x > 0) println("x is positive")
```

The condition for an `if` statement, in this case `x > 0`, must be an expression that evaluates to a Boolean `true` or `false` value, and it must be given in parentheses after the `if` keyword. The code after the condition will run if the condition is `true`, so this snippet prints a message only when the variable `x` has a positive value. If `x` isn't positive, the `println` statement will simply be skipped.

An `if` statement can have an optional `else` clause that gets executed if the test expression evaluates to `false`. Once an `else` clause is included (or if there's just an `if` clause but it includes multiple lines of code), the body of each clause should be indented and enclosed in braces, as shown here:

```
val x = -10
if (x > 0) {
    println("x is positive")
} else {
    println("x is not positive")
}
```

This time, since the `x > 0` condition evaluates to `false`, the code in the `else` clause will run. Notice how the opening brace goes on the same line as the `if` or `else` keyword, while the closing brace goes on a new line after the last statement in that clause.

You can extend an `if...else` structure to include three or more possible branches by adding `else if` clauses between the initial `if` and the final `else`. Each `else if` clause adds a new condition to test if the previous condition is found to be `false`. Here's an example:

```
fun main() {
    val a = 100
    val b = -30
  ❶ val max: Int

    if (a > b) {
        max = a
        println("a is greater than b.")
        println("max of $a and $b is: $max")
    } else if (a < b) {
        max = b
        println("b is greater than a.")
```

```
        println("max of $a and $b is: $max")
    } else
        println("a and b have the same value: $a")
}
```

Here, we initialize two variables, `a` and `b`, with the values `100` and `-30`, respectively. Subsequently, we declare a variable `max` of type `Int` without providing an initial value ❶. (In Kotlin, this is allowed as long as the variable will eventually be initialized before using it for the first time.) We then use an `if...else if...else` structure to compare the values of `a` and `b` and print an appropriate message. First, the `if` clause tests if `a` is greater than `b`. If this fails, the `else if` clause tests if `a` is less than `b`. If this also fails, `a` and `b` must be equal, as the `else` clause says.

The syntax for this kind of control structure is:

```
if (condition 1) {
    // code to execute when condition 1 is true
} else if (condition 2) {
    // code to execute when condition 2 is true
} else {
    // code to execute when conditions 1 and 2 are not true
}
```

You can adjust this template by adding more `else if` blocks or deleting them altogether, depending on your needs.

when Statements

A `when` statement checks a value against multiple conditions and executes the code block of the first matching condition. If you're familiar with `switch` statements from languages like Java, C, or C++, the concept is similar. A `when` statement can also have an `else` clause for when none of the conditions match, as shown here:

```
fun main() {
    val x = 5

    when {
        x > 0 -> println("x is positive")
        x == 0 -> println("x is zero")
        x < 0 -> println("x is negative")
        else -> println("x is not a real number")
    }
}
```

We assign a value of `5` to the variable `x`, then use a `when` statement to test this variable's value. Each line of the `when` statement has its own conditional test (such as `x > 0`) followed by the `->` symbol, which points to the expression that should be executed if the condition is `true`. As soon as a `true` condition is found, the remainder of the `when` statement is skipped, even if it contains additional tests that would pass. In this case, since `x` is greater than `0`, the `when` statement will print `"x is positive"`.

As with `if` statements, including an `else` clause in a `when` statement is optional. However, it's considered good practice to provide an `else` clause; it improves the robustness of your code by offering a fallback option for unmatched cases.

It's also possible to provide the variable to be tested in parentheses immediately after the `when` keyword. In this case, the `when` statement conducts tests based on exact values or ranges of values for that variable, without repeating the variable name. For example:

```
fun main() {
    val hour = 13

    when (hour) {
        in 0..11 -> println("Good morning")
        in 12..16 -> println("Good afternoon")
        in 17..23 -> println("Good evening")
        else -> println("Invalid hour")
    }
}
```

Here, we pass the `hour` variable to the `when` statement, which prints different greetings based on the variable's value. For example, the condition `in 0..11` tests whether the value of `hour` is between `0` and `11`, inclusive. We'll look more closely at how to use the `..` operator to specify a range of values in the next section, when we discuss loops.

EXERCISE

Create a program that determines a user's eligibility to vote based on their age. Follow these steps to complete the exercise:

1. Prompt the user to enter their age as an integer. As we'll discuss in "Basic Input and Output" on page 37, this can be done as follows:

```
println("Enter your age:")
val age = readln()
```

2. Use a conditional `if...else` statement to check the user's age and provide an appropriate response:
    a. If the age is less than 18, display a message saying, "You are not eligible to vote yet."
    b. If the age is between 18 and 120 (inclusive), display a message saying, "You are eligible to vote."
    c. If the age is greater than 120, display a message saying, "Please enter a valid age."

(continued)

3. Run the program and test it with various ages to verify that it accurately determines eligibility to vote based on the age entered.
4. Repeat these steps using a `when` statement instead of `if...else`.

Loops

Loops are essential constructs in programming that allow you to repeat a block of code multiple times. Kotlin offers a few types of loops, each with its own use cases and advantages. In this section, we'll explore loops in Kotlin, including how to specify ranges for iteration.

for Loops

A `for` loop iterates through the elements in a collection, such as an array, list, or range. One common usage is to loop through a range of numbers, effectively creating a loop with a fixed number of repetitions. As we just saw with `when` statements, an inclusive range is specified in Kotlin using the `..` operator. Here's how this syntax fits with a `for` loop:

```
// inclusive range (1 to 4)
for (i in 1..4) {
    println("Current value of i is: $i")
}
```

The logic governing a `for` loop is given in parentheses immediately after the `for` keyword. In this case, we create the loop variable `i`, which takes on values `1` through `4` (inclusive). In the body of the loop, which is enclosed in braces, we print the current value of `i` using a string template. This `for` loop should produce the following output:

```
Current value of i is: 1
Current value of i is: 2
Current value of i is: 3
Current value of i is: 4
```

If you want to create a range that excludes the last value, use the `until` keyword instead of the `..` operator. Additionally, with either style of range, you can specify a *step* value to increment the loop variable by some amount other than `1`. Here we use an `until` range with a step size of `3`:

```
// exclusive range with step
for (i in 1 until 10 step 3) {
    println("Current value of i is: $i")
}
```

In this example, the loop variable `i` begins with a value of `1` and then increases by `3` each time through the loop, taking on the values `4` and `7` before the loop terminates. The output should look like this:

```
Current value of i is: 1
Current value of i is: 4
Current value of i is: 7
```

Notice that there's no output line for when `i` is `10`. This is because we used `until`, which excludes the upper bound of the range.

If you need a `for` loop to iterate in reverse order, create a range with the `downTo` keyword. This keyword allows you to specify a range where the loop variable starts at a higher value and counts down to a lower value. Like the `..` operator, `downTo` ranges are inclusive. Here's how it works:

```
for (i in 4 downTo 1) {
    println("Current value of i is: $i")
}
```

This `for` loop should output the following:

```
Current value of i is: 4
Current value of i is: 3
Current value of i is: 2
Current value of i is: 1
```

Thanks to the `downTo` keyword, the looping variable `i` counts down from `4` to `1`.

continue and break Statements

Kotlin makes it possible to disrupt the flow of a `for` loop using the `continue` and `break` keywords. Usually, these keywords work in conjunction with an `if` statement to interrupt the loop when some condition is met. The `continue` keyword halts the current iteration of the loop and immediately skips ahead to the next iteration. Here's an example:

```
for (i in 1..4) {
    if (i == 3) {
        // Skip the current iteration when i is 3.
        continue
    }
    println("Current value of i is: $i")
}
```

When the loop variable is set to `3`, the loop will continue by moving on to the next value of the loop variable. As a result, the `println()` function won't execute when `i` is `3`, yielding the following output:

```
Current value of i is: 1
Current value of i is: 2
Current value of i is: 4
```

The `break` keyword, by contrast, completely terminates a `for` loop, as shown here:

```
for (i in 1..4) {
    if (i == 3) {
        // Exit the loop when i is 3.
        break
    }
    println("Current value of i is: $i")
}
```

This loop will "break" when the loop variable equals `3`. As a result, the output should look like this:

```
Current value of i is: 1
Current value of i is: 2
```

Even though the range had a few values to go, the `break` keyword ended the loop early.

Nested and Named for Loops

It's common to nest one `for` loop inside another such that the entire inner `for` loop executes multiple times, as determined by the outer loop. Here's an example of a nested set of `for` loops that prints a pattern of asterisks in a square shape:

```
fun main() {
    val size = 4 // Change this value to adjust the size of the square.

    // nested for loops to print a square pattern of asterisks
    for (i in 1..size) {
        for (j in 1..size) {
          ❶ print("* ")
        }
      ❷ println() // Move to the next line after each row.
    }
}
```

In this example, the outer loop `for (i in 1..size)` iterates over the rows, and the inner loop `for (j in 1..size)` iterates over the columns within each row. The `print("* ")` statement prints an asterisk followed by a space for each element in a row. Unlike `println()`, the `print()` function ❶ doesn't add a newline character each time it's called, so each time through the outer loop, the inner loop will print a series of asterisks all together on one line. The empty `println()` statement after the inner loop ❷ moves the cursor to the next line to separate the rows. The output of this code, with `size` set to `4`, should look like this:

```
* * * *
* * * *
```

```
* * * *
* * * *
```

When you have nested loops, it can be helpful—though not strictly necessary—to assign each loop a name. The name comes before the `for` keyword and must be followed by an at sign (`@`). If the loops are named, you can explicitly apply keywords like `continue` and `break` to one loop or the other by adding the loop's name after the keyword. This gives you more control over when and how the nested loops can be disrupted. Here's an example:

```
loop1@ for (i in 1..5) {
    loop2@ for (j in 1..5) {
        print("$i,$j ")
      ❶ if (i == j) break@loop2
    }
    println()
}
```

We have two nested `for` loops, named `loop1` and `loop2`. The loop variables, `i` and `j`, both vary from `1` to `5` inclusive. Ignoring the conditional logic inside `loop2` for a moment, this nested loop would iterate for a total of 25 (5×5) times, printing each `(i,j)` pair, starting with `(1,1)` through `(1,5)` for the first time through `loop1`, then `(2,1)` through `(2,5)` for the second, and so on. The output would look like this, with the `println()` after each full cycle of the inner loop again ensuring that each group of five pairs is printed on its own line:

```
1,1 1,2 1,3 1,4 1,5
2,1 2,2 2,3 2,4 2,5
3,1 3,2 3,3 3,4 3,5
4,1 4,2 4,3 4,4 4,5
5,1 5,2 5,3 5,4 5,5
```

Now consider what the conditional logic in the inner loop does ❶. It applies the `break` keyword specifically to `loop2` (using the syntax `break@loop2`) if `i` and `j` are equal, halting the inner loop and returning to the outer loop for the call to `println()`. (Notice how the `@` symbol follows the loop name when the name is being assigned but precedes it when the name is being referenced.) This logic prints only the lower-left portion of the grid of `(i,j)` pairs, up to the main diagonal, where `i` and `j` are equal:

```
1,1
2,1 2,2
3,1 3,2 3,3
4,1 4,2 4,3 4,4
5,1 5,2 5,3 5,4 5,5
```

In truth, specifying that `break` applies to `loop2` isn't necessary here, since by default, keywords like `break` and `continue` apply to the innermost loop that contains them—in this case, `loop2`. Still, including the loop name helps clarify the intent behind the code. Furthermore, consider that another way to

achieve a similar result would be to replace `break@loop2` with `continue@loop1`, in which case referencing the loop name becomes necessary. I encourage you to try making this change as an exercise—you'll also need to think about what to do with the call to `println()` to keep the output nicely formatted.

while Loops

A `while` loop is best employed when you need to repetitively execute a code block but you don't know the exact number of iterations in advance. It keeps looping until a termination condition is satisfied. The condition is given in parentheses after the `while` keyword, before the start of the loop body. Here's an example:

```
var count = 0
while (count < 4) {
    println("Current value of count: $count")
    count++
}
```

We initialize the `count` variable to `0`, then create a `while` loop that will continue repeating as long as the condition `count < 4` remains `true`. The loop checks this condition *before* each repetition. Inside the loop, we print the current value of `count` and then increase its value by `1` using a unary `++` operator to get ready for the next iteration. This should produce the following output:

```
Current value of count: 0
Current value of count: 1
Current value of count: 2
Current value of count: 3
```

At the end of the fourth time through the loop, `count` increments from `3` to `4`. Then, when the loop gets ready to start its fifth iteration, it finds that the loop condition is no longer `true` and the loop terminates.

Another style of `while` loop uses the condition `while (true)`. Since `true` always evaluates to `true`, this theoretically sets up an infinite loop. The real conditional logic of halting the loop is instead moved inside the loop body itself. Here's the same `while` loop as before, implemented in this other style:

```
var count = 0
while (true) {
    println("Current value of count: $count")
    count++
    if (count >= 4) break
}
```

This time we use an `if` statement at the end of the loop body to trigger the `break` keyword and terminate the loop when `count` is greater than or equal to `4`. Without this conditional, the program would get stuck in an infinite loop, causing it to run indefinitely.

One further variation on the `while` loop is the `do...while` loop, which has the following syntax:

```
do {
    // code to be executed
} while (condition)
```

A `do...while` loop checks the stopping condition *after* each iteration, rather than before. This ensures that the loop will be executed at least once. By contrast, if the condition for a regular `while` loop is already `false` before the loop begins, it won't execute at all.

Just like `for` loops, you can nest multiple `while` loops and use them with the `continue` keyword as well as with `break`.

EXERCISES

Use `for` and `while` loops to do the following:

1. Write a `for` loop that prints a right-angled triangle pattern using asterisks (*). The number of rows in the triangle should be given by a variable called `n`. For example, if `n` is `5`, the output should be:

```
*
**
***
****
*****
```

2. Write a `while` loop that checks if a given string is a palindrome. A *palindrome* is a word or phrase that's the same when read forward or backward, such as *radar* or *madam*. The loop should compare the first and last characters of the string, and then move inward until either the characters don't match or the middle of the string is reached. The loop should print "Palindrome" if the string is a palindrome or "Not a palindrome" otherwise.

Functions

A *function* is a reusable block of code that performs a specific task or computation. Functions are a fundamental building block of any Kotlin program and are used to encapsulate logic, promote code reusability, and improve code organization. Kotlin's standard library comes with many built-in functions to simplify common programming tasks. One example is the `println()`

function we've been using throughout this chapter to output text to the console; other examples include various mathematical functions, some of which we'll explore shortly. For more specialized tasks particular to the application at hand, you'll have to create your own custom functions. We'll discuss how to do that as well.

Built-in Mathematical Functions

Throughout this book, we'll leverage a multitude of mathematical functions that come prepackaged in the standard Kotlin library. For example, you can effortlessly compute the square root of a number with the `sqrt()` function or raise a value to a specific power (exponent) using the `pow()` function. These functions are part of the `kotlin.math` package and must be *imported* at the start of a program using the `import` keyword. Here's a simple program that uses these two functions:

```
import kotlin.math.sqrt
import kotlin.math.pow

fun main() {
    val x = 100.0
    val y = 10.0

    val squareRoot = "The square root of $x is: ${sqrt(x)}"
    val toThePower2 = "$y raised to the 2nd power is: ${y.pow(2.0)}"

    println(squareRoot)
    println(toThePower2)
}
```

First, we import the two math functions we need. Then we use them to calculate the square root of `x` (`squareRoot`) and `y` to the second power (`toThePower2`) inside the string templates using the `${}` syntax discussed in "String Templates" on page 15. The code segment should produce the following output:

```
The square root of 100.0 is: 10.0
10.0 raised to the 2nd power is: 100.0
```

In some cases, you may need to use many built-in functions in the same module. Technically, it's possible to import the entire collection by including an asterisk (*) in the import statement. For example, `import kotlin.math.*` would import every function in the `kotlin.math` module. It's considered good practice to import only the functions you need, however. This approach helps avoid *namespace pollution*, where your code gets cluttered with unnecessary—or worse, conflicting—identifiers like variable and function names. Importing only what you need gives you more flexibility to name your own variables and functions without causing conflicts with the names of the built-in functions.

The remaining functions in the `kotlin.math` module handle trigonometry and other useful mathematical computations. For a comprehensive list of standard mathematical functions available and instructions on how to use them, search for `kotlin.math` in the official Kotlin documentation at *https://kotlinlang.org*.

Custom Functions

When you have some code that accomplishes a particular task in your application—especially a task that will be repeated—it's common to encapsulate that code in a custom function. This keeps your code well organized and efficient. Each custom function must be *declared*, or defined, before it can be used. Here's Kotlin's syntax for a function declaration:

```
fun functionName(parameter1: Type,
                 parameter2: Type, ...): ReturnType {
    // function body
}
```

The declaration begins with the `fun` keyword, followed by the function name and a set of parentheses. Inside the parentheses, you provide the names of the function's *parameters*, if any, along with their data types. These parameters serve as placeholders for values that the function expects to receive when it's called. They allow you to pass data into the function. A function can have many parameters of different data types (including parameters that are other functions) or no parameters at all, in which case the function name will be followed by a set of empty parentheses. When a function is called, specific values, called *arguments*, are provided for the parameters. Notice that function parameters are implicitly treated as read-only (immutable) variables within the function body. Using `val` or `var` on function parameters is not allowed.

After the parameter list, the function declaration continues with a colon (`:`) followed by the function's *return type*. This specifies the data type of the value the function will generate and provide to its caller. The built-in `sqrt()` function we discussed in "Built-in Mathematical Functions" on page 28 returns the square root of its parameter as either a `Double` or a `Float`, for example. If a function returns nothing, you can omit the return type. It will be assumed to be `Unit`, akin to the `void` type in other languages. A function would return nothing if, for example, its purpose is to print output to the console, make changes to global variables, make changes to elements of an array or object supplied to the function as an argument, or call other functions.

Taken together, the function name and the parameter names and types define the *signature* of the function. The compiler uses function signatures to determine which function to use when multiple functions have the same name but different parameters (a practice called *function overloading* that we'll discuss shortly). The return type isn't part of the function signature, but it's an important part of the function declaration nonetheless.

Once the function signature and the return type are specified (when applicable), all that remains is to declare the function *body*, which is enclosed in braces. This is the block of code that will be executed when the function is called. It may include additional variable declarations, conditional statements, loops, and expressions—anything necessary for the function to do its work.

Let's now examine a real function that performs a simple task: adding two integers together and returning the result. Here's the function's declaration:

```
fun add(x: Int, y: Int): Int {
    return x + y
}
```

We declare a function named `add()` that takes two parameters, `x` and `y`, both of type `Int`, and returns a value also of type `Int`. The function body calculates `x + y` and uses the `return` keyword to deliver that result to the function caller. Note that if the return type implied by the value being returned is different from the function's declared return type, the compiler will generate an error. In this case, since both `x` and `y` are integers, `x + y` will be an integer as well.

With the `add()` function declared, we can call it from `main()` as follows:

```
fun main() {
    // Declare the variables.
    val a = 3
    val b = 6

    // Call the function to add two integers.
    val sum = add(a, b)
    println("The sum of $a and $b is $sum.")
}
```

We declare the `a` and `b` variables and initialize them to `3` and `6`, respectively. Next, we declare `sum` and assign it the value returned by `add()`. Running this code should yield the following output:

```
The sum of 3 and 6 is 9.
```

Kotlin enforces strong typing, so the compiler will flag errors for mismatches between declared parameter types and the types of the arguments passed to the function. The number of arguments should also match the number of parameters, unless some parameters are assigned default values.

SINGLE-EXPRESSION FUNCTIONS

In Kotlin, when the compiler can infer a function's return type and the function body consists of a single expression, you can declare the function using a

concise syntax known as a *single-expression function*. For example, here's how to declare a simple `add()` function using this syntax:

```
fun add(x: Int, y: Int) = (x + y)
```

The function declaration is compressed to one line, without the need for braces around the function body or a declared return type. In this case, the return type is inferred to be `Int`, and the function body `(x + y)` is a single expression that directly represents the result of the function—with no need for the `return` keyword. This shorthand is particularly useful for simple functions with a short, one-line expression.

Providing Default Parameter Values

If a parameter has the same value most of the time a function is called, it can be given a default, preset value when the function is declared. This way, you need to provide an argument for that parameter only when you want it to be something other than the default. The parameters with default values should be listed last in the function declaration. Here's an example of how to declare a default parameter value:

```
fun greet(name: String, greeting: String = "Hello") {
    println("$greeting $name!")
}
```

The `greet()` function takes two parameters, `name` and `greeting`, and combines them to print out a customized greeting. A value for `name` will need to be provided each time the function is called, but if no value for `greeting` is specified, the function will use the default value of `"Hello"`. This default value is set in the parameter list, after the parameter's data type.

If we call the function with `greet("Nathan")`, supplying an argument only for the `name` parameter, it should result in the following output, including the default value for `greeting`:

```
Hello Nathan!
```

Consider how you would call this function if you wanted it to instead print `"Good morning, Nathan!"` as a message.

Using Named Arguments

When a function has many parameters, keeping track of their order and type can be a hassle. Using *named arguments* comes in handy in such situations. This is a style of function call where you include the parameter names along with the desired argument values. With the names included, the arguments can be given in any order.

Say you've declared this function with four parameters:

```
fun printPersonInfo(firstName: String, lastName: String,
                    age: Int, gender: String) {
    println("Person info: $firstName $lastName, " +
            "Age: $age, Gender: $gender")
}
```

Calling the function using named arguments relieves you from the burden of remembering the order in which the parameters were declared:

```
printPersonInfo(lastName = "Keller", firstName = "Jeffrey",
                age = 40, gender = "Male")
```

Here each argument is given in the form *parameterName* = *value*. This function call will work just fine, even though the parameters are out of order, thanks to the use of named arguments. This is allowed as long as you provide names for all arguments or the compiler is able to figure out the order of the arguments.

Overloading a Function

Function overloading in Kotlin allows you to define multiple functions with the same name in the same scope but with different parameter lists. Perhaps they have a different number of parameters, or the parameters are of different data types. Here's an example of function overloading:

```
// function to add two integers
fun add(a: Int, b: Int): Int {
    return a + b
}

// function to add two doubles
fun add(a: Double, b: Double): Double {
    return a + b
}
```

We declare two functions called `add()` with different parameter lists; one adds two integers, and the other adds two floating-point numbers. When the `add()` function is called, the compiler will determine which version of the function to invoke by comparing the argument types to the declared function signatures. This is how you would call both functions:

```
val result1 = add(2, 3)
val result2 = add(40.5, 23.9)
```

With function overloading, you can use the same function name for operations that conceptually do the same thing (in this case, addition) but with different parameter types. This makes your code more readable, intuitive, and error resistant. In this example, we've anticipated that we may

want to add floating-point numbers as well as integers. Overloading the function gives us the flexibility to do either without triggering an error.

Referencing a Function Without Calling It

In Kotlin, you can use the member reference operator (`::`) to refer to a function by its name without actually invoking it. This is helpful in many situations, such as when you need to assign a function reference to a variable. Say you have two functions and you're trying to decide which to use in your code. Perhaps they're both ways of accomplishing the same task and you want to see which is more efficient, or perhaps they implement two different operations that would be appropriate in different situations. Rather than rewriting all the code to call one function or the other, you can assign to a variable a reference to the appropriate function and then call the function (whichever one you choose) via the variable, minimizing the changes to the code. Here's an example illustrating how this works:

```
fun add(x: Double, y: Double): Double {
    return x + y
}

fun multiply(x: Double, y: Double): Double {
    return x * y
}

// Change this condition to use add() or multiply().
val useAdd = true

fun main() {
    // Declare a function variable using member reference.
  ❶ val selectedFunction = if (useAdd) ::add else ::multiply

    val x = 3.0
    val y = 4.0

    // Calculate the value of the selected function.
  ❷ val result = selectedFunction(x, y)

    // Print the result.
    println("Result: $result")
}
```

We first declare two functions: `add()` calculates the sum of `x` and `y`, and `multiply()` calculates the product of `x` and `y`. We'll want to use only one of these functions in `main()`. To control which one, we declare a Boolean variable `useAdd` and set it to `true`. Inside the `main()` function, we then create another variable named `selectedFunction` and use *conditional expression syntax* to set its value to one of the functions ❶. This syntax uses an `if...else` structure to return a value and assign it to a variable based on a condition—in this case, the state of the `useAdd` variable. If `useAdd` is `true`, `selectedFunction` is assigned a reference to `add()`; otherwise, it references `multiply()`. Notice how we precede each function name with the `::` operator and don't include

parentheses after the function name, since we're referencing rather than invoking the functions. With the `selectedFunction` variable holding a reference to a function, we can now invoke that function by calling `selectedFunction()` rather than by calling `add()` or `multiply()` directly ❷. We store the returned value in the `result` variable and print it to the console.

Try changing the value of `useAdd` from `true` to `false` to switch between using `add()` and `multiply()`. Then consider how convenient this solution is for toggling between two functions, especially if you need to use the function in multiple places in the code. Instead of updating the function name at each usage, we can simply change the value of `useAdd` and rely on the `selectedFunction` variable to stand in for whichever function we want.

The `::` operator is particularly useful when you need to pass a function reference as an argument to another function. Here's an example:

```
fun printMessage(message: String) {
    println(message)
}

fun applyFunction(function: (String) -> Unit, input: String) {
    function(input)
}

fun main() {
    // using :: to reference the printMessage function
    applyFunction(::printMessage, "Hello, Kotlin!")
}
```

We define a function called `printMessage()` that takes a string parameter and prints it to the console. We also define another function called `applyFunction()` with two parameters: `function`, which will hold a reference to a function, and `input`, a string. The `function` parameter's type needs to match the parameter and return types of the function it will reference; `(String) -> Unit` indicates that the function will take a string as a parameter and return nothing. In the body of `applyFunction()`, we call whatever function was passed in, using the `input` string as its argument.

In `main()`, we create a reference to `printMessage()` using `::` and pass it to `applyFunction()` along with the string `"Hello, Kotlin!"` as the second argument. This will effectively make `applyFunction()` call `printMessage()` with the given string and print it to the console. The output should be:

```
Hello, Kotlin!
```

Of course, you can do more with the `::` operator than we've discussed here. I encourage you to consult the official Kotlin documentation to explore the operator's other use cases.

Scope Functions

In Kotlin, *scope functions* are a set of built-in functions used to manage the scope of variables, access properties of objects, and execute a block of code

in a specific context. The scope functions in Kotlin are `run`, `with`, `let`, `also`, and `apply`. They're often used to simplify and improve the readability of code, especially when working with objects or managing the flow of operations. Here is a simple example of how to use the `run` function:

```
val result = run {
    val x = 10
    val y = 20
    x + y // The value of this final expression is returned.
}
println("Result: $result") // prints "Result: 30"
```

We begin by declaring a variable called `result`. Its value is determined by the last expression within a code block executed using the `run` scope function. Inside this block, we define and assign values to two integers, and the final expression computes their sum. The resulting value from this final expression is returned by the code block and assigned to the `result` variable. Finally, we use `println()` to print the value of `result`, which will display `Result: 30`.

Lambda Expressions

Lambda expressions, often referred to simply as *lambdas*, are a way to define and pass around blocks of function-like code in a flexible and concise manner. They're essentially anonymous functions, allowing you to create functions on the fly without giving them a name. Lambda expressions are a fundamental part of functional programming, a programming style or paradigm that uses functions as the main building blocks. They make it easier to work with *higher-order functions*, functions that can take functions as arguments, return functions as values, or both. Higher-order functions can help create reusable and modular code that can be customized with different functions.

Here's an example of a simple lambda that takes in a name and generates a greeting:

```
val greet : (String) -> String = { name -> "Hello, $name!" }
```

The lambda itself is the part of the code enclosed in braces: `{ name -> "Hello, $name!" }`. It consists of input parameters (in this case, just one, `name`) and a body (`"Hello, $name!"`), separated by an arrow (`->`) symbol. Think of this arrow as passing the parameters along to the lambda body. The `return` keyword is implied in the lambda body; if the body consists of a single expression, that expression will automatically be returned.

In this example, we assign the lambda to the `greet` variable. The `(String) -> String` before the assignment operator specifies the lambda's parameter and return types, again using the `->` symbol to separate the two. We can also incorporate these type declarations into the lambda itself, in which case we'd write the entire expression as:

```
val greet = { name: String -> "Hello, $name!" }
```

Here we specify that the `name` parameter is of the `String` type from within the braces. With this syntax, the return type, like the `return` keyword itself, is implied.

Whichever syntax we use, we now have a function that returns a string greeting inside the `greet` variable. We can therefore call the lambda via this variable, just like we would call an ordinary function:

```
println(greet("Alice")) // output: Hello, Alice!
println(greet("Bob"))   // output: Hello, Bob!
```

Lambdas are often used for quick manipulation of data, such as adding two numbers or calculating the square of a number:

```
val sum: (Int, Int) -> Int = { a, b -> a + b }
println(sum(3, 4)) // output: 7

val square: (Int) -> Int = { it * it }
println(square(5)) // output: 25
```

Lambda parameters can be explicitly typed or inferred. For simple lambdas, Kotlin can infer types automatically. Also, if a lambda has a single parameter, you can omit the parameter declaration and use the implicit `it` keyword directly in the lambda's body as a stand-in for the parameter. This is what we've done for the `square` lambda: `{ it * it }` indicates that the lambda will take a single unnamed parameter and multiply it by itself.

Lambdas can span multiple lines to carry out more complex tasks and can be nested like `for` loops. We'll use multiline, nested lambdas in the next section in relation to copying the content of a file.

EXERCISES

Use the following exercises to practice creating lambda expressions:

1. Write a lambda expression that takes two integers as parameters and returns the larger one. If the integers are equal, return either one. Assign the lambda expression to a variable called `max`, and use it to find the maximum of some pairs of integers.
2. Write a lambda expression that takes a string as a parameter and returns a new string that's the reverse of the original string (without using Kotlin's built-in `reversed()` method). For example, if the input is **`Kotlin`**, the output should be `niltoK`. Hint: Inside the body of the lambda, define a variable called `reversed` as an empty string, then use a `for` loop to iterate over the characters of the input string in reverse order, appending them to `reversed`. When done, use `reversed` as the last line inside the lambda block to return the reversed string.

Basic Input and Output

Most commercial software these days has a graphical user interface (GUI) so that you can easily interact with it. You can create a GUI for your Kotlin app using third-party tools such as JavaFX or Jetpack Compose, but we won't cover that in this book. Instead, in this section we'll focus on how to work with text-based input and output in Kotlin. This helps you quickly test and debug your code. Text-based output is also useful for tasks like scripting, keeping logs, and watching what's happening on networks of computers and devices, which is what system administrators often do.

Console-Based Input and Output

To get user input from the console in Kotlin, use the `readln()` function. It returns whatever text the user enters into the console or an empty string if the user simply presses the ENTER key. Here's an example:

```
println("Enter some text:")
val userInput = readln()
println("You entered: $userInput")
```

Here, the user input (a line of text) is read into the `userInput` variable and then displayed as output to the console using the familiar `println()` function. If the user just presses ENTER, then `readln()` will return an empty string, and the program will end normally after printing the following:

```
You entered:
```

When reading input from the console in Kotlin, keep in mind that all input is initially treated as text, resulting in a `String` data type. If you require a different data type, you must perform type casting, assuming the types are compatible. It's also helpful to inform users about the expected input type by using prompts such as `Enter your name` or `Enter an integer`.

Even with a clear prompt, though, you shouldn't automatically assume that the input will be valid. The user may input incorrect characters that can't be successfully type-cast into the desired format. To prevent the program from crashing due to potential errors, it's essential to implement error-handling mechanisms when receiving input from users. This extra step ensures robustness and a smoother user experience. Here's a complete example of a fail-safe method for reading an integer from the console:

```
fun main() {
    while (true) {
        print("Enter an integer: ")
        val num = readln()

        // Validate using a try...catch block.
        try {
            val intValue = num.toInt()
            println("You entered: $intValue")
```

```
            break // Stop the loop on success.
        } catch (e: NumberFormatException) {
            println("Invalid input. Try again.")
        }
    }
}
```

We begin by creating a `while` loop that continues to run until a valid input is provided. Next, we read the user input as a string and assign that to a variable called `num`. We check the validity of the input inside a `try...catch` construct for graceful error handling. This construct consists of two code blocks: a `try` block containing the code you'd like to run and a `catch` block containing an alternative code path or fallback option in case an error occurs during the `try` block. The `catch` block prevents the entire program from crashing abruptly from an error. This mechanism helps with debugging during testing and improves the overall user experience in commercial applications.

In this case, the `try` block attempts to convert the user's input, stored in the variable `num`, into an integer using `toInt()`. If this conversion is successful, a message containing the integer value is printed.

However, if the conversion fails and a `NumberFormatException` is raised, the `catch` block is activated, printing an error message before the next iteration of the `while` loop begins. Notice the `(e: NumberFormatException)` immediately after the `catch` keyword. This specifies the particular type of exception that the `catch` block is designed to handle.

A SNEAK PEEK AT OBJECT-ORIENTED PROGRAMMING

"Simple File Operations" on page 39 references terms and concepts that will be discussed in detail in Chapter 2 in relation to classes and object-oriented programming. For now, you can think of *classes*, such as the `File` and `Scanner` classes discussed in the section, as blocks of code that have associated *properties* (variables) and *methods* (functions). An *object* is a particular instance of a class, created using the blueprint that the class provides. Likewise, Kotlin's built-in data types are essentially classes, and instances of those data types are objects of those classes. For example, a string such as `"Hello, world!"` is an instance of the `String` class. This means it automatically has properties such as `length` (to report the number of characters in the string) and methods such as `plus()` (for concatenation with another string).

Simple File Operations

Kotlin provides simple and effective ways to read and write files, a feature that's very helpful when you want to retrieve previously saved data or save the data from the current run of a program. For this functionality, Kotlin relies on the Java standard library. For example, here's how to read data from a file using Java's `File` and `Scanner` classes:

```
import java.io.File
import java.util.Scanner

fun main() {
    // Replace the path below with the path to your file.
    val inputFile = "inputfile.txt"

    try {
      ❶ val file = File(inputFile)
      ❷ val sc = Scanner(file)
        while (sc.hasNextLine()) {
            val line = sc.nextLine()
            println(line)
        }
    } catch (e: Exception) {
        println("An error occurred: ${e.message}")
    }
}
```

The example shows how to read a text file line by line. After importing the `File` and `Scanner` classes, we assign a string containing the full pathname of the input file to the `inputFile` variable. We then use this variable to create the `File` object called `file` ❶, which in turn is used to create a `Scanner` object called `sc` ❷ that gives us access to the file's contents. Then, in a `while` loop, we read the content of the file one line at a time using the `Scanner` object's `nextLine()` method and print the result to the console. The loop continues until we reach the end of the file, indicated when the `Scanner` object's `hasNextLine()` method returns `false`. We place all this code in a `try` block and use a corresponding `catch` block to handle any errors that arise while trying to access the file—for example, if the filename or filepath is wrong. The `(e: Exception)` indicates the `catch` block can handle any kind of exception, unlike the earlier `catch` block that was designed specifically for exceptions of type `NumberFormatException`. In this case, the `catch` block prints the default error message associated with the exception, accessible as `e.message`.

My test file at location *inputfile.txt* contained a limerick, and the program reproduced it on the console line by line:

```
There once was a man named Bob
Who loved to eat corn on the cob
He ate so much corn
That he grew a horn
And now he is known as Corn-Bob
```

To read from *and* write to a file, you can't use the `Scanner` class, since it doesn't support writing output. Instead, you can use the `appendText()` method of the `File` class. Here's a simple example:

```
import java.io.File

fun main() {
    // Replace the file locations as needed.
  ❶ val inputFile = File("inputfile.txt")
  ❷ val outputFile = File("outputfile.txt")

    // Read all lines from the input file.
  ❸ val lines = inputFile.readLines()

    // Write all lines to the output file.
  ❹ for (line in lines) {
        outputFile.appendText("$line\n")
    }
    println("Copied inputfile.txt to outputfile.txt")
}
```

This Kotlin code reads all the lines from an input file (*inputfile.txt*) and writes them to an output file (*outputfile.txt*). The input file is represented using one `File` object ❶ and the output file with another ❷. We use the `File` class's `readLines()` method to read all the lines from the input file and return them as a list of strings ❸. (A *list* in Kotlin is a collection of items; in this case it's a collection of strings, one for each line of the file. We'll discuss lists in detail in Chapter 2.) We store this list of strings in the `lines` variable. We then use a `for` loop to iterate over the `lines` list, with the looping variable `line` standing for one line at a time ❹. For each line, we use the `appendText()` method to add the line to the output file. We also append a newline character (`\n`) to the end of each line to ensure it's written on its own line of the output file. We conclude the code by printing a message to the console, indicating that the input file has been copied to the output file.

Notice that we didn't use the `try...catch` block in this example, as the goal was to quickly show how to write data to a file. In real-world applications, file operations may have to be wrapped in a `try...catch` block to handle potential exceptions or errors, depending on your specific requirements.

You can use many other techniques for reading from and writing to files in Java and Kotlin. See the official Kotlin documentation for information about other methods.

Project 1: Build a Console-Based Calculator

Now that we've explored some of the basic features of Kotlin, let's put that knowledge to work in a real project. We'll develop an interactive, console-based calculator application. The application will take a pair of valid numbers as input, ask the reader to choose an arithmetic operation (addition, subtraction, multiplication, or division), and then display the result of that

operation in the console. We'll also program the application to show helpful error messages when required.

At the start of any coding project, it's crucial to begin by creating a mental map of the application's structure. This involves identifying the necessary variables and data structures, as well as pinpointing the essential functionalities that the program should include. Once these components are identified, we can move forward by generating a list of the key components that need to be implemented, followed by the actual coding phase.

For more complex projects, it can also be beneficial to create a flowchart visualizing the application logic or to develop detailed pseudocode offering step-by-step instructions for coding the entire project. However, given the relatively straightforward nature of the calculator project, we'll start by listing its key components:

1. Input collection: We'll gather user input for two numbers and ensure the inputs are valid.
2. Operation selection: The user will choose addition, subtraction, multiplication, or division.
3. Calculation: The selected operation will be applied to the input numbers.
4. Result display: The calculated result will be presented to the user via the console.
5. Error messages: Throughout steps 1 through 3, we'll display helpful error messages for invalid inputs, such as nonnumerical characters for the number inputs or an unrecognized mathematical operation.

We'll use these five key components to guide the development process as we start coding our first mini project.

The Code

We'll discuss the code from the top down, starting with the `main()` function, which coordinates the program's actions through a series of helper functions. This approach allows us to align the code with the key components we outlined.

```
import kotlin.system.exitProcess

fun main() {
    println("***  Console Calculator  ***")

    // step 1: input collection
    println("\nEnter two numbers:\n")
    val number1 = readDoubleInput("Number 1: ")
    val number2 = readDoubleInput("Number 2: ")

    // step 2: operation selection
    showChoices()
    val operation = getArithmeticOperation()
```

```
    // step 3: calculation
    val result = performCalculation(number1, number2, operation)

    // step 4: result display
    println("\nResult:\n" +
            "$number1 $operation $number2 = $result")
}
```

We begin by importing the `exitProcess()` function from Kotlin's standard library. We'll use this method to exit the program if the user provides invalid input or if the code comes across an invalid operation type (for example, division by zero).

The `main()` function itself is divided into four clear steps, each related to one of the key project functions. In the first step, we ask the user to provide two numbers, which we store in the `number1` and `number2` variables. To manage and verify the input, we use the `readDoubleInput()` function, declared here:

```
fun readDoubleInput(prompt: String): Double {
    print(prompt)
    val num = readln()

    // Check input validity.
    try {
        return num.toDouble()
    } catch (e: Exception) {
        println("Error reading input: ${e.message}")
        exitProcess(1) // Exit with error code 1.
    }
}
```

This function takes a single argument, a string serving as a prompt for user input in the console. It returns a numeric value (of type `Double`) if the user provides a valid input. The function displays the prompt using `print()` rather than `println()` so that the user's response will go on the same line. Then it reads the user's input with `readln()` and processes the string input inside a `try...catch` block. When the string is successfully converted into a numeric value of type `Double`, its value is returned. Otherwise, we enter the catch part of the block, where an error message is printed, and the program exits with an error code of 1.

NOTE *When using the `exitProcess()` function, any integer can be used as an error code. However, it's important to first decide on a scheme for error codes based on the different types of errors that the application might generate. This will allow you to quickly locate the source of the error. In more complex projects, creating and maintaining a log or wiki of error codes is recommended.*

After the `main()` function receives two valid numeric values (`number1` and `number2`), we move on to step 2, choosing the mathematical operation. For that, we first call the `showChoices()` function, which offers a list of arithmetic operations to the user. This function is simply made up of a number of `println()` calls:

```
fun showChoices() {
    println("\nOperation Options:")
    println("1. Addition (+)")
    println("2. Subtraction (-)")
    println("3. Multiplication (*)")
    println("4. Division (/)")
}
```

Next, we use the `getArithmeticOperation()` function to take in a valid operation from the user. The result is assigned to the string variable `operation` in `main()`. Here's what the `getArithmeticOperation()` function looks like:

```
fun getArithmeticOperation(): String {
    print("\nEnter an arithmetic operation (+, -, *, /): ")
    val operation = readln()

    if(!"+-*/".contains(operation, true)){
        println("\nInvalid operation. Exiting.")
        exitProcess(2) // Exit with error code 2.
    }
    return operation
}
```

Within this function, the user is prompted to select one of the four valid arithmetic operators. This choice is captured using the `readln()` method. Subsequently, we use an `if` statement to verify whether the user input is valid. Specifically, if the string `"+-*/"` does not include the user's input, an error message is printed, indicating that an invalid operator has been provided. The program then terminates with an error code of 2.

Back in `main()`, we can now move to step 3 and call the `performCalculation()` function to carry out the selected arithmetic operation and return the result. Here's that function's declaration:

```
fun performCalculation(number1: Double, number2: Double,
                       operation: String): Double {
    return when (operation) {
        "+" -> number1 + number2
        "-" -> number1 - number2
        "*" -> number1 * number2
        "/" -> if (number2 != 0.0) number1 / number2
               else {
                   println("\nDivision by zero is not allowed. Exiting.")
                   exitProcess(3)
               }
      ❶ else -> {
            println("\nUnexpected error encountered. Exiting.")
            exitProcess(4)
        }
    }
}
```

This function takes in the two input numbers and the string containing the desired operation as parameters and returns the result of the calculation as a number of type `Double`. It employs a `when` statement to execute the desired calculation based on the value of `operation`. By this stage, both the numbers and the operation type have been validated. However, one potential source of error remains: division by zero. We address this with an `if... else` block in the `"/"` case of the `when` statement, which prints an error message and exits the program if `number2` is `0.0`.

Notice that we also include an `else` clause for the overall `when` statement ❶, even though no further errors should remain at this point (hence the "Unexpected error encountered" message). Having this clause as a fallback is good practice in case unexpected issues or compiler bugs lead to unpredictable errors.

With the calculation performed, our `main()` function moves on to step 4 and displays the result using a string template. Looking back at `main()`, notice how our use of custom functions to encapsulate the various individual tasks of the program has kept the `main()` function itself tidy and easy to read. In this way, functions help us maintain well-organized applications.

The Result

Here's a sample run of the program to multiply two numbers, 37 and 9. The user input is shown in bold.

```
***  Console Calculator  ***

Enter two numbers:

Number 1: 37
Number 2: 9

Operation Options:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)

Enter an arithmetic operation (+, -, *, /): *

Result:
37.0 * 9.0 = 333.0
```

Notice that even though the inputs are integers, they get converted to the `Double` type before conducting the multiplication. That is, `37` becomes `37.0` and `9` becomes `9.0`, as shown at the end of the output under `Result`. Feel free to experiment with invalid numbers or operators and observe how the program behaves. We can learn a lot from our mistakes!

EXERCISE

In Project 1, you created a basic calculator. Now it's time to level up your calculator's capabilities. In this exercise, you'll add more advanced mathematical operations and scientific functions, making your calculator a more versatile tool. Here are some tasks to enhance the program:

1. Implement the modulo (%) operation, allowing users to find the remainder when one integer is divided by another.
2. Add the ability to raise an input number x to a given power n (that is, x^n) and display the result.
3. Allow the calculation of the square root of a positive number.
4. Incorporate the ability to calculate and display the sine (sin), cosine (cos), and tangent (tan) of a given angle x. Users should provide the angle in degrees.
5. Take in a value x and calculate e^x (the exponential function).
6. Enable the natural logarithm ($\log_e$) function, allowing users to find the natural logarithm of x.

Hint: Notice that some of these operations require two input numbers, while others require just one. You should therefore prompt the user for the operation they want to perform first, and then request the necessary numeric inputs. You'll need mathematical functions from the `kotlin.math` module to implement many of these operations.

Summary

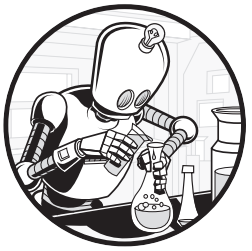
In this chapter, we covered an array of fundamental elements of the Kotlin language. You learned how to use comments to enhance code clarity, variables to store and manage data, and operators to manipulate that data. You explored flow control structures like conditionals and loops to dictate your program's behavior, and functions to encapsulate and reuse code effectively. With lambda expressions, you saw a way to write and use functions on the fly and got a glimpse of the functional programming style. You also practiced receiving input and providing output, for both console- and file-based interactions. To conclude the chapter, you gained hands-on practice bringing these elements together through a project implementing a console-based calculator.

Resource

Kotlin. "Kotlin Docs." (The official Kotlin documentation.) Accessed June 15, 2024. *https://kotlinlang.org/docs/home.html*.

2

ARRAYS, COLLECTIONS, AND CLASSES



In this chapter, we'll continue to explore the fundamentals of the Kotlin language by learning how to store and manipulate data in various ways. We'll move beyond the simple data types of Chapter 1 and explore data structures that can hold multiple values in a single unit. We'll start with arrays, then move on to collections such as lists, sets, and maps, which provide more functionality and flexibility for working with data. Then we'll learn how to create custom containers in the form of classes. We'll investigate various kinds of classes, including regular, data, abstract, and enum classes.

Classes are the foundation of the object-oriented style of programming, allowing us to model and manipulate data by defining our own data

types with specific properties and behaviors. As we discuss classes, we'll also demystify other object-oriented concepts, such as encapsulation, inheritance, polymorphism, and interfaces. At the end of the chapter, we'll synthesize the topics we've covered into a project where we create a basic task manager app to help you track and organize your daily tasks.

Arrays

A Kotlin *array* is a collection of elements in a contiguous block of memory. The number of elements in the array is determined when the array is created; therefore, it can't be changed, meaning you can't add an extra element to an array once it is created. However, the values of an array's elements are mutable, so they can be modified as needed. An array can hold elements of any type, including user-created types, as long as all elements within the same array have the same type or are derived from a common parent type (supertype).

Each array element has an index that allows you to access it individually. By default, the first element of an array will have an index of `0`, the second element an index of `1`, and so on. The index of an array's last element is therefore always one less than the array's size. For example, if we create an array of size `10` (that is, containing 10 elements), the index of the last element would be `9`. To access an array element, place its index in square brackets after the name of the array.

Here we create two different arrays, one comprising integers and one comprising string values, and print the last element of the first array and the first element of the second:

```
val arrInt = arrayOf(10, 20, 30, 40)
println(arrInt[3]) // output: 40
val arrString = arrayOf("one", "two", "three", "four", "five")
println(arrString[0]) // output: one
```

We create each array with the `arrayOf()` function, passing the initial values for the array's elements as arguments. Notice that we don't have to specify the data type (`Int` or `String`) explicitly; the compiler can infer the array type from the values provided.

In Kotlin, we can create an array with elements of different subtypes as long as they're all derived from the same supertype. For example, `Any` is a Kotlin supertype that encompasses all other data types, such as `Int`, `String`, and `Boolean`. Therefore, if we create an array of type `Any`, we're free to mix these data types, as shown here:

```
val myArray: Array<Any> = arrayOf(1, "bye", false)
```

We use the `Array<Any>` type declaration to indicate that `myArray` can contain elements of any data type. Indeed, it contains an integer (`1`), a string (`"bye"`), and a Boolean (`false`). Since all these types are derived from the common supertype `Any`, they can be stored in the same array. We could have skipped declaring the array type using `Array<Any>` in this case; Kotlin is smart

enough to figure it out on its own. However, if you're creating an array of elements with a user-defined type, it may be a good idea to declare the type explicitly to remind yourself that you're working with elements of a nonstandard type. For example, if you have a custom `Person` class and want to create an array of objects of this class, you could declare the type as follows:

```
val people: Array<Person> = arrayOf(...)
```

This indicates that the elements in the `people` array will all be `Person` objects.

> **WHAT IS AN OBJECT?**
>
> An *object* is a fundamental concept in object-oriented programming (OOP). It is an instance of a class, which is a blueprint for creating objects. Each object can have its own set of properties and functions, which are defined by the class. In general, objects are used to organize code into reusable, modular components. We'll cover classes and objects in detail later in this chapter.

Primitive Arrays

Kotlin provides specialized array types for select data types including `Byte`, `Short`, `Int`, `Long`, `Char`, `Float`, `Double`, and `Boolean`. For example, we can use `IntArray` for integers, `DoubleArray` for floating-point values, and `CharArray` for individual characters. Primitive arrays are more memory efficient than their nonprimitive counterparts, making them a good choice for performance-critical operations. The syntax for creating primitive arrays is similar to that of regular arrays, with a specialized function, equivalent to `arrayOf()`, for each type of primitive array:

```
val intArray = intArrayOf(1, 2, 3, 4, 5)
val doubleArray = doubleArrayOf(1.0, 2.0, 3.0, 4.0, 5.0)
val charArray = charArrayOf('a', 'b', 'c', 'd', 'e')
val booleanArray = booleanArrayOf(true, false, true, false)
```

No primitive array for the `String` type exists in Kotlin, since `String` is a reference type, not a primitive type, and is handled differently by the Java Virtual Machine (JVM) compared to primitive types during runtime. Because of this, creating a special primitive string array wouldn't provide any significant memory or performance advantage in the way something like `intArray` or `booleanArray` does.

The Array Constructor

Another way to create an array is with the `Array` constructor. As we'll discuss in more detail later in the chapter when we look at classes, a *constructor* is a

function for creating an object of a particular class—in this case, the `Array` class. In Kotlin, you can use the `Array` constructor to create an array of a given size and set its elements to values determined using a lambda expression or function. Once the array elements are initialized, you can access them later and update their values as required. Using a lambda expression or a function to initialize a large array can be more efficient than hardcoding the values as arguments to the `arrayOf()` function or reading them from an input file.

The `Array` constructor takes two parameters: the size of the array and a function that returns the initial value for each array element based on its index. For example:

```
val num = Array(4, {i -> i * 2})
```

Here, we call the `Array` constructor to create an array of size `4` and initialize its elements using a lambda expression. The lambda takes the index of each element (`i`) and doubles it to set the element's value. The result is an array of the integer values `0`, `2`, `4`, and `6`.

Array Operations

Arrays in Kotlin offer a variety of methods that can help you access and manipulate their elements. Table 2-1 summarizes some of the commonly used methods for array manipulation.

Table 2-1: Commonly Used Array Methods

Operation	Description	Example
Access	Retrieve an element by its index.	`val element = array[index]`
Update	Modify an element at a specific index.	`array[index] = newValue`
Size	Get the number of elements in the array.	`val size = array.size`
Iterate	Loop through each element in the array.	`for (element in array) {/* ... */ }`
Search/find	Check if an element exists in the array (`true` or `false`).	`val found = array .contains(element)`
Slice	Extract a portion of the array.	`val subArray = array .slice(startIndex..endIndex)`
Sort	Arrange elements in ascending or descending order.	`array.sort()` or `array.sortDescending()`
Filter	Create a new array with elements that meet a condition.	`val filteredArray = array.filter {/* condition */ }`
Map/transform	Apply a function to each element and create a new array with the results.	`val mappedArray = array.map {/* transformation */ }`
Join	Combine elements into a single string with a delimiter.	`val joinedString = array .joinToString(", ")`

Notice that when we apply methods such as `filter` and `map` to a data container such as an array, and the method name is followed by a lambda expression, we don't have to include parentheses after the method name like we typically would when calling a function. I encourage you to try out these operations by creating and manipulating different types of arrays.

Multidimensional Arrays

A *multidimensional array* is an array whose elements are themselves arrays. Nested arrays are widely used in scientific and numerical computation. For instance, a two-dimensional array can represent a grid of pixels in an image or the coordinates of locations on a map. Similarly, a three-dimensional array can be used to track the location and movements of an object in space, such as in three-dimensional gaming or for real-world objects like satellites.

In Kotlin, you can create multidimensional arrays using the built-in array creation functions. Here's how to create a two-dimensional array using the `Array` constructor:

```
val numRow = 3
val numCol = 4
// Create a (3x4) array.
val twoDimArray = Array(numRow) { Array(numCol) { 0 } }
// Access and modify an element using its indices.
twoDimArray[2][3] = 99
```

In this example, we use the `Array` constructor to create a two-dimensional array with `3` rows and `4` columns and initialize all 12 of its elements to `0`. We then replace the value of the last element, which has a row index of `2` and a column index of `3`, with a new value (`99`). Notice that we use separate sets of square brackets for the two indices. Creating and manipulating a three-dimensional array follows the same pattern:

```
// Create a 3D array.
val threeDimArray = Array(2) { Array(3) { Array(4) { "" } } }
// Access and modify an element using its indices.
threeDimArray[1][2][3] = "Hello, world!"
```

In this example, we first create an array of dimensions 2×3×4 and initialize its elements to empty strings. As before, we access and change the last element in the array using its dimensional indices.

We can also use nested calls of the `arrayOf()` function (or equivalent primitive array functions) to create a multidimensional array. Here's an example of creating a two-dimensional array of integers:

```
val arr2D = arrayOf(
    intArrayOf(0, 1, 1),
    intArrayOf(2, 0, 2),
    intArrayOf(3, 3, 0)
)
println(arr2D[2][2])  // output: 0
```

We use `arrayOf()` to create an array of arrays called `arr2D`. Each element of `arr2D` is an array of integers created using `intArrayOf()`. For example, the first element of `arr2D` is an array containing integers `0`, `1`, and `1`. Printing the value of the last element of `arr2D`, designated by `arr2D[2][2]`, will produce an output of `0`.

EXERCISES

Try your hand at the following array-based problems:

1. Find the second-largest element in an array of integers, and print it to the console. For example, if the array is `[1, 2, 3, 4, 5, 6]`, the code should print `5`.
2. Given an array of integers, find the contiguous subarray with the largest sum, and return that sum. For example, given the array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the contiguous subarray with the largest sum is `[4, -1, 2, 1]`, and its sum is `6`.
3. Given an array of integers and a positive integer `k`, rotate the array to the right by `k` positions. For example, given the array `[1, 2, 3, 4, 5]` and a `k` of `2`, the rotated array should be `[4, 5, 1, 2, 3]`.

Collections

Kotlin *collections* are containers that can hold data or objects of the same type or different subtypes of a common supertype (for example, `Any`). Collections can be resized as needed when declared as mutable. This is in contrast to arrays, whose size can't be changed once they're initialized. Kotlin provides various types of collections, such as lists, sets, and maps, each with distinct properties and use cases.

Lists

A *list* in Kotlin is an ordered collection of elements that can be either read-only or mutable. A read-only list is an immutable collection of elements that can't be modified once created. You can perform only read operations like `size`, `contains`, `indexOf`, and `subList` on a read-only list. On the other hand, a mutable list is an ordered collection of elements that supports adding and removing elements or changing the value of a particular element.

Read-Only

You create a read-only list using the `listOf()` function:

```
val list = listOf(1, 2, 3, 2)
```

Notice that this list includes the value 2 twice. The potential for duplicate values is a key feature that distinguishes lists from sets, another type of Kotlin collection. A list created using `listOf()` can contain only one type of data, which will be inferred from the elements of the list.

The elements of a list can be accessed the same way we access array elements: using an index system that starts with 0. Lists also provide the `first()` and `last()` methods for convenient access to the first and last elements directly, without the need for an index. Here's an example:

```
val names = listOf("Mary", "Sam", "Olivia", "Mike", "Ian")
println(names[1])       // output: Sam
println(names.first())  // output: Mary
println(names.last())   // output: Ian
```

Since lists are zero indexed like arrays, `name[1]` returns the second element in the array. Meanwhile, `names.first()` and `names.last()` return the first and last array elements, respectively.

Mutable

If you want the flexibility to modify a list, use `mutableListOf()` to create a *mutable list*. This way, you can change both the list's content and its size, as shown here:

```
val mutableList = mutableListOf(1, 2, 3)
mutableList.add(4)       // Add an element.
mutableList.removeAt(1)  // Remove an element.
mutableList[0] = 5       // Modify an element.
println(mutableList)
```

After creating a list with three elements, we use the `add()` method to append a fourth element to the end of the list and the `removeAt()` method to delete the element at index 1; the later elements in the list slide over to fill in the gap. We also set a new value for the list's first element (index 0). When you run this code, the output should look like this:

```
[5, 3, 4]
```

Another way to append an element to the end of a mutable list is to use the += operator, and you can likewise remove the first instance of an element with the -= operator. You can also use the `removeAll()` method to delete all the elements of a list that meet a specific condition. Here's another example:

```
val fruits = mutableListOf("apple", "banana", "berry", "cherry")
// Add an element using the += operator.
fruits += "plum"
// Remove all elements that start with the letter "b".
fruits.removeAll { it.startsWith("b") }
```

We create a mutable list of fruits and use `+=` to add an extra element to it. Then we call `removeAll()`, providing a lambda expression that checks if each list element starts with `b`. Remember that if a lambda has a single parameter, the `it` keyword can stand in as that parameter. In this case, `it` is a proxy for each element of the list.

One method that's particularly helpful while working with mutable lists is `clear()`, which deletes the list's entire contents:

```
mutableList.clear()
```

This method allows us to reuse a mutable list repeatedly instead of creating new lists that would take up additional memory resources.

It's common to initially create an empty mutable list and to then fill it with elements as needed. In this case, you must include a type declaration for the list when calling the `mutableListOf()` function, as shown here:

```
❶ val list = mutableListOf<Any>()
list.add("hello")
list.add(2)
list.add(33.33)
println(list.joinToString(", "))
```

We use `<Any>` when creating the list ❶ to indicate it can contain any of this supertype's subtypes, including `String`, `Int`, and `Double`. When you run this code, the output should look like this:

```
hello, 2, 33.33
```

Notice how we've used the `joinToString()` method to merge the list elements into a single string that's printed out, with a comma separating each element.

Sets

A *set* is a collection of unique elements, meaning each element can occur only once. Sets don't have a defined order, so two sets are considered equal if they contain the same elements in any order. Sets come in read-only and mutable varieties, created with the `setOf()` or `mutableSetOf()` functions, respectively. Here's an example of each:

```
val readonlySet = setOf(1, 2, 3, 4, 5)
val mutableSet = mutableSetOf("apple", "banana", "cherry")
```

When assigning values to a set in Kotlin, the compiler automatically ignores any duplicate elements. Consider the following code snippet:

```
val mySet = setOf(1, 3, 3, 4, 5, 5, 6)
println(mySet)
```

When you run this code, the output should be:

```
[1, 3, 4, 5, 6]
```

The duplicate values (`3` and `5`) have been filtered out while creating `mySet`. In this way, sets ensure that each element appears only once, making them useful for maintaining unique collections of data.

Every set has a `size` property that reports its number of elements. Sets also have standard methods like `add()`, `remove()`, and `contains()`. In addition, you can use the `union()`, `intersect()`, and `subtract()` methods to create a new set based on the contents of two other sets, as shown here:

```
val set1 = setOf(1, 2, 3)
val set2 = setOf(3, 4, 5)
// set operations
val unionSet = set1.union(set2)
val intersectionSet = set1.intersect(set2)
val differenceSet = set1.subtract(set2)
```

We call the methods on one set, passing the second set as an argument. In this example, `unionSet` holds `{1, 2, 3, 4, 5}`, all the unique elements from both input sets; `intersectionSet` holds `{3}`, the only element present in both input sets; and `differenceSet` holds `{1, 2}`, the elements from `set1` that aren't in `set2`.

EXERCISES

1. Create two sets, `mathStudents` and `scienceStudents`, containing the names of students who have chosen to take math and science courses, respectively. (Assume that students can take both courses if they wish.) Perform the following operations on these sets to analyze the data:
   a. Find and print the students who have chosen both math and science courses (intersection).
   b. Find and print the students who have chosen math courses, science courses, or both (union).
   c. Find and print the students who have chosen math but not science courses (difference).
   d. Find and print the students who have chosen science but not math courses (difference).
   e. Calculate and print the total number of students who have chosen math or science courses.
2. Given a set of integers, write a code snippet to find all possible subsets of the set and return them. For example, given the set `{1, 2}`, the possible subsets are `{}`, `{1}`, `{2}`, and `{1, 2}`.

Maps

A *map* is a collection of key-value pairs, where each key is a label associated with a value. If you've encountered dictionaries in Python or hash maps in Java, the idea is similar. The keys in a map must be unique. As with lists and sets, you can create a map using the `mapOf()` or `mutableMapOf()` functions, as shown here:

```
val ages = mapOf("Alice" to 30, "Bob" to 25, "Charlie" to 35)
val vertices = mutableMapOf("circle" to 0, "triangle" to 3,
    "rectangle" to 4, "pentagon" to 5)
```

We use `ages` to `map` people's names to their ages and `vertices` to map different shapes to their number of vertices. When creating each map, notice how we use `to` for pairing each key (such as `"Alice"`) with a value (such as `30`). The key-value pairs are separated by commas.

Common properties and methods for maps include `size` to return the number of key-value pairs, `get()` to return the value associated with a key, `remove()` to delete a key and its value, `put()` to add a new key-value pair, and `containsKey()` to check if a key is present. Here are a few examples based on the maps created in the previous snippet:

```
val bobAge = ages.get("Bob") // returns the associated value: 25
vertices.put("hexagon", 6)   // adds a new key-value pair
vertices.remove("circle")    // removes the circle-0 pair
val testForCircle = vertices.containsKey("circle") // returns false
println(bobAge)
println(vertices)
println(testForCircle)
```

We retrieve a value from the ages map with `get()` and add a new key-value pair to vertices with `put()`. Then we delete the `"circle"` entry from vertices with `remove()`, meaning that `vertices.containsKey("circle")` should return `false`. This code snippet should produce the following output:

```
25
{triangle=3, rectangle=4, pentagon=5, hexagon=6}
false
```

Notice that when we call the `put()` method, we don't use the same *`key to value`* syntax used when creating a map. Instead, we provide the key and value as individual arguments, separated by a comma.

We've only scratched the surface of Kotlin's various collections and their properties and methods. For more, I recommend consulting the official Kotlin documentation at *https://kotlinlang.org/docs/home.html*.

EXERCISES

Use maps and their methods to solve the following problems:

1. Given an array of integers, find the most frequent element in the array and return it. For example, given the array `[1, 2, 3, 2, 1, 4, 2]`, the most frequent element is `2`. (Hint: Make each unique element in the array a key in a map, paired with a value counting how many times that element occurs.)
2. Given a string, create a map where the keys are individual characters from the string and the values represent the count of each character. Return the resulting map.

An Introduction to Classes

A *class* in Kotlin is a template for creating custom objects. It specifies the properties (variables) and methods (functions) that all objects of that class should have. When you use a class in your code to make an object, you create an *instance* of that class. This process is called *instantiation*. Classes are the core building blocks of the object-oriented style of programming. While objects are codified models of real-world entities or concepts, you can also think of the classes used to create them as custom containers that encapsulate data and functionality into a single unit.

Classes simplify building complex systems by providing abstraction levels. When we categorize objects into classes, we can abstract their common features and behaviors into a single unit. For example, consider a `Person` class representing any person in code. It has properties like `name` and `age`, along with methods like `speak()` and `walk()`. Instances of this class (representing specific people) fill in their own values for properties and have access to the shared methods.

Classes further help us model complex systems in a modular way using *subclasses*. For instance, the `Person` class can have subclasses like `Teacher`, `Student`, and `Athlete`. Each subclass inherits general properties and methods from the `Person` superclass while adding new features specific to the subclass. For example, a `Teacher` might have an `isTenured` property, and a `Student` might have a `gradeLevel` property.

To create a class in Kotlin, you declare it with the `class` keyword, followed by the name of the class and its body, which is enclosed in braces. By convention, class names should begin with a capital letter. Here's an example of a simple class declaration:

```
class Person {
    var name: String = ""
    var age: Int = 0
}
```

Here, we define a class called `Person` that has two properties: `name` and `age`. These properties are declared just like regular variables in the class body and are assigned initial values of `""` (an empty string) and `0`, respectively. With the `Person` class declared, we can now create an individual instance of the class and change its properties, like this:

```
val person1 = Person()
person1.name = "John"
person1.age = 25
```

Here we create a new `Person` object and store it in the `person1` variable. We do this by invoking the `Person()` constructor, which returns a new object of the `Person` class (you'll find more on constructors in the next section). We then assign values to the object's `name` and `age` properties, accessing these properties using dot notation. Why limit ourselves to just one `Person` object, however? The beauty of classes is that we can use them to create as many different objects of that class as we want. Let's create another `Person` object:

```
val person2 = Person()
person2.name = "Irina"
person2.age = 21
```

This time we store the object in the `person2` variable and give it its own values for the `name` and `age` properties, distinct from those of `person1`.

Constructors

In the previous example, you saw how class properties can be initialized directly in the class body, but it's more common to initialize class properties using a *constructor*. This is a special function that's automatically called when a new object of a class is created. In Kotlin, you can initialize properties using a primary constructor or a secondary constructor, both of which we'll discuss in this section.

Primary Constructors

A *primary constructor* is defined in the class header, a set of parentheses after the class name. The primary constructor lays out the names and data types of class properties using the following syntax:

```
class MyClass(val property1: Type1, val property2: Type2) {
// class body
}
```

In this syntax, class properties are listed as a sequence of *`name`*`: `*`type`* entries, similar to declaring function parameters. All property names must be preceded by `val` or `var`. With the class properties declared in this way, we can then provide specific values for the properties when creating an object of the class, much like passing argument values to a function.

Besides declaring the name and type of each property, we can also provide default values for the properties in the class header as part of the primary constructor. To illustrate, let's redefine the `Person` class, adding a header with a primary constructor. We'll then create a couple of instances of the class and initialize their properties through the constructor:

```
class Person(val name: String = "", val age: Int = -99) {
    override fun toString(): String {
        return "Person(name=$name, age=$age)"
    }
}

fun main() {
    val person1 = Person("John", 25)
    val person2 = Person("Irina", 21)
    println(person1)
    println(person2)
}
```

This code segment defines a class named `Person` with two properties, `name` and `age`, specified in the class header, which serves as the primary constructor. We give `name` a default value of `""` (an empty string) and `age` a default value of `-99`. The class also overrides (redefines) the `toString()` method; we'll discuss this in more detail shortly.

In the `main()` function, we create two objects of the `Person` class, passing values for the `name` and `age` properties as arguments to the constructor. This saves us from having to write separate statements setting these property values, as we did in our original implementation of the class. We then print the two `Person` objects to the console. When you run this code, the output should be as follows:

```
Person(name=John, age=25)
Person(name=Irina, age=21)
```

When you pass an object to `println()`, Kotlin automatically invokes that object's `toString()` method to display some kind of string representation of the object. All classes come with a default implementation of this method, inherited from the general `Any` class, but this default implementation displays only the class name and the object's hash code (a unique integer identifier), which is not very informative or readable. Overriding the `Person` class's `toString()` method with a customized definition allows us to display the properties of the object in a more meaningful way. We'll learn more about overriding methods inherited from a parent class later in this chapter.

Secondary Constructors

Kotlin classes can also have one or more *secondary constructors* containing additional parameters or logic that should be invoked when new objects are created. The secondary constructor isn't part of the class header but is instead defined inside the class body with the `constructor` keyword. If a

primary constructor also exists, a secondary constructor must always delegate to it (that is, call it), either directly or indirectly through other secondary constructors, using the `this` keyword.

A class can have just a primary constructor, primary and secondary constructors, or just secondary constructors. While secondary constructors aren't mandatory, they can be useful in certain scenarios. For instance, when we need to initialize too many class properties, it may be more convenient to initialize them inside a secondary constructor. This mechanism also allows you to set default values for properties that aren't initialized in the primary constructor. Moreover, secondary constructors allow you to create instances of the class with different combinations of properties. This is like function overloading, where you have multiple functions with the same name but different parameters.

The following example shows how to create and use secondary constructors. In the process, it illustrates all three ways of initializing class properties: inside the class body, using a primary constructor, and using secondary constructors.

```
class Car(val make: String, val model: String, val year: Int) {
    // property initialization inside class body
  ❶ var color: String = "Unknown"

    // 1st secondary constructor (no args)
    constructor() : this("Unknown", "Unknown", 0)

    // 2nd secondary constructor (1 arg)
  ❷ constructor(make: String) : this(make, "Unknown", 0)

    // 3rd secondary constructor (2 args)
    constructor(make: String, model: String) : this(make, model, 0)

    override fun toString(): String =
        "Make: ${make}, Model: ${model}, Year: ${year}, Color: ${color}"
}

fun main() {
    val c1 = Car()
    val c2 = Car("Nissan")
    val c3 = Car("Toyota", "Prius")
    val c4 = Car("Ford", "Mustang", 2024)

    c1.color = "Blue"
    c2.color = "Red"
    c3.color = "Black"
    c4.color = "Yellow"

    println(c1)
    println(c2)
    println(c3)
    println(c4)
}
```

In this example, the `Car` class has a primary constructor with three parameters: `make`, `model`, and `year`. It also has three secondary constructors with zero, one, and two parameters, respectively. These secondary constructors use the `this` keyword (after a colon) to call the primary constructor, passing along the received parameter values while filling in default values for the missing ones. For example, the second secondary constructor ❷ takes in a value for the `make` property while providing default values of `"Unknown"` and `0` for the `model` and `year` properties. Notice how the `Car` class also has a `color` property that's initialized to `"Unknown"` in the class body ❶, separate from any of the constructors. This property can be set to a different value after a `Car` object is created, because it was declared with a `var`.

In the `main()` function, we create four `Car` objects using the different constructors. Kotlin determines which one to invoke based on the number of arguments supplied. For example, `c1` will be created with the first secondary constructor, since no arguments are provided, while `c4` will be created with the primary constructor, since all three arguments are provided. We then set the `color` property of each object and print out the details of each object using the class's `toString()` method, which we've again overridden. The code should produce the following output:

```
Make: Unknown, Model: Unknown, Year: 0, Color: Blue
Make: Nissan, Model: Unknown, Year: 0, Color: Red
Make: Toyota, Model: Prius, Year: 0, Color: Black
Make: Ford, Model: Mustang, Year: 2024, Color: Yellow
```

Notice how the objects range from having all default values when no arguments were supplied to having all custom values when three arguments were supplied.

The init Block

In Kotlin, you can use an `init` block within a class to run a code segment during object construction. The `init` block is automatically executed when an object of the class is created. If you have multiple `init` blocks, they'll run in the same order in which they appear inside the class.

Here's an example of how to use the `init` block to initialize properties within a class:

```
class Person (var name: String, var age: Int) {
    // additional property
    var isMinor: Boolean = false

    // init block for custom initialization
    init {
        if (age < 18) isMinor = true
    }
}
```

We give the `Person` class an `init` block that changes the `isMinor` property from `false` to `true` when the `age` property of the `Person` object is less than `18`.

This block will be executed whenever a Person object is created to adjust the isMinor value as needed.

Both init blocks and secondary constructors can be used to initialize class properties or run additional logic when an object is created. The init block is preferred over the secondary constructor when we need to run additional code after initializing certain properties. (In the previous example, we changed the value of the isMinor property after initializing it using an init block.) The init block can use primary constructor parameters and is executed immediately after the primary constructor but before executing any secondary constructors.

On the other hand, secondary constructors are more useful when you need to provide additional ways to instantiate a class with different combinations of properties. Still, the two mechanisms share many similarities; when coding, you often have multiple ways to complete a task.

Methods

Methods are functions associated with a class that can be called through objects of that class to perform certain actions. A class's methods are declared as part of the class body. To illustrate, let's add a sayHello() method to our Person class:

```
class Person (var name: String = "Unknown", var age = -99) {
    fun sayHello() {
        println("Hello, my name is $name, " +
                "and I am $age years old.")
    }
}
```

Inside the class body, we declare a sayHello() method that uses string templates to print a greeting that includes the person's name and age properties. Notice that declaring a method follows the same syntax as declaring an ordinary function, with the fun keyword and a body enclosed in braces.

To use this method, create a Person object and invoke the method via dot notation, like so:

```
val person = Person("John", 25)
person.sayHello()
```

This will output:

```
Hello, my name is John, and I am 25 years old.
```

You can add more methods to the Person class to perform other actions or calculations based on the object's properties. Methods can also take parameters and return values, just like regular functions.

Encapsulation

Encapsulation is a fundamental principle of object-oriented programming that helps you control access to the internal state of an object. This control is achieved through *access modifiers*, keywords that specify the visibility or accessibility of a property or method. Access modifiers allow you to hide (*encapsulate*) the implementation details of a class and maintain the integrity of the objects of a class by reducing the risk of unintended modifications. The two most important access modifiers in Kotlin are `public` and `private`.

Unless declared otherwise, all properties and methods of a class are considered *public* by default. This means they can be accessed from anywhere in the code. *Private* properties and methods, by contrast, can be accessed only within the class declaration itself. For instance, if you designated the `Person` class's `age` property as private, you could reference it within the declarations of `Person` class methods, but you couldn't use it in code outside the class declaration, such as to update a `Person` object's `age` value from the `main()` function. This protects the `age` property from being altered in unintended ways.

Here's an example of how to harness encapsulation and the `private` access modifier within the `Person` class:

```
class Person(private var name: String, private var age: Int) {
    fun introduce() {
        println("Hi, I'm $name, and I'm $age years old.")
    }

    fun haveBirthday() {
        age++
    }
}

fun main() {
    val person = Person("Alice", 30)

    // Access and modify properties using public methods.
    person.introduce()
    person.haveBirthday()
    person.introduce()

    // Trying to access private properties directly
    // will result in a compilation error.
❶   // println(person.name)
    // person.age++
}
```

In the `Person` class header, we use the `private` access modifier to designate the `name` and `age` properties as private. This way, the properties can be accessed and modified only within the class itself. We also give the class two methods, `introduce()` and `haveBirthday()`, which are considered public by default. These methods provide controlled access to the private properties, with `introduce()` displaying the values of `name` and `age`, and `haveBirthday()`

incrementing age. In effect, this restricts how a Person object's age property can be updated; it can increase by only one year at a time, rather than by jumping abruptly from, say, 30 to 40.

In the main() function, we create a Person object, passing initial values for name and age (it's still possible to set the values of private properties through the constructor). Then we call the public introduce() and haveBirthday() methods, and introduce() again, producing the following output:

```
Hi, I'm Alice, and I'm 30 years old.
Hi, I'm Alice, and I'm 31 years old.
```

In this way, we're able to access and modify the object's private properties indirectly through its public methods. However, we can't access or modify the private properties directly, as we attempt to do in the commented-out lines of code ❶. If you try removing these comments to execute these statements, you'll get compilation errors because the properties are private.

Kotlin also has two additional access modifiers, protected and internal. *Protected* properties and methods are like private ones, except they can be accessed within subclasses of a class, as well as within the class itself (more on subclasses shortly). *Internal* properties and methods can be accessed only within the same module.

NOTE *A* module *is a set of Kotlin files that must be processed together during compilation. Files or functions accessed through an* import *statement aren't considered part of the module itself but rather external dependencies used by the module.*

The this Keyword

Inside a Kotlin class declaration, the this keyword is a reference to the current instance of the class. For example, if you see this.name inside a method definition for the Person class, it simply refers to the value of the name property of whatever Person object the method is called on. We haven't used the this keyword in our Person class examples so far, because it's been clear from the code that variables like name and age are properties of the class. The keyword becomes important when we need to distinguish between class properties and method parameters with the same name. Here's an example of a Book class where this is necessary:

```
class Book(var title: String, var author: String) {
    fun displayInfo() {
        println("Title: $title")
        println("Author: $author")
    }

    fun updateInfo(title: String, author: String) {
        this.title = title
        this.author = author
    }
}
```

```
fun main() {
    val book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
    // Display book information.
    book1.displayInfo()

    // Update book information.
    book1.updateInfo("To Kill a Mockingbird", "Harper Lee")

    println("\nUpdated book information:")
    book1.displayInfo()
}
```

In this example, we have a `Book` class with `title` and `author` properties and two methods. The `displayInfo()` method displays the book's title and author, and the `updateInfo()` method takes in new values for `title` and `author` as arguments and updates the properties of the class with these values. Notice how we use the `this` keyword in the `updateInfo()` method to distinguish `this.title` and `this.author` (the `title` and `author` properties of the class) from `title` and `author` (the method's parameters). In this way, we can update the book's information correctly without any naming conflicts.

In the `main()` function, we instantiate a `Book` object and then use its `updateInfo()` method to change its title and author, displaying the book's properties before and after the change with `displayInfo()`. The output should appear as follows:

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald

Updated book information:
Title: To Kill a Mockingbird
Author: Harper Lee
```

Try removing the `this` keywords from the `updateInfo()` method, leaving `title = title` and `author = author`. The code will no longer work: Kotlin will try to interpret `title` and `author` as local variables being declared without a `val` or `var`, and it won't be able to initialize those local variables with the arguments supplied (which wouldn't be our intention anyway).

While the `this` keyword refers to the current instance of the class when it appears inside a class method, it has other meanings in other contexts. As we've already seen, when `this` appears after a colon in a secondary constructor, it serves to delegate to the primary constructor of the same class. For information about additional uses of the `this` keyword, see the official Kotlin documentation.

Inheritance and Polymorphism

Inheritance and *polymorphism* are interrelated tenets of object-oriented programming. Inheritance allows a more specialized subclass, or *child class*, to take on properties and methods from a more general superclass, or *parent class*; polymorphism enables the child class to override and extend the behavior of its parent. Together, inheritance and polymorphism foster

flexibility and code reusability, and they enable different child classes to extend the same inherited parent method in distinct ways.

Unlike some other programming languages, Kotlin classes aren't inheritable by default. Instead, you need to explicitly mark the parent class with the `open` keyword to make it inheritable by child classes. Then, after the header in the child class declaration, you specify the parent class name (with a colon before its name). This establishes the inheritance relationship.

Here is a simple example of creating a child class that has its own unique property in addition to what it inherits from the parent class:

```
open class ParentClass(val name: String, val age: Int) {
    init {
        println()
        println("Hello, I am $name, and I am $age years old.")
    }
}

class ChildClass(name: String, age: Int, val occupation: String)
    : ParentClass(name, age) {

    init {
        println("My occupation is $occupation.")
    }
}

fun main() {
    // Create instances of parent and child classes.
    val person1 = ParentClass("John", 33)
    val person2 = ChildClass("Sarah", 24, "accountant")
}
```

Notice that we've used the `open` keyword before the parent class. This indicates that the class can be inherited by a child class. The primary constructor of the parent class requires two properties: `name` of type `String` and `age` of type `Int`. Since no default values are provided for these properties, their values must be supplied when instantiating a child class. Additionally, the child class introduces a new property called `occupation`, which also requires a value during instantiation.

In the `main()` function, we create `person1` as an instance of the parent class and `person2` as an instance of the child class. Since both classes have `init` blocks, when you run this program the output should resemble the following:

```
Hello, I am John, and I am 33 years old.

Hello, I am Sarah, and I am 24 years old.
My occupation is accountant.
```

If you intend to customize or override properties or methods from the parent class in the child class, you also need to mark them individually with the `open` keyword in the parent class. Then, in the child class, use

the override keyword before these properties or methods. This ensures that the compiler recognizes your intention to override the parent class's implementation.

Here is a simple example of overriding a parent class's method within the child class:

```
// parent class
open class Vehicle {
    open fun startEngine() {
        println("Vehicle engine started")
    }
}

// child class
class Car : Vehicle() {
    override fun startEngine() {
        println("Car engine started")
    }
}

fun main() {
    val myCar = Car()
    myCar.startEngine()
}
```

In this example, we employed the open keyword twice—once preceding the parent class (Vehicle) and once when declaring the startEngine() method within the parent class. Then, within the child class (Car), we modified the method using the override keyword. As a result, when you run this code, it should yield the following output:

```
Car engine started
```

While we've covered the basics of inheritance and polymorphism, you still have much more to explore. I recommend consulting the official Kotlin documentation for additional use cases.

Common Classes and Custom Types

Now that we've outlined some basic principles of classes and object-oriented programming, in this section we'll explore several commonly used classes and custom types in Kotlin. These include data classes, pairs and triples, abstract classes, interfaces, and enum classes. If you're new to classes and haven't used any of these features before, you might want to start with a quick overview before getting into the details. Table 2-2 provides names, short descriptions, and use cases of the features we'll cover. You can refer to this table if you need to refresh your memory or clarify any concepts.

Table 2-2: Commonly Used Classes and Custom Types

Class	Definition	Use cases
Data class	A simple class primarily used to hold data. It doesn't have any user-defined methods. Data classes are marked with the `data` keyword.	Used as basic building blocks for modeling data by providing descriptive names to pair with values. They often serve as building blocks for more complex data structures.
`Pair` and `Triple`	Simple classes for holding two (`Pair`) or three (`Triple`) values of the same or different data types.	Used to store or return two or three values in a single instance, especially when you don't need descriptive names for the values.
Abstract class	A class that can't be instantiated and can have abstract members that must be overridden by its subclasses.	Used for defining a common set of features for a group of related classes.
Interface	A collection of functions and properties that must be implemented by inheriting classes or types.	Used to enforce methods and properties on other types (classes, functions, custom types).
Enum class	A special class type that represents a group of constants with optional properties and methods.	Used for representing a fixed set of values.

We'll review each of these concepts with detailed examples in the sections that follow.

Data Classes

In Kotlin, a *data class* is a class whose main purpose is to hold data and not to perform complex operations or logic. Essentially, it's a class with properties but no custom methods (although adding such methods isn't prohibited). To declare a data class, you need to add the `data` keyword before the `class` keyword and include at least one parameter in the primary constructor.

Based on the properties declared in the primary constructor, a data class can automatically generate a number of methods, including:

`equals()` Compares two data class instances for equality.

`toString()` Returns a human-readable string representation of the object.

`copy()` Creates a shallow copy of the data class instance. (See "Copying Objects" on page 75 for information about shallow copies.)

`hashCode()` Generates a *hash code*, a unique integer based on a hashing algorithm applied to one or more properties of a class. This method is used in conjunction with `equals()` to determine if two objects are equivalent.

By contrast, a regular class in Kotlin doesn't autogenerate such methods; you would have to manually implement them if required.

Here's an example of how to create and use a simple data class in Kotlin:

```
// Declare a data class.
data class Person(val name: String, val age: Int)

fun main() {
    // Create an instance.
    val person = Person("Steve", 40)
    println(person)
}
```

We create a `Person` data class with `name` and `age` properties. Since these properties are declared in the class header, a class body isn't needed. In `main()` we create an instance of the data class, `person`, and pass it directly to the `println()` function. When `println()` encounters an instance of a data class, Kotlin calls the object's `toString()` method automatically, which generates the following output:

```
Person(name=Steve, age=40)
```

Data classes are extremely useful for modeling and working with data in a clean and efficient manner by grouping related values into a single, custom-designed object. They share some similarities with maps, which use key-value pairs rather than class properties to associate names with data values. However, while maps are primarily used to store and retrieve values by key, data classes are better suited to modeling data in a more meaningful and structured way. Data classes also provide those useful autogenerated methods we just discussed.

DECONSTRUCTION

Deconstruction is a feature in Kotlin that allows you to extract multiple properties of an object and assign them to individual variables in a single statement. This way, you can use those properties independently of the object. Say you have a `Person` data class with `name` and `age` properties, and an instance of that class called `person`. You could use deconstruction to extract the values of those properties from the object as follows:

```
val (name, age) = person // deconstruction
```

The variables that receive the deconstructed values go on the left side of an assignment statement, within parentheses, while the object to be deconstructed goes on the right side. The object's properties are extracted in the order in which they're declared in the constructor, so here the value of the `name` property goes to the `name` variable and the `age` property goes to the `age` variable. (The names happen to match, but this doesn't have to be the case.)

(continued)

After deconstruction, the values are available through the simple variables—for example, for printing:

```
println("Name: $name")    // Print the value of name.
println("Age: $age")      // Print the value of age.
```

Notice how we're using the `name` and `age` variables in the `println()` calls, without having to reference the `person` object.

Pairs and Triples

A *pair* in Kotlin is a data class that can store exactly two values, which can be of the same or different types. Pairs are useful for storing two related values in a single object, such as the x- and y-coordinates for a point on a graph or the name and age of a person. They also provide a way to associate a key with a value. In the latter case, the first value in the pair is a string descriptor of the second value. You can create a pair with the `Pair()` constructor, passing the two values as arguments, or without the constructor by placing `to` between the two values in an assignment statement. Here's an example of each technique:

```
val pair1 = Pair("Alice", 20)
val pair2 = "Bob" to 25
```

A *triple* is a similar structure for storing three related values in a single object, such as the name, age, and gender of a person, or the RGB color components of a pixel. You can create a triple with the `Triple()` constructor as follows:

```
val triple1 = Triple("Alice", 20, "Female")
```

Once a pair or triple is created, it's immutable, so its values can't be updated. Those values are accessible using dot notation as the `first`, `second`, and `third` properties. For example:

```
val pair = "Hello" to "World"
val triple = Triple(1, 2, 3)

println(pair.first)   // Hello
println(triple.third) // 3
```

You can also use deconstruction syntax (see the "Deconstruction" box) to extract the data elements of a pair or triple into individual variables. Here's an example:

```
val pair = Pair("John", 29)
val (name, age) = pair // deconstruction
println("Name: $name") // Print the value of name.
println("Age: $age")   // Print the value of age.
```

We create a `Pair` called `pair` with two values: `John` and `29`. We then use deconstruction to extract these values and assign them to the `name` and `age` variables. From there, we can use the variables independently to print out the `name` and `age` of the `pair` object.

Abstract Classes

In Kotlin, an *abstract class* is a class that can't be instantiated on its own. Instead, it serves as a blueprint for other classes to extend through inheritance and polymorphism. You can use abstract classes when you want to provide a common foundation or framework—including methods and properties—that must be implemented and fleshed out by various child classes but that wouldn't itself hold up as a fully functional class. In this sense, an abstract class serves much like a regular parent class declared with `open` in that it allows inheritance and permits the overriding of properties and methods. The key difference lies in the fact that you can't directly instantiate an abstract class.

You declare an abstract class using the `abstract` keyword. It can have abstract properties (with no initial values, just names and data types) and abstract methods (with no implementation, just names and return types). Abstract properties and methods are declared using the `abstract` keyword, just like the class itself. Abstract classes can also have concrete (nonabstract) properties and methods—complete variable or function declarations that provide default behavior.

Any class that inherits from an abstract class must implement the inherited abstract properties and methods, giving them concrete values and definitions. If the child class doesn't do this, it must be declared as an abstract class as well. Subclasses also have the option to override the concrete members of the abstract class.

Here's an example of how all of this works, where we create an abstract `Shape` class and use it as a model for `Circle` and `Square` classes:

```
abstract class Shape {
    abstract fun area(): Double  // abstract method
    val name: String = "Shape"   // concrete property

    fun describe() {
        println("This is a $name")
    }
}

class Circle(val radius: Double): Shape() {
    override fun area(): Double {
        return Math.PI * radius * radius
    }
}
```

```
class Square(val side: Double): Shape() {
    override fun area(): Double {
        return side * side
    }
}

fun main() {
    val circle = Circle(5.0)
    val square = Square(4.0)

    circle.describe()
    println("Area of the circle: ${circle.area()}")

    square.describe()
    println("Area of the square: ${square.area()}")
}
```

We use the `abstract` keyword to designate `Shape` as an abstract class. It has an abstract `area()` method that should return a value of type `Double` and a concrete `name` property with a value of `"Shape"`, as well as a concrete `describe()` method that prints a message. We then declare both `Circle` and `Square` as nonabstract subclasses of `Shape`. Each is given a property unique to the subclass (`radius` for `Circle` and `side` for `Square`), and each inherits the `name` property and `describe()` method from `Shape`. The subclasses must also provide a concrete implementation for the inherited `area()` method using the `override` keyword. In this way, the abstract `Shape` class serves as a common structure for both types of shape, enforcing that any subclass must implement a method that calculates the shape's area.

In `main()` we create an instance of each concrete class and invoke its `area()` method within a string template. The code should produce the following output:

```
This is a Shape
Area of the circle: 78.53981633974483
This is a Shape
Area of the square: 16.0
```

In addition to ensuring consistency through a shared structure between the parent and the child classes, abstract classes reduce code duplication, improve code readability, and simplify code maintenance.

Interfaces

An *interface* is a collection of methods and properties that form a common set of behaviors that the types implementing the interface must follow. These methods and properties are abstract in the sense that we can't use them directly, but we don't use the `abstract` keyword when defining them. Interfaces can contain declarations of abstract methods and properties, as well as method implementations. However, they can't store state, meaning they can't contain any fields or properties that store data.

A class or object can implement one or more interfaces. When a class implements an interface, it must provide full definitions for all the abstract methods and properties declared in that interface. In this sense, the interface acts as a common contract for the classes that implement it, laying out the features that any implementing class must agree to have.

Here's an example of how to define and use an interface in Kotlin:

```
import kotlin.math.PI

interface Properties {
    fun area(): Double
    fun perimeter(): Double
}

class Circle(val radius: Double): Properties {
    override fun area() = PI * radius * radius
    override fun perimeter() = 2 * PI * radius
}

fun main() {
    val circle = Circle(4.0)
    val area = circle.area()
    val perimeter = circle.perimeter()

    println("Properties of the circle:")
    println(" radius = ${circle.radius}\n area = $area\n" +
            " perimeter = $perimeter")
}
```

We use the `interface` keyword to declare the `Properties` interface. It defines two abstract methods, `area()` and `perimeter()`, both of which return a floating-point value. Any class that implements the interface, such as the `Circle` class declared here, must include definitions for both methods.

The syntax for implementing an interface is similar to that of inheritance: a colon after the class header, followed by the name of the interface. Notice that we also need to use the `override` keyword when implementing the functions from the interface.

In `main()`, we create an instance of the `Circle` class and invoke its `area()` and `perimeter()` methods, storing the results in the local `area` and `perimeter` variables. Then we print these values to the console, generating the following output:

```
Properties of the circle:
 radius = 4.0
 area = 50.26548245743669
 perimeter = 25.132741228718345
```

Kotlin interfaces can also inherit from other interfaces, meaning they can provide implementations for the inherited members and declare new functions and properties. However, classes implementing such an interface are required to define only the missing implementations.

ABSTRACT CLASSES VS. INTERFACES

Abstract classes and interfaces share some common features, but they also have some important differences. An abstract class is a higher-level framework (superclass) for a group of related classes. It can have functions and properties, and it can hold state. Methods and properties designated as abstract in an abstract class must be fully implemented in the inheriting class. An abstract class can inherit multiple interfaces but can extend only one class. Furthermore, an abstract class can't be directly instantiated.

An interface is a collection of methods and properties that an inheriting type is forced to implement. Like an abstract class, an interface can't be instantiated directly (it's not a class at all but rather a custom type). Its purpose is limited to forcing the implementation of its methods and properties. Unlike an abstract class, an interface can't hold state.

Enum Classes

An *enum* (short for *enumeration*) is a special kind of class that defines a finite set of constant values. Enums are typically used to represent a fixed set of related values, like days of the week, cardinal directions, status codes, playing card suits, and seasons. In Kotlin, we use the `enum class` keywords to define an enum, followed by the class name. Then comes a comma-separated list of the enum's constants, enclosed in braces. Here's an example of an enum in Kotlin:

```
// Define an enum class for days of the week.
enum class DayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}

fun main() {
    // using the enum values
    val today = DayOfWeek.MONDAY

    when (today) {
        DayOfWeek.MONDAY -> println("It's a manic Monday!")
        else -> println("It's some other day.")
    }
}
```

In this example, we defined an enum class called `DayOfWeek` representing the days of the week. The body of the class contains a comma-separated list of the enum's constant values, which, by convention, are written in all caps. In the `main()` function, we create a variable `today` and assign it the value `DayOfWeek.MONDAY` from the enum. Enum constants are always accessed this way, using dot notation that couples the enum class name with the specific

constant name. We then use a `when` expression to check the value of `today` and print a message based on the day. The program should print `It's a manic Monday!` because `today` is set to `DayOfWeek.MONDAY`.

EXERCISE

Create a simple game that involves different types of characters (wizards and warriors). Each character has a name, a level, and a set of abilities. Some characters can fly, while others can't. This is an open-ended problem, so use your imagination!

1. Create an abstract class called `Character` that defines the basic properties and methods of any character.
2. Create an interface called `Flyable` that defines a `fly()` method.
3. Create two subclasses of `Character` called `Wizard` and `Warrior`. The `Wizard` class should implement the `Flyable` interface, while the `Warrior` class should not.
4. Create a `Game` class that creates instances of `Wizard` and `Warrior` and calls their respective methods.
5. Finally, create an instance of the `Game` class in the `main()` function and play the game.

Copying Objects

In many cases, you'll need to copy an object, meaning you create a new instance of an object with the same or modified values compared to the original object. In Kotlin, you can create either a shallow copy or a deep copy of an object. The difference comes down to whether and how the original and the copy are connected. Which type of copy you use depends on the circumstances and the structure or complexity of the original object.

Shallow Copy

A *shallow copy* in Kotlin involves creating a new object that mirrors an existing one. However, the copy doesn't fully replicate any nested objects within the original object. Instead, the copied object retains the same references to the nested objects as the original one. Therefore, a change to a nested object in the original version affects the copied version as well, and vice versa. As noted earlier, the `copy()` method that comes built in to any data class creates shallow copies.

To illustrate, say we define a `Person` data class with a `name` property and a `hobbies` property, the latter being a `MutableList<String>`. The `hobbies` property is considered to be nested, since a list of strings is itself an object within

the overarching `Person` object. If we use the class's built-in `copy()` method to copy a `Person` object, the copy will be shallow. The new instance will share the same list reference as the original one, so whether we modify the original list or the shallow-copied list, both instances of the data class will be affected. Here's some code that demonstrates this behavior:

```
data class Person(val name: String,
                  val hobbies: MutableList<String>)
fun main() {
    val person1 = Person("Bob", mutableListOf("Reading", "Gaming"))
  ❶ val person2 = person1.copy()

    // Print both objects.
    println(person1)
    println(person2)

    // Add a new element to the mutable list of person1.
    person1.hobbies.add("Coding")

    // Print both objects again.
    println(person1)
    println(person2)
}
```

We declare the `Person` data class as described and create two instances of the class. The first, `person1`, is instantiated from scratch, while `person2` is created by copying `person1` ❶. To see the implications of shallow copying, we print both objects, add one more hobby to the `person1` object's `hobbies` list, and print both objects again. The code should generate the following output:

```
Person(name=Bob, hobbies=[Reading, Gaming])
Person(name=Bob, hobbies=[Reading, Gaming])
Person(name=Bob, hobbies=[Reading, Gaming, Coding])
Person(name=Bob, hobbies=[Reading, Gaming, Coding])
```

Notice that even though we modified the `hobbies` property only of `person1`, `person2` was affected in the same way. This is because the `hobbies` property of `person2` isn't a true clone; it references the same memory location as the `hobbies` property of `person1`. Keep in mind this applies only to nested objects; if we updated the `name` property of `person1`, the change would apply only to `person1`, since this property isn't nested.

Shallow copying can be useful for performance optimization as it avoids duplicating large amounts of data and taking up additional memory space. But what if you need the two instances of the `Person` class to be entirely independent of each other? This is where deep copies come into play.

Deep Copy

A *deep copy* creates a new, completely independent object from an existing object by copying all its nested objects as well as its nonnested properties.

This results in two separate and unrelated objects, so changes to one won't affect the other. In Kotlin, you usually have to write a custom function tailored to the class at hand to make deep copies. Here's a simple example:

```
data class Address(var street: String, val city: String)
data class Person(val name: String, val address: Address)

fun deepCopyPerson(person: Person): Person {
  ❶ val clonedAddress = Address(person.address.street,
                                person.address.city)
    return Person(person.name, clonedAddress)
}

fun main() {
    val originalPerson = Person("Alice", Address("123 Main St", "Cityville"))
    val copiedPerson = deepCopyPerson(originalPerson)

    // Modify the original address.
    originalPerson.address.street = "456 Elm St"

    // Check if the copied address remains unchanged.
    println(originalPerson.address.street) // output: 456 Elm St
    println(copiedPerson.address.street)   // output: 123 Main St
}
```

We declare two data classes, `Address` and `Person`. Notice how the `Person` class has an address property of type `Address`, meaning this property is a nested object. To achieve deep copying for the `Person` class, we declare a `deepCopyPerson()` function. The function first creates a new `Address` object by manually extracting the nested address properties from the original `Person` object ❶. Then it returns a new `Person` object containing the original object's `name` property along with the deep-copied `Address` object.

In `main()`, we create a `Person` object, then use `deepCopyPerson()` to copy it. At that point, we can modify the `address` property of the original `Person` object and it won't have any effect on the copy, since the nested object has been copied during the deep-copying process.

It's also common to have to make a deep copy of a list of objects. This can be done with just one line of code by using the list's `map()` method to call `copy()` on each object. Here's how it works:

```
data class Person(var name: String, var age: Int)

fun main() {
    // original mutable list
    val originalList =
        mutableListOf(Person("Alice", 30), Person("Bob", 25))

    // Deep-copy the list using map() and copy().
    val deepCopyList =
        originalList.map{ it.copy() }.toMutableList()
}
```

In this example, we have a mutable list of `Person` data class objects, where each `Person` has two properties: `name` and `age`. We create a deep copy of the list using the `map()` method, which iterates through the elements of the original list, applies a function to each one, and stores the results in a new list. In this case, the applied function is the lambda expression `it.copy()`, which copies the current `Person` objects (this is possible because the `Person` class doesn't have any nested objects). We chain a call to `toMutableList()` after the lambda, since `map()` results in a regular, read-only list rather than a mutable list.

I invite you to add a few more lines of code to the previous listing, modifying the properties of a `Person` object in the original list and then printing both lists. You should find that making changes to the original list doesn't affect the copied list (and vice versa).

Project 2: Build a Versatile Task Manager

Let's apply what we've learned about data structures in this chapter to a simple project: we'll create a console-based task manager application. The application will allow users to keep track of their daily tasks, with the following key functionalities:

- Adding tasks to the task list
- Displaying a list of all the tasks
- Marking a task as done
- Deleting unwanted or completed tasks
- Exiting the program

Our primary challenge is to maintain a list of tasks. Since tasks can be added and removed, a mutable list is an appropriate structure. We also must decide on the attributes that define each task. We can encapsulate these in a data class. Additionally, we need to handle user interactions effectively, providing users with options and ensuring robust error handling for invalid inputs, concepts we touched on in Chapter 1.

The Code

Let's start with a high-level overview of the program's structural components before getting into the details of individual pieces and how they interact. Here's an outline of the data structures, functions, and logic that we'll need:

```
// macro view of the task manager program

data class Task(val title: String,
                val description: String,
                var status: String = "not done"
)

class TaskManager {
  ❶ val taskList = mutableListOf<Task>()
```

```
    fun addTask(task: Task) {...}
    fun listTasks() {...}
    fun markTaskAsDone(taskIndex: Int) {...}
    fun deleteTask(taskIndex: Int) {...}
}

fun printOptions() {...}
fun readIndex(taskListSize: Int): Int? {...}

fun main() {
    val taskManager = TaskManager()

    while (true) {
        printOptions()
        when (readln()) {
            "1" -> {...}
            "2" -> {...}
            "3" -> {...}
            "4" -> {...}
            "5" -> return   // breaks the while loop
            else -> println("\nInvalid choice. Please try again.")
        }
    }
}
```

The project comprises five main code blocks. First, the `Task` data class defines the structure of each individual task; each task will have a `title`, a `description`, and a `status` property set to `"not done"` by default. Next, the `TaskManager` class holds methods responsible for all task management work, like adding, listing, and deleting tasks. Notice its `taskList` property ❶, a mutable list for storing all the current tasks. Two stand-alone helper functions, `printOptions()` and `readIndex()`, support user interaction and input handling. Finally, the `main()` function oversees presenting options to the user and directing flow based on user choices.

We'll now explore the `main()` function and its components in a top-down manner. As we go, we'll implement the missing code blocks designated by `{...}` in the previous listing.

The `main()` function begins by creating an instance of the `TaskManager` class named `taskManager`. This class, in turn, initializes a mutable list of type `Task` as its `taskList` property. This list starts out empty, but it's a mutable list, so we can add or remove elements as needed.

Next, we invoke a `while` loop to repeatedly present the user with a menu of task management options and respond to the user's requests. The loop's condition is simply `true`, meaning it will repeat indefinitely unless the user chooses the option for exiting the program (more on this mechanism later). The first part of the loop is a call to the `printOptions()` function, defined here:

```
fun printOptions() {
    println("\nTask Manager Menu:")
    println("1. Add Task")
```

```
    println("2. List Tasks")
    println("3. Mark Task as done")
    println("4. Delete Task")
    println("5. Exit")
    print("Enter your choice (1-5): ")
}
```

The function simply displays the five available commands the user can enter, denoted by the numbers `1` through `5`.

Adding a Task

After printing the options, the `while` loop uses a `when` expression to trigger the appropriate code based on which number was entered. Here's a look at the `when` expression again, including the implementation of the `"1"` branch for adding a task:

```
❶ when (readln()) {
    "1" -> {
        print("\nEnter task title: ")
        val title = readln()
        print("Enter task description: ")
        val description = readln()
      ❷ val task = Task(title, description)
      ❸ taskManager.addTask(task)
    }
    "2" -> {...}
    "3" -> {...}
    "4" -> {...}
    "5" -> break   // breaks the while loop
  ❹ else -> println("\nInvalid choice. Please try again.")
}
```

The `when` expression initiates by using `readln()` to take in a line of input (a string) from the console representing the user's menu choice ❶. If the entered value doesn't match any of the five options (`"1"`, `"2"`, `"3"`, `"4"`, or `"5"`), the `else` block within the `when` expression is triggered ❹, signaling to the user that their choice was invalid and they should make another selection.

When the user selects option `"1"`, the code block under `"1" ->` is executed. Since this choice is associated with adding a task, the user is prompted to provide a task title and description. We use these input values (or empty strings if the user simply presses ENTER) to create a new instance of the `Task` data class ❷, which is then passed as an argument to the `taskManager` object's `addTask()` method ❸. This method appends the task to the `taskList` mutable list, like this:

```
fun addTask(task: Task) {
    taskList.add(task)
}
```

We call the list's `add()` method to insert the new task at the end of the list.

Listing the Tasks

When the user selects option `"2"` (listing the tasks), the code block under `"2" ->` of the `when` expression is executed. This makes a call to the `listTasks()` method of the `taskManager` object:

```
when (readln()) {
--snip--
    "2" -> taskManager.listTasks()
```

Let's take a look inside the `listTasks()` method, which is the second method defined in the `TaskManager` class:

```
fun listTasks() {
    if (taskList.size > 0) {
        println("\nTasks:")
        for ((index, task) in taskList.withIndex()) {
          ❶ println("${index+1}. ${task.title} - " +
                    "${task.description} - ${task.status}")
        }
    } else
        println("Task list is empty.")

}
```

Inside `listTasks()`, we first check if the `taskList` has any tasks. If it does, we iterate over the tasks and print them out, showing their indices, titles, descriptions, and completion statuses. While Kotlin lists are indexed from 0, most humans think of the first item in a list as item 1, so we add 1 to each index before printing it ❶. If the task list is empty, we print a simple message indicating this.

Marking a Task as Done

When the user selects option `"3"` to mark a task as done, the code block under `"3" ->` is executed, as shown here:

```
when (readln()) {
--snip--
    "3" -> {
        taskManager.listTasks()
      ❶ if (taskManager.taskList.size <= 0) {
            continue
        } else {
            print("\nEnter the task number to mark as done: ")
          ❷ val taskNumber =
                readIndex(taskManager.taskList.size)
            if (taskNumber != null) {
                taskManager.markTaskAsDone(taskNumber -1)
            }
        }
    }
```

Within this code block, we first call the `listTasks()` method to display the current list of tasks. Then, if `taskList` is found to be empty ❶, the program will continue, meaning the remaining code will be skipped, the overarching `while` loop will restart, and the user will be presented with the menu options again. Otherwise, the user is prompted to select a task by its index, from the displayed list of tasks. We process the user input using the `readIndex()` function ❷, which validates the data as follows:

```
fun readIndex(taskListSize: Int): Int? {
    val input = readln()
  ❶ if (input.isBlank()) {
        println("Invalid input. Please enter a valid task number.")
        return null
    }

  ❷ val taskNumber = input.toIntOrNull()
    if (taskNumber != null && taskNumber >= 1 &&
                          taskNumber <= taskListSize) {
        return taskNumber
    } else {
        println("Invalid task number. Please enter a valid task number.")
        return null
    }
}
```

In this code, we initially read a line of text from the console. If the input is empty ❶, we display an `Invalid input` message, and a `null` value is returned. This `null` return value will result in no task being marked as done.

If the input isn't empty, we convert it to the `IntOrNull` type ❷. Then we perform further checks to ensure that the `Int` value is greater than or equal to `1` and less than or equal to the size of the `taskList` (which was passed as an argument to the function). If these conditions are met, the user's input value is returned; otherwise, we return `null`, which again will skip the remainder of the option `"3"` code.

Returning to the `when` expression's `"3"` branch, if the user input is valid and not `null`, we call the `markTaskAsDone()` method from the `TaskManager` class, passing in `taskNumber - 1` as an argument (remember, we added 1 to each task's index number before displaying the tasks to the user). The method is defined here:

```
fun markTaskAsDone(taskIndex: Int) {
  ❶ if (taskIndex in taskList.indices) {
        taskList[taskIndex].status = "done"
    } else {
        println("Invalid task index. Task not found.")
    }
}
```

We verify whether the `taskIndex` parameter falls within the valid range of indices in the `taskList`, which we access using the list's built-in `indices` property ❶. If `taskIndex` is within this range, we set the `status` property of the corresponding task to `"done"`. The `else` block in this method, which handles the case when the `taskIndex` is out of range, is included for potential unforeseen circumstances, even though it isn't strictly necessary since the `readIndex()` function has already verified that the chosen index is within the valid range of indices.

Deleting a Task

When the user selects option `"4"` to delete a task, the code block under `"4" ->` in the `when` expression is executed, as shown here:

```
when (readln()) {
--snip--
    "4" -> {
        taskManager.listTasks()
        if (taskManager.taskList.size <= 0) {
            continue
        } else {
            print("\nEnter the task number to be deleted: ")
            val taskNumber =
                readIndex(taskManager.taskList.size)
            if (taskNumber != null) {
                taskManager.deleteTask(taskNumber - 1)
            }
        }
    }
```

This code block is nearly identical to the one for option `"3"`: we display the task list, skip the remainder of the code if the list is empty, and otherwise take in a task number from the user with the `readIndex()` function. The difference is that for a valid, nonnull input, we call the `deleteTask()` method from the `TaskManager` class rather than the `markTaskAsDone()` method. Here's the `deleteTask()` definition:

```
fun deleteTask(taskIndex: Int) {
    if (taskIndex in taskList.indices) {
        taskList.removeAt(taskIndex)
    } else {
        println("Invalid task index. Task not found.")
    }
}
```

This time, if `taskIndex` is within the valid range, we use `removeAt()` to delete the corresponding task from `taskList`.

Exiting the Program

The last available option is "5", to exit the program. This triggers the "5" -> branch of the when expression, which terminates the while loop and returns the program flow to outside the loop:

```
when (readln()) {
--snip--
    "5" -> break   // breaks the while loop
```

As no more code remains to execute in the main() function after the while loop, disrupting the loop results in the program terminating normally.

The Result

Try launching and experimenting with the task manager program for an extended period. Here's some sample output for various arbitrary choices I made while trying out the program:

```
Task Manager Menu:
1. Add Task
2. List Tasks
3. Mark Task as done
4. Delete Task
5. Exit
Enter your choice (1-5): 1

Enter task title: Task 1
Enter task description: Reply to Nathan's email

Task Manager Menu:
1. Add Task
2. List Tasks
3. Mark Task as done
4. Delete Task
5. Exit
Enter your choice (1-5): 1

Enter task title: Task 2
Enter task description: Complete Chapter 2 by this weekend

Task Manager Menu:
1. Add Task
2. List Tasks
3. Mark Task as done
4. Delete Task
5. Exit
Enter your choice (1-5): 2

Tasks:
1. Task 1 - Reply to Nathan's email - not done
2. Task 2 - Complete Chapter 2 by this weekend - not done
```

The output you'll see will likely be different because your choices will differ from mine. Despite its somewhat limited capabilities, this program successfully integrates some of the fundamental features found in real task management tools. We've accomplished all of this with just around 110 lines of Kotlin code, while using structures like mutable lists and classes with their own properties and methods to keep the code organized.

EXERCISE

Try expanding the features of the task manager application we developed in Project 2 to turn it into a program that you can use to manage your own tasks. Here are some suggested improvements to try:

- The main limitation of the task manager program is that it doesn't save the list of tasks for future use. Once you exit the program, all information is lost. Use the file input/output methods we discussed in Chapter 1 to add a feature that allows the user to save the data before exiting and reload the data the next time they launch the program. You can also allow the user to choose which saved datafile to use, which would allow multiple users to save and work with their own files.
- Add more attributes to the `Task` data class so you can store additional information about each task. Extra attributes might include a due date, a task priority level, and a task reminder. The last feature can be added by changing the value of the `status` property to `"0verdue"` when the due date has passed (based on the current date).
- Add further functionalities to the program. Among the many possibilities, you can try adding the following: the ability to edit a task's description; the ability to sort tasks based on priority, due date, or status (for example, not started, in progress, or done); and the ability to search and filter tasks using keywords in the title or description.

Adding these features won't just make the program more useful; it will also hugely improve your coding skills in Kotlin and strengthen your ability to plan and implement a complete project with multiple interactive components.

Summary

In this chapter, we explored essential aspects of data manipulation and object-oriented programming. We began with arrays, which store values of a specific type or its subtypes. Arrays are rigid in size but can have their values modified. Lists, in contrast, are immutable in both content and size, though mutable lists offer flexibility when needed. Lists are a type of collection, along with sets and maps.

We ventured into the world of user-defined classes, which store data in the form of properties, along with methods for manipulating that data. We saw how encapsulation safeguards data within a class, while inheritance and polymorphism enable code reuse and modular design. We also covered topics like abstract classes, data classes, interfaces, and enum classes, each of which has distinct roles and advantages. For example, an abstract class provides a higher-level framework (superclass) for a group of related classes, whereas an interface enforces a consistent implementation of methods and properties across all inheriting types.

You learned these concepts through examples and exercises, reinforcing your understanding. The chapter culminated in a practical project in which you studied and then transformed a text-based task manager into a versatile tool (assuming you completed the exercise).

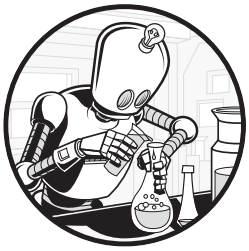
This chapter, along with Chapter 1 on Kotlin basics, equips you with the essential concepts and foundational knowledge necessary to begin an exciting journey into the world of Kotlin applications. You're now well prepared to explore a wide array of fun and progressively complex challenges in the fields of mathematics, science, modeling, algorithms, and optimization. First, though, we'll explore the basics of data visualization with JavaFX, a tool we'll use in many of our upcoming projects.

Resource

Kotlin. "Kotlin Docs." (The official Kotlin documentation.) Accessed June 15, 2024. *https://kotlinlang.org/docs/home.html*.

3

VISUALIZING WITH JAVAFX



Data visualization is the art of presenting complex data in a visually accessible format, allowing for quick and effective understanding. Through charts, graphs, maps, and other graphical representations, data visualization not only simplifies data interpretation but also uncovers patterns, trends, and insights that might otherwise be overlooked.

By transforming raw data into meaningful and actionable knowledge, visualization plays a pivotal role in decision-making across various domains, such as business analytics and scientific research. Another facet of data visualization involves the creation and presentation of intricate objects and patterns on a screen, often incorporating dynamic or moving elements within simulations or optimization processes.

In this chapter, we'll explore creating data visualization and other output with Kotlin code. We'll focus on the JavaFX library, allowing us to build standard charts, free-form drawings, and even animations. The

programming patterns we cover in this chapter will pay dividends in later chapters when we work on projects with more elaborate visual components.

Data Visualization Tools for Kotlin

Several data visualization tools, both commercial and free, are available to run with Kotlin on the JVM. These tools make it possible to create interactive plots and charts and, in some cases, free-form drawings and fully featured user interfaces (UIs). They offer a variety of features and customization options to suit your needs. Here's an overview of some of the visualization tools available:

Lets-Plot

A multiplatform plotting library for Kotlin that can be used to create interactive plots and charts. It's primarily used to access graphics features within a Jupyter Notebook, a web application that facilitates the creation and sharing of documents containing live code, narrative text, and visualizations. You can access Lets-Plot's graphics library through application programming interface (API) calls with predefined syntax. Lets-Plot doesn't have a free-form drawing or sketching tool comparable to Tkinter in Python or the `Canvas` class in JavaFX.

Plotly

Another tool for creating interactive plots and charts for Kotlin applications. Plotly works on various platforms, including the JVM, JavaScript, and Python. Plotly is user friendly, has a simple API, and offers many customization options. Like Lets-Plot, it lacks a free-form drawing tool.

Jetpack Compose

A modern UI toolkit for building native Android, desktop, and web applications using a single codebase. While Jetpack Compose doesn't have built-in support for charting, third-party libraries are available for creating interactive plots and charts. Jetpack Compose provides a canvas API that can be used to draw custom graphics and shapes.

JavaFX

An open source framework that lets you use Java to create applications for desktop, mobile, and embedded systems. It results from a collaborative effort by many individuals and companies that aim to provide a modern, efficient, and fully featured toolkit for developing rich client applications. You can use JavaFX to create user interfaces and interactive visualizations, as well as various charts such as line charts, bar charts, pie charts, and scatterplots. JavaFX works well with Kotlin because of its compatibility with Java.

In this book, we'll use JavaFX as the graphics library of choice. This is because it's a mature and feature-rich library that's well documented

and can seamlessly integrate with both Java and Kotlin applications. Most important, it has built-in features for charting and free-form drawing, including pixel-level manipulation of the display screen. These features will be useful in some projects we'll work on later in the book.

An Overview of JavaFX

JavaFX was first introduced by Sun Microsystems in 2007 as a modern replacement for the aging Java Swing framework. It marked a significant step forward in Java's capabilities for creating GUIs and multimedia-rich applications. Initially, JavaFX was shipped as part of the Java Development Kit (JDK). However, Oracle, the company that acquired JavaFX from Sun Microsystems, announced in 2018 that JavaFX would be open sourced and moved to the OpenFX project. The same year, JavaFX 11 was released as a stand-alone framework, decoupled from the JDK.

JavaFX has evolved over the years, thanks to numerous updates and improvements from its developers. It continues to thrive as an open source project under the stewardship of the OpenJFX community and is included as a standard library in many Java distributions.

Key Functionalities

JavaFX is a comprehensive toolkit for building cross-platform applications. While this chapter focuses on its charting and drawing features, JavaFX offers many other features that you can use in your projects. Here's a quick overview of its key functionalities:

UI development

JavaFX enables the developer to simplify and enhance the development of visually rich and interactive user interfaces for desktop, web, and mobile applications. It provides a wide variety of UI controls, layouts, and styles, allowing the creation of visually appealing and highly customizable interfaces.

Cross-platform compatibility

JavaFX is designed to create applications that can run on various platforms, including Windows, macOS, Linux, and mobile devices, without major modifications. This cross-platform compatibility reduces development effort and allows for broader application distribution.

Charts and data visualization

JavaFX includes built-in support for creating various charts and graphs, making it a preferred choice for data visualization applications.

3D graphics

JavaFX provides versatile 3D graphics capabilities for developing mathematical and scientific applications that require 3D visualizations.

Rich media support

JavaFX is known for its robust multimedia support, making it suitable for applications that require video, audio, animations, and 2D or 3D graphics.

High performance

JavaFX provides hardware acceleration and optimizations for improved rendering performance, making it suitable for applications demanding smooth animations and responsive interfaces.

Integration with Java and Kotlin

JavaFX seamlessly integrates with the Java programming language, leveraging the robustness, security, and ecosystem of Java. Since Kotlin is fully interoperable with Java, JavaFX is a natural choice for developing desktop and web-based applications in Kotlin.

For more information about JavaFX features, see the project website at *https://openjfx.io*.

Setup

If you've followed the steps for installing IntelliJ IDEA and the Azul Zulu JDK as described in the appendix, you're all set to start using JavaFX with Kotlin. You can access JavaFX features just as you've been accessing Kotlin features from the IDE. While the support for Kotlin is integrated within the IDE itself, the access to JavaFX is gained through the installed JDK.

If you haven't followed the instructions in the appendix, I recommend using a JDK with JavaFX prepackaged so you can avoid the extra steps needed to link the library to your code. For example, Azul JDK FX and Liberica Full JDK are well-known distributions with integrated JavaFX support. Alternatively, you can download and install JavaFX separately from the OpenJFX website. There you'll find detailed instructions on how to set up JavaFX for various operating systems, such as Windows, macOS, and Linux, including how to access JavaFX from your IDE.

Once both Kotlin and JavaFX are accessible from the IDE, you can focus on creating new JavaFX applications in Kotlin. The process is similar to creating regular Kotlin programs, except that you need to add some boilerplate JavaFX code, which I'll explain in detail next.

Project 3: Build "Hello, World!" in JavaFX

In this project, I'll walk you through the process of building a simple "Hello, world!" application using JavaFX and Kotlin. This will serve as the foundation for constructing other applications that leverage JavaFX's charting and visualization features. First, follow these steps to create a new JavaFX-enabled application from scratch:

1. Open IntelliJ IDEA and create a new Kotlin project by navigating to **File ▸ New ▸ Project**. You'll be taken to the project setup window shown in Figure 3-1.

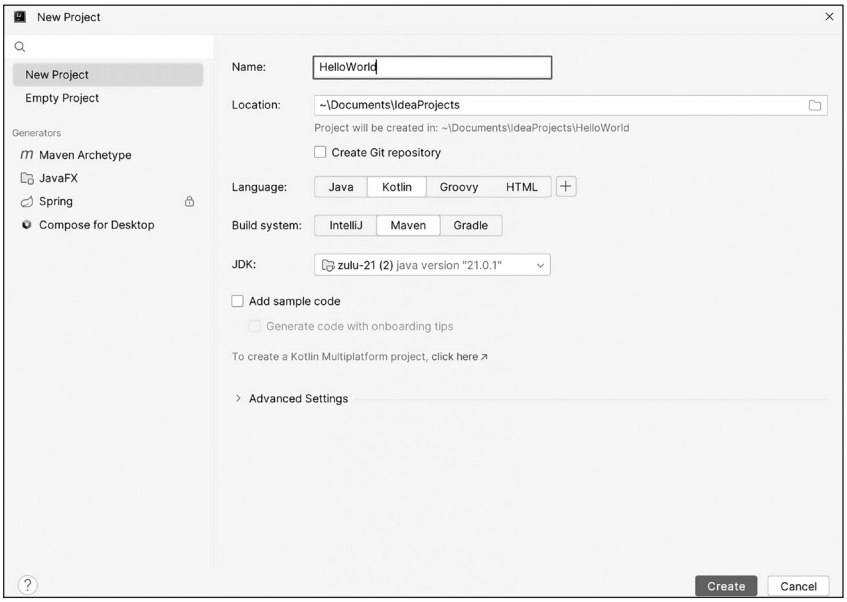


Figure 3-1: Configuration options for JavaFX-based Kotlin projects

2. Name the project *HelloWorld* and note its location. Then select **Kotlin** as the language and **Maven** as the build system and choose a JDK that includes integrated support for JavaFX. (If you've followed the instructions in the appendix, that'll be the latest long-term support version of Azul Zulu JDK FX.) For all JavaFX-based projects covered in this book, we'll use these same settings (apart from the project name), as they eliminate the need for additional steps, such as manually linking a separate JavaFX library to the project.

NOTE *You may notice that the left-hand panel of IntelliJ IDEA has an option to autogenerate a new JavaFX project, complete with a default "Hello!" window. This preconfigured setup comes with objects and files that may not be necessary for each project, so I recommend creating a project from scratch and including only the required code and objects, as outlined here.*

3. Click **Create** to create a new project with the correct configuration.
4. In the project panel on the left side of the IDE, expand the project tree by navigating to **HelloWorld ▸ src ▸ main ▸ kotlin**.

5. Right-click the *kotlin* folder and choose **New ▸ Kotlin Class/File**. Then choose **File** and enter **HelloWorld** as the name of the Kotlin file. (For other projects, you can choose any other meaningful name.) Note that including the file extension, *.kt*, is optional. At this stage, the project screen should look like Figure 3-2.

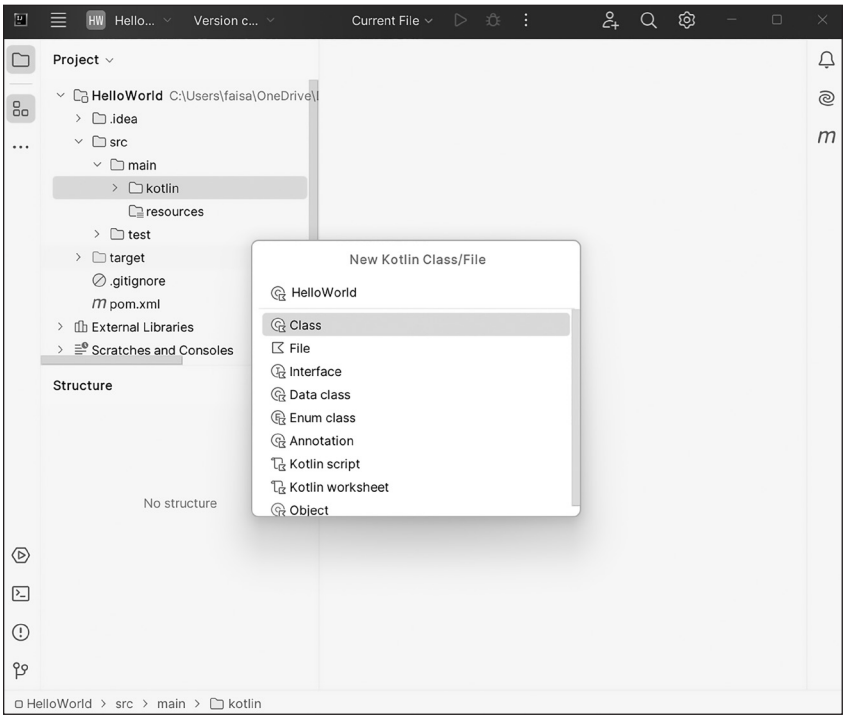


Figure 3-2: Creating the Kotlin file for a JavaFX application

6. Once you've typed in the Kotlin filename, press ENTER to create the file, which also takes you to the code editor window.

With that, we're ready to start coding our basic JavaFX application. For more complex projects, you may have to create additional classes or files. You can do so by using the same method outlined here.

The Code

The center of any JavaFX project is the `Application` class, an abstract class built into JavaFX that provides a framework for managing a graphical application. It features an abstract `start()` method that serves as the application's entry point, much like the `main()` function is the point of entry for a console-based program. The project must feature a new class—what we'll call `HelloWorld`, in this case—that extends the `Application` class and overrides the `start()` method. In the overridden method, you write the code for setting up and configuring the graphics window for data visualization and other aspects of your application. The project still needs a separate `main()`

function as well, but the sole purpose of that function is to call the built-in `launch()` method of the `Application` class, which in turn launches the JavaFX application and calls the `start()` method.

With that in mind, here's the code to create a simple graphical application that displays a "Hello, world!" message to the user:

```
import javafx.application.Application
import javafx.geometry.Pos
import javafx.stage.Stage
import javafx.scene.Scene
import javafx.scene.layout.VBox
import javafx.scene.text.Font
import javafx.scene.text.Text

class HelloWorld : Application() {
  ❶ override fun start(primaryStage: Stage) {
        primaryStage.title = "Primary Stage"
        val text = Text("Hello, world!")
        text.font = Font.font("Verdana", 20.0)

        val vbx = VBox(text)
        vbx.alignment = Pos.CENTER
        val scene = Scene(vbx, 300.0, 300.0)
        primaryStage.scene = scene
        primaryStage.show()
    }
}

fun main() {
  ❷ Application.launch(HelloWorld::class.java)
}
```

We first import several JavaFX classes, including `Application`, `Pos`, `Stage`, `Scene`, `VBox`, `Font`, and `Text`. These are required to create and manage the graphical elements for the "Hello, world!" application. Any JavaFX application will begin with a long import block like this, though the specifics will vary depending on the application's functionality.

Next, we declare the `HelloWorld` class to inherit from the `Application` class, which, as mentioned, is standard practice in JavaFX applications. Inside the class, we override the `start()` method ❶ to define the entry point for the application that will be executed when the program is launched. The `start()` method takes an argument called `primaryStage`, an object of the JavaFX `Stage` class, representing the application's primary viewing window. Within the method, we do the following:

- Use `primaryStage.title` to set the title of the application window to `"Primary Stage"`.
- Create a `Text` object named `text` containing the message `"Hello, world!"` with a custom font of Verdana size 20.0.
- Create a `VBox` object named `vbx` and add the `text` object to it. In JavaFX, a `VBox` is a layout container that arranges its contents (in this case, `text`)

in a vertical column. We set its `alignment` property to `Pos.CENTER` to ensure the text will be centered within the window.

- Create a `Scene` object named `scene` with the `vbx` attached to it and a size of 300.0×300.0 pixels. (The dimensions have to be given in the floating-point format as they're of type `Double`.)
- Assign the `Scene` object to the `primaryStage` parameter's `scene` property, which adds the scene to the viewing window.
- Call the `show()` method of the `primaryStage` object to display the JavaFX application window.

We'll discuss the significance of these JavaFX constructs in more detail shortly.

The final segment of the program is the `main()` function, which consists of just a single line of code ❷. As mentioned earlier, its sole purpose is to call the `launch()` method of the `Application` class, which activates the `start()` method of the `HelloWorld` application. The parameter inside the `launch()` method, `HelloWorld::class.java`, specifies the Java class corresponding to the `HelloWorld` Kotlin class that serves as the entry point for the JavaFX application.

HOW TO HIDE ::CLASS.JAVA

Some Kotlin developers may prefer not to use constructs such as `::class.java` inside the `main()` function. We can avoid doing that by creating an inline function in a second file inside the *kotlin* folder (where other Kotlin files are located). Let's name this new file *Inline.kt* and add the following lines to this file:

```
import javafx.application.Application

inline fun <reified T : Application> runApplication() {
    Application.launch(T::class.java)
}
```

Here, `<reified T: Application>` introduces a type parameter `T` whose type is determined at runtime based on the actual argument passed to the function. Now we can call the inline function from the `main()` function as follows:

```
fun main() {
    runApplication<HelloWorld>()
}
```

You can use this trick if you think this will make the main body of the code look nicer!

The Result

Try running the application in IntelliJ IDEA. The output window shown in Figure 3-3 should pop up.

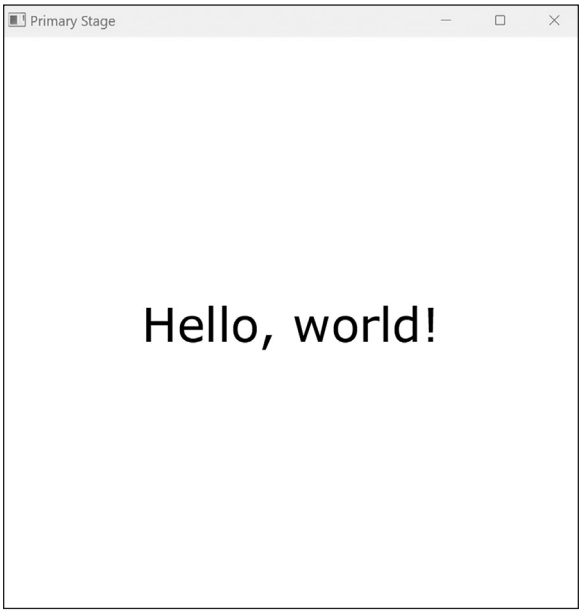


Figure 3-3: The minimal "Hello, world!" application in Kotlin and JavaFX

Let's highlight a few details in this output. First, the window has a title, "Primary Stage," which is what the `title` property of `primaryStage` was set to. Second, the text "Hello, world!" is neatly centered inside the window. This was achieved by setting the alignment property of `vbx` to `Pos.CENTER` so that its contents would be aligned centrally within its boundaries. Third, the initial size of the window was set by specifying the width and height of the scene. However, you can change the window size by dragging any of its boundaries. Finally, the application continues to run in the background as long as the display window is open. Once you close the window, the application terminates normally.

This code will be the foundational template for all our JavaFX-based applications, requiring only minor modifications. For example, you'll be replacing the `Text` object with a `LineChart` or a `Canvas` object to adapt the code for creating a graph or a free-form drawing, respectively. JavaFX is a feature-rich framework, and I highly recommend exploring the resources listed at the end of this chapter to gain a thorough understanding of its core components and their relationships. For now, I'll provide a brief overview of key features that we'll be using in our upcoming JavaFX-based projects.

The JavaFX Object Hierarchy

JavaFX uses a hierarchy of objects to represent the structure of a GUI and the elements that make up a visual display. At the top of this hierarchy is the `Stage`, which serves as the primary window for an application. The `Stage` is a container for one or more `Scene` objects, and each `Scene` is a container for various graphical elements, including other layout containers, controls, and shapes. All these components are considered *nodes* in the JavaFX object hierarchy.

We've already worked with several of these objects in our "Hello, world!" application. Let's now take a closer look at what they are and how they relate to each other. Understanding and manipulating JavaFX's object hierarchy is fundamental to creating visually appealing and interactive applications.

The Stage

The `Stage` is the top-level container in a JavaFX application. It represents the application window, complete with its title bar, borders, and any native OS-level components. The `Stage` serves as the primary object for creating a graphical window. You create a `Stage` and set its properties, such as the window title and dimensions:

```
val myStage = Stage()
myStage.title  = "This is a secondary window"
myStage.width  = 800.0 // in pixels
myStage.height = 600.0 // in pixels
```

In JavaFX applications, at least one `Stage` object is essential. This primary `Stage` is created when you override the `start()` method. However, in this example, we're explicitly creating an additional `Stage` object, which will appear as a second graphical window within our application. We achieve this by invoking the `Stage()` constructor and assigning it to a new class member, `myStage`.

Scenes

A `Scene` represents a single, self-contained GUI component within the `Stage`. It acts as a container for all the visible elements in a specific part of an application, such as the main menu, a settings screen, or a game level. Typically, the root node of a `Scene` is set to a *layout container*, such as a `Group`, `Pane`, `VBox`, or `HBox` object, which in turn contains other nodes, such as buttons, labels, or shapes. Here's an example of creating a `Scene` and setting its root node at the same time:

```
val root = Group()
val scene = Scene(root, 800.0, 600.0)
```

We pass `root` as the first arguments to the `Scene` constructor. This mirrors how we assigned a `VBox` as the root node of our "Hello, world!" application's `Scene` object.

Layout Containers

Layout containers like `Group` and `VBox` often serve as the root or parent node for a `Scene`. They're used to position and size other *child nodes* displayed in the application window. In this way, layout containers are essential for designing the structure and layout of a user interface in JavaFX. In upcoming projects, we'll use a few of these containers, so it's worth taking a closer look at their properties to see which is most appropriate for different applications. Sometimes more than one type can be equally suitable.

Group

A `Group` is a container that groups other nodes together without providing any layout capabilities. This can be useful when you want to apply a transformation or effect to a set of shapes collectively. For example, you can group multiple shapes together and apply a rotation or scaling effect to the entire group.

Pane

A `Pane` is a layout container that serves as a building block for organizing and positioning nodes in a user interface. It's not specialized for a particular layout, which makes it a versatile choice for various UI design scenarios. JavaFX provides several subclasses of `Pane` that you can choose from based on the requirements of your application. These subclasses include:

`FlowPane` A container for arranging nodes in a fluid manner, automatically wrapping when the container's boundaries are reached. By default, a `FlowPane` is horizontal, meaning it lays out nodes in rows, starting from the left boundary. When the right boundary is reached, nodes wrap (move) to the next row.

`StackPane` A container for stacking nodes on top of one another, such that one node blocks another. This enables creative layering of elements. All child nodes are automatically centered both horizontally and vertically within the available space.

`BorderPane` A container for positioning nodes in a structured manner in the top, bottom, left, right, and center positions.

`AnchorPane` Similar to `BorderPane`, except it allows nodes to attach to the top, bottom, left, and right edges of the pane.

`GridPane` A container for arranging nodes in a grid of rows and columns.

Panes are essential components of JavaFX applications, as they provide the layout and structure for the user interface. By using panes, you can create flexible and responsive UIs that can adapt to different screen sizes and resolutions.

VBox and HBox

`VBox` and `HBox` are layout containers designed to organize their child nodes neatly. `VBox` arranges its children in a vertical column, while `HBox` places them in a horizontal line.

Child Nodes

Child nodes are the fundamental building blocks of a JavaFX application. They represent specific visual elements, including shapes (for example, rectangles or circles), images, text, and interactive controls like buttons or text fields. The different types of chart objects that we'll soon create are also considered nodes in JavaFX. By adding, removing, and modifying nodes within the application's hierarchy of objects, you can design intuitive and engaging user interfaces.

You can add child nodes to parent nodes such as a `Group`, `Pane`, `VBox`, or `HBox` to create complex hierarchies of UI components. Here's an example:

```
// Create a rectangle object (child node).
val rectangle = Rectangle(100.0, 100.0, Color.BLUE)

// Pass the child node to its parent node.
val root = Group(rectangle)
```

We create a blue `Rectangle` object that's 100×100 pixels. We then assign the rectangle as a child node of a `Group` object called `root`. In this case, we assign the child node by passing it as an argument to the parent node's constructor, but children don't always have to be assigned immediately upon the creation of the parent. You can also assign a child node to a parent later by using the `children.add()` method:

```
// Create a Button object with the text "Click me".
val button = Button("Click me")

// Create a Pane object.
val pane = Pane()

// Add the Button as a child node to the Pane.
pane.children.add(button)
```

In this example, we create a `Button` object (`button`) and a `Pane` object (`pane`). Then we add the button as a child node of the pane by using the `children.add()` method. This method takes a single node as an argument and appends it to the list of children of the pane. We could also add or pass multiple children to the pane by using the `children.addAll()` method.

Creating JavaFX Charts

JavaFX provides a set of built-in features for creating visually appealing charts. You can feed a dataset into a chart for visualization and customize

the result by changing the colors, fonts, and other properties. Here are the basic steps for creating a chart with JavaFX and Kotlin:

1. Create objects to represent the x-axis and y-axis of the chart. These can be of two types: `NumberAxis` and `CategoryAxis`. Use the former for visualizing numerical data and the latter for string-type data.
2. Create an instance of a class representing the type of chart you want to design, such as `LineChart`, `BarChart`, `ScatterChart`, `PieChart`, or `BubbleChart`. You pass the objects created in step 1 as arguments to the chart's constructor.
3. Create one or more `Series` objects to represent the data series you want to display in the chart.
4. Add data points to each series by using the `data.add()` method for a single data point or the `data.addAll()` method for several data points at once.
5. Add each series to the chart also by using the `data.add()` method.
6. Create a new `Scene` object and add the chart to it. Although you can directly assign a chart object as the root of a scene, you may want to first assign the chart as a child of a layout container and make that container the root of the scene. This way, you can have better control over the result.
7. Assign the `Scene` to the `Stage` object where the chart should be displayed. Don't forget to display the `Stage` by using its `show()` method!

These steps are generic and can be used to create any of the chart types available in JavaFX. We'll now look at two representative examples of creating different chart objects that use actual data: a bar chart and a line chart.

Project 4: Visualize Data as a Bar Chart

In this project, we'll generate a bar chart to visualize the monthly sales data for a fictitious company named ABC & Co. over the first quarter of a year. The chart will feature the months of January through March along the x-axis, with vertical bars illustrating the sales total for each month along the y-axis. The code will follow the same general outline we used for the simple "Hello, world!" application, with modifications to implement the charting steps we discussed.

The Code

Here's the code for creating a bar chart:

```
import javafx.application.Application
import javafx.geometry.Side
import javafx.scene.Scene
import javafx.scene.chart.CategoryAxis
import javafx.scene.chart.BarChart
```

```
import javafx.scene.chart.NumberAxis
import javafx.scene.chart.XYChart
import javafx.stage.Stage

class BarChartExample : Application() {
    override fun start(primaryStage: Stage) {
      ❶ primaryStage.title = "Bar Chart Example"

      ❷ // Create XYAxis objects and set their properties.
        val xAxis = CategoryAxis()
        val yAxis = NumberAxis()
        xAxis.label = "Months"
        yAxis.label = "Sales in thousands of dollars"

      ❸ // Create BarChart object and set its properties.
        val barChart = BarChart(xAxis, yAxis)
        barChart.title = "Monthly Sales"
        barChart.legendSide = Side.TOP

      ❹ // Create Series, populate with data, and assign to chart.
        val dataSeries = XYChart.Series<String, Number>()
        dataSeries.name = "Q1 Data for ABC & Co."
      ❺ getData(dataSeries)
        barChart.data.add(dataSeries)

        val scene = Scene(barChart, 400.0, 400.0)
        primaryStage.scene = scene
        primaryStage.show()
    }
}

fun main() {
    Application.launch(BarChartExample::class.java)
}
```

We begin by importing the necessary JavaFX classes. In addition to the general `Application`, `Scene`, and `Stage` classes common to any JavaFX application, we import some classes specific to charting, such as `Side`, `CategoryAxis`, `BarChart`, `NumberAxis`, and `XYChart`. After the import block, we declare the `BarChartExample` class, which once again extends JavaFX's abstract `Application` class. Much like the "Hello, world!" application, we override the `start()` method with a custom definition. This time we use the method to create a bar chart.

The `start()` method is organized into several blocks. In the first block, we set the title of `primaryStage` (the `Stage` object passed into the `start()`

method) to "Bar Chart Example" ❶. In the second block ❷, we create two objects representing the chart's x- and y-axes. We use the CategoryAxis class for the x-axis, where the data points will be months of the year (strings), and the NumberAxis class for the y-axis, where the data points will be numeric sales totals. We also use the label property of each axis object to give the axis a descriptive label. In the third block ❸, we create the BarChart object, passing the xAxis and yAxis objects as arguments. We also assign the chart a title and position its legend at the top of the chart. The legend explains the significance of the colors or patterns used to visualize the data.

In the fourth code block ❹, we create a series called dataSeries of type XYChart.Series. Each data point of this series will have a pair of elements of type <String, Number>, representing a month and its corresponding sales total in thousands of dollars. We set the name for the series; this name will appear in the chart's legend. Then, to keep the start() method itself concise, we call a custom getData() function ❺ to add data points to the series before adding the series to the BarChart object to plot the data on the chart. Here's a look at the getData() function:

```
fun getData(dataSeries: XYChart.Series<String, Number>) {
    dataSeries.data.addAll(
        XYChart.Data("Jan", 150),
        XYChart.Data("Feb", 100),
        XYChart.Data("Mar", 225)
    )
}
```

This function's sole job is to add the data points to the dataSeries object in bulk by using the data.addAll() method. Each data point is an instance of the XYChart.Data class, which encapsulates the x- and y-axis values of a data point together in one container. In particular, each data point has a string month abbreviation for the x-value and an integer sales total for the y-value. We could also provide the sales totals as floating-point values; JavaFX recognizes both forms as numbers.

The remaining part of the application class is standard JavaFX template code, virtually identical to that of our "Hello, world!" application. We construct a Scene object, assigning barChart as its root node. Then we assign the Scene object to primaryStage and call the latter's show() method, which displays the bar chart we've created on the screen.

The Result

If you run this code, the resulting bar chart should look like Figure 3-4.

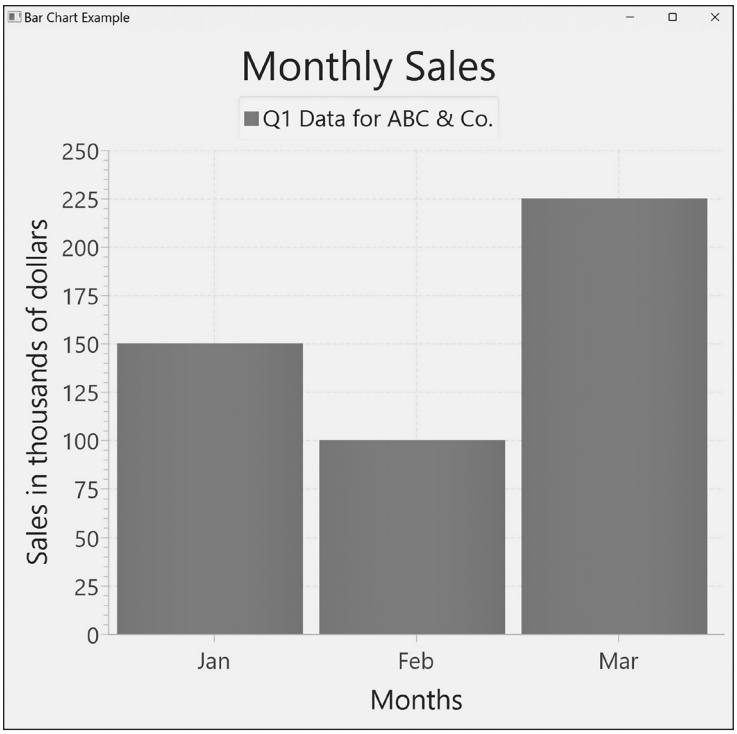


Figure 3-4: A bar chart drawn using JavaFX

Notice how JavaFX has automatically generated a bar with the correct height for each data point, with the months spaced evenly along the x-axis and ticks labeled in increments of 25 along the y-axis, stopping just past the maximum value displayed in the chart. You can also verify that the chart correctly displays the properties we set explicitly, such as the title ("Monthly Sales"), the x- and y-axes labels ("Months" and "Sales in thousands of dollars"), and the legend ("Q1 Data for ABC & Co."), which is positioned correctly at the top.

Project 5: Create a Multiseries Line Chart

In this next project, we'll create a line chart in JavaFX that displays the average heights of males and females at different ages. In a line chart, both axes are numeric, and their properties can be adjusted—for example, by setting the tick intervals and bounds. We'll visualize age on the x-axis and height (in inches) on the y-axis. A key feature of the chart is that it will have two datasets, one for males and one for females, each in its own `Series` object. JavaFX will automatically plot the datasets separately, giving each a different colored line.

The Code

Our code will follow the same structure as the code for the bar chart, with modifications to instead make a line chart with two series. The program can be easily extended to chart three or more series. Here's the code:

```
import javafx.application.Application
import javafx.geometry.Side
import javafx.scene.Scene
import javafx.scene.chart.LineChart
import javafx.scene.chart.NumberAxis
import javafx.scene.chart.XYChart
import javafx.stage.Stage

class LineChartExample : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title = "Multiseries Line Chart Example"

        // Create XYAxis objects and set their properties.
        val xAxis = NumberAxis()
        xAxis.label = "Age"
        val yAxis = NumberAxis()
        yAxis.label = "Height (inches)"

      ❶ // Adjust tick interval and lower/upper bounds.
        xAxis.isAutoRanging = false
        xAxis.tickUnit = 5.0    // custom tick interval
        xAxis.lowerBound = 0.0  // minimum value for x-axis
        xAxis.upperBound = 35.0 // maximum value for x-axis

        yAxis.isAutoRanging = false
        yAxis.lowerBound = 20.0 // minimum value for y-axis
        yAxis.upperBound = 75.0 // maximum value for y-axis

        // Create LineChart object and set its properties.
      ❷ val lineChart = LineChart(xAxis, yAxis)
        lineChart.title = "Average Heights at Different Ages"
        lineChart.legendSide = Side.TOP

        // Create Series, populate with data, and assign to chart.
        val maleData = XYChart.Series<Number, Number>()
        maleData.name = "Male"
      ❸ getMaleData(maleData)
        val femaleData = XYChart.Series<Number, Number>()
        femaleData.name = "Female"
      ❹ getFemaleData(femaleData)

      ❺ lineChart.data.addAll(maleData, femaleData)

        val scene = Scene(lineChart, 800.0, 600.0)
        primaryStage.scene = scene
        primaryStage.show()
    }
}
```

```
fun main() {
    Application.launch(LineChartExample::class.java)
}
```

We begin with a typical JavaFX import block, this time importing the `LineChart` class instead of `BarChart`. Then we declare an application class called `LineChartExample` and override its `start()` method as usual. Inside the method, we first set the `primaryStage` title to `"Average Heights at Different Ages"`, create the x- and y-axes, and set their labels. We use the `NumberAxis` class for both axes, since ages and heights are both numerical data.

In the next code block ❶, we further customize the axes. For both axes, we turn off the autoranging capability by setting the `isAutoRanging` property to `false` (this is necessary for the subsequent changes to take effect). Then we set the lower and upper bounds for the axes: ages `0.0` to `35.0` on the x-axis and heights `20.0` to `75.0` on the y-axis. We also set a tick interval of `5.0` on the x-axis. We let JavaFX work out all these settings automatically for the bar chart, but here we exert more control to remove the unnecessary autogenerated space and better position the two series in the chart window. In particular, JavaFX defaults to a lower bound of `0.0` for any numeric axis (when x- and y-values are positive), which in this case would create a lot of extra space near the bottom of the chart since the lowest height in the dataset is `36.0`.

The next few blocks are very similar to the bar chart example. We create a `LineChart` object ❷, set its title, and set the position of the legend. Then we create two `XYChart.Series` objects, `maleData` and `femaleData`, and call two helper functions, `getMaleData()` ❸ and `getFemaleData()` ❹, to populate the series with data. Here are the definitions of those functions:

```
fun getMaleData(maleData: XYChart.Series<Number, Number>) {
    maleData.data.addAll(
        XYChart.Data(5, 38.0),
        XYChart.Data(10, 50.0),
        XYChart.Data(15, 62.0),
        XYChart.Data(20, 68.0),
        XYChart.Data(30, 69.0)
    )
}

fun getFemaleData(femaleData: XYChart.Series<Number, Number>) {
    femaleData.data.addAll(
        XYChart.Data(5, 36.0),
        XYChart.Data(10, 48.0),
        XYChart.Data(15, 60.0),
        XYChart.Data(20, 64.0),
        XYChart.Data(30, 65.0)
    )
}
```

Much like the bar chart project, we use the `data.addAll()` method to add all the data points to each series in one go. As before, the x- and y-values for each data point are packaged into an `XYChart.Data` object.

Returning to the main code, once both series (`maleData` and `femaleData`) are populated, we add them to the `lineChart` object by using the `data.addAll()` method ❺. Finally, we assign `lineChart` to `scene` and `scene` to `primaryStage`, and call the `primaryStage` object's `show()` method to display the line chart, following our normal pattern of displaying a JavaFX visualization.

The Result

Figure 3-5 shows the line chart that results from running the code.

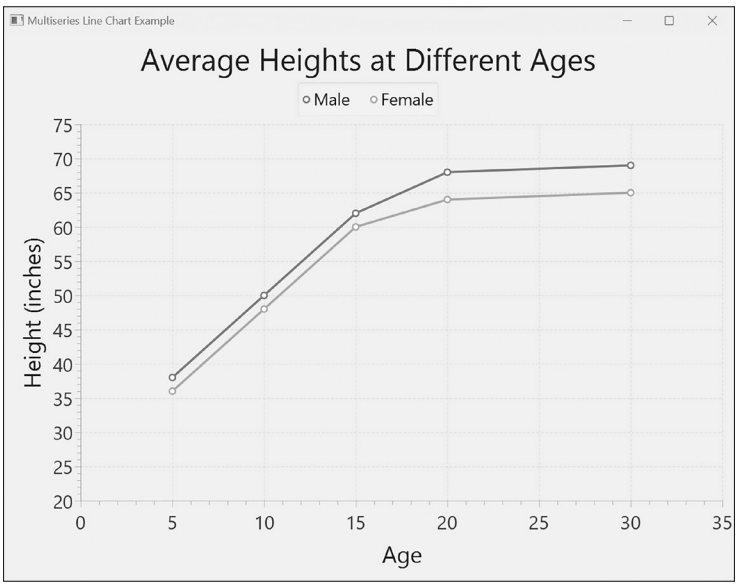


Figure 3-5: A multiseries line chart drawn using JavaFX

The line chart displays the average heights for males and females of different age groups as a set of two series plotted on the same x- and y-axes. JavaFX uses the chart's legend to distinguish one plot from the other. You can verify that the chart has been configured based on the properties we've set manually; for example, the y-axis has a lower bound of 20, and the x-axis ticks are labeled in increments of 5.

You can do a lot more customization beyond the basic settings we've adjusted in this project. I'll leave it to you to explore other options based on your additional reading and experimentation with JavaFX.

EXERCISE

Try developing a simple pie chart application using JavaFX and Kotlin. It should show the market shares of four different products as percentages. Use these data points:

- Product A: 13.0 percent
- Product B: 28.0 percent
- Product C: 35.0 percent
- Product D: 24.0 percent

Here are some tips on how to proceed:

- Be sure to import the `javafx.scene.chart.PieChart` class. You can also leave out some of the other typical imports, since pie charts don't have x- and y-axes.
- You can create a `PieChart` object without passing any arguments to the constructor:

```
val pieChart = PieChart()
```

- You don't have to create a series for the data points. Instead, you can provide the data points directly to the `PieChart` object by using its `data.addAll()` method. I recommend writing a helper function to fetch the data instead of doing it all from within the `start()` method.
- Inside the helper function, add data points to the `pieChart` object as follows:

```
pieChart.data.addAll(
    PieChart.Data("Product A", 13.0),
    --snip--
    PieChart.Data("Product D", 24.0),
)
```

Notice that the data points are supplied using a `<String, Double>` format—that is, for a given data point we specify the product name as a string and the percent share as a floating-point value.

- Experiment with turning the pie chart's data labels off (they're on by default). This can be done by setting the `labelsVisible` property of `pieChart` to `false`.
- JavaFX will assign default colors to the pie chart slices. Try changing those colors or using patterns instead of colors, since patterns are easily recognized even in grayscale images. Search for "change PieChart color in JavaFX" online for more information on how to do this.

Drawing with the Canvas

In JavaFX, a *canvas* is a region that provides a drawing surface for rendering custom 2D graphics. Whereas JavaFX's charting features handle much of the drawing automatically, a canvas allows you to write low-level code for drawing individual lines, shapes, text fields, and more. You create a canvas by instantiating the `Canvas` class. Then you draw to the canvas object by accessing its *graphics context*.

In this section, we'll look at a simple first example of working with a JavaFX canvas and explore some commonly used drawing methods. Then we'll get better acquainted with the canvas through a more elaborate drawing project.

A Simple Shape

Let's get started with the JavaFX `Canvas` class by using it to draw a simple shape. Here's the code for an application that draws a rectangle to the canvas:

```
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.stage.Stage

class CanvasExample_1 : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title = "Canvas Example"

      ❶ val canvas = Canvas(400.0, 200.0)
      ❷ val gc = canvas.getGraphicsContext2D()
        val pane = Pane(canvas)
        val scene = Scene(pane)
        primaryStage.setScene(scene)
        primaryStage.show()

      ❸ drawRectangle(gc)
    }

    fun drawRectangle(gc: GraphicsContext) {
      ❹ with(gc) {
            stroke = Color.RED
            strokeRect(100.0, 50.0, 200.0, 100.0)
        }
    }
}

fun main() {
    Application.launch(CanvasExample_1::class.java)
}
```

Inside the application class's `start()` method, we create an instance of the `Canvas` class with a size of 400×200 pixels and call it `canvas` ❶. Then we call the canvas object's `getGraphicsContext2D()` method. It returns a reference to the canvas's `GraphicsContext` object, which we store in the `gc` variable ❷. This object provides the interface for drawing to the canvas. We then follow the usual steps of assigning the canvas to a layout container (a `Pane` object), the container to a scene, and the scene to the primary stage, which we display with the `show()` method. We need the `Pane` since `Canvas` isn't a parent-type node and thus can't be passed directly to a scene. Also, by making the `Canvas` object a child node to a container such as a `Pane` or `VBox`, we can further customize its placement and size if needed.

To draw the rectangle, we call a custom `drawRectangle()` method ❸ that takes one argument, the graphics context. In the method's definition, we use Kotlin's `with` scope function to group the actions requiring access to the graphics content, `gc` ❹. This saves us from adding `gc.` to the start of each line of code, which is a big help in longer drawing methods. We set the `stroke` color of the graphics context to red (in computer graphics, a *stroke* is the outline of a geometric shape), then call JavaFX's `strokeRect()` method to draw a rectangle. The first two arguments (`100.0` and `50.0`) set the x- and y-coordinates of the rectangle's top-left corner, and the remaining arguments (`200.0` and `100.0`) define its width and height (in pixels). By default, the origin of the coordinate system (`0.0, 0.0`) is positioned at the top-left corner of the canvas.

Figure 3-6 shows the result of running this simple canvas application.

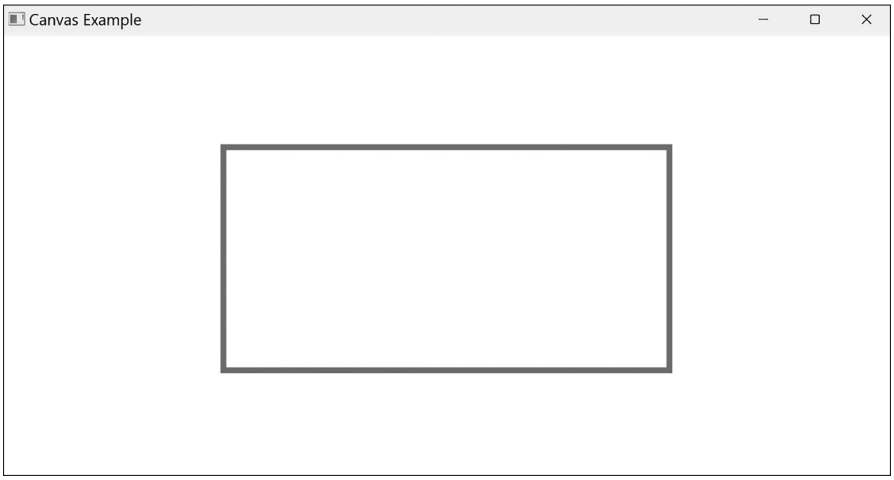


Figure 3-6: A red-outlined rectangle (shown here in gray) on a canvas

In this example, we've defined the `drawRectangle()` method within the `CanvasExample_1` class. Alternatively, we could have created it as an independent function, similar to the `getData()` functions in the bar chart and line chart examples. Both approaches are valid. When a function is generic and could be reused by other classes or modules, consider making it a standalone function. Otherwise, creating it as an internal method of a class enhances data encapsulation.

EXERCISE

Modify the rectangle-drawing code to achieve the following:

1. Write a `drawSquare()` method to draw a filled square in the middle of a canvas. You'll need to use the `gc.fill` feature to set the fill color and the `gc.fillRect()` method to draw the square.
2. Write a `drawTriangle()` method that will draw a filled triangle in the middle of a canvas. You'll again need to use the `gc.fill` feature to set the fill color, plus the `gc.fillPolygon()` method to draw the triangle. This method takes three arguments: an array of x-coordinates for the polygon's vertices, an array of corresponding y-coordinates, and an integer specifying the number of vertices. In this case, use `doubleArrayOf(x1, x2, x3)`, `doubleArrayOf(y1, y2, y3)`, and `n`, where `x1`, `x2`, and `x3` are the x-coordinates; `y1`, `y2`, and `y3` are the y-coordinates; and `n` is `3`.

Common Graphics Context Methods

The graphics context is an essential tool associated with any canvas that enables you to design 2D graphics in JavaFX. Before moving on to more advanced projects involving the canvas, let's consider some of the key features and capabilities of the `GraphicsContext` class and how to apply them in Kotlin. The most commonly used features are listed in Table 3-1 for ease of reference.

Table 3-1: Commonly Used Graphics Context Methods

Feature	Description	Usage in Kotlin

Drawing shapes	Draw various 2D shapes, such as lines, rectangles, circles, and polygons.	`gc.strokeRect(x, y, width, height)`
		`gc.fillOval(x, y, radiusX, radiusY)`
		`gc.fillPolygon(x-array, y-array, n)`

Setting colors	Set stroke (outline) and fill colors for shapes.	`gc.stroke = Color.RED` `gc.fill = Color.BLUE`
Line width	Set line width and line style.	`gc.lineWidth = 2.0` `gc.setLineDashes(dash, gap)`
Text rendering	Draw text on the canvas with specified fonts and sizes.	`gc.font = Font("Arial", 14.0)` `gc.fillText("Hello, world!", x, y)`
Image rendering	Draw images on the canvas.	`val image = Image("image.png")` `gc.drawImage(image, x, y)`
Transformations	Translate (move the origin of the coordinate system by the specified x- and y-values) and rotate (rotate the subsequent drawings by a specified angle).	`gc.translate(x, y)` `gc.rotate(angle)`

Clearing canvas	Clear the entire canvas or a specific region.	`gc.clearRect(0.0, 0.0,`
		`canvas.width, canvas.height)`
		`gc.clearRect(x, y, width, height)`

Table 3-1 shows the rich set of capabilities that the graphics context provides. For example, we can draw and fill geometric shapes such as rectangles and ovals with different colors. We can also adjust the width and style of a line. We have many options for rendering text as well. Besides shapes, we can use images of various formats (for example, JPG, PNG, or GIF) and creatively place them on the canvas. Moreover, we can move the origin of the coordinate system and rotate objects drawn on the canvas, altering their orientation relative to the current coordinate system. And finally, we can wipe the entire canvas clean, providing a fresh starting point for dynamic simulations.

In the projects to follow in this chapter and elsewhere in the book, we'll use many of these features and see how they work in more detail.

Project 6: Draw a Spiral Seashell

In this project, we'll dive deeper into the drawing capabilities of the JavaFX canvas and its 2D graphics context. We'll create a complex figure with many circles of increasing sizes, arranged in a spiral around the center of the canvas. By changing the key parameters, we can produce various visual effects. Here we'll use the parameters to make a figure that resembles a seashell with a spiral growth pattern.

The Strategy

Before diving into the code, let's strategize the approach required to generate a spiral pattern. A spiral resembles a circle, with one important difference: its leading edge never returns to the origin point. Instead, it continuously moves farther away from the center while encircling the initial starting point. To achieve this mathematically, we'll employ a method involving a sequence of lines, as illustrated in Figure 3-7.

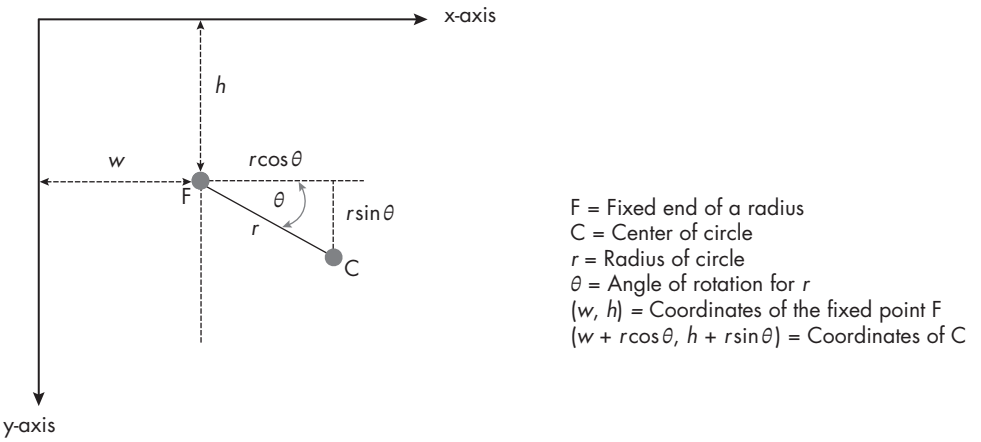


Figure 3-7: The strategy for making a spiral pattern

Each line has one end anchored at a specific point F, which we'll set to the center of the canvas. The position of the other end of the line, C, is determined by two key factors: the length r of the line and an angle of rotation θ relative to the positive x-axis (or another fixed reference). Given the values of r and θ, we can use some basic trigonometry to calculate the coordinates of point C. By gradually increasing both the angle of rotation and the length of the line by preset amounts, the successive values of C will form a spiral as the points simultaneously circle around the fixed point F and grow more distant from it. We'll repeat this until the spiral has achieved the desired number of turns (one turn equals 360 degrees).

To create the spiral pattern, we'll treat each point C as the center of a circle with a radius of length r. These circles will grow larger and move farther away from the starting point as we repeat the process.

The Code

Here's the general structure of the code that will be used to draw the seashell spiral:

```
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.stage.Stage

import kotlin.math.cos
import kotlin.math.sin

class MultiTurnSpiral : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title = "Multi-Turn Spiral"

        // Create a canvas and set its graphics context.
        val canvas = Canvas(600.0, 600.0)
        val gc = canvas.graphicsContext2D

      ❶ primaryStage.scene = Scene(Pane(canvas))
        primaryStage.show()

        // Call helper function to draw the spiral.
        drawMultiTurnSpiral(gc, canvas.width, canvas.height)
    }

    fun drawMultiTurnSpiral(
        gc: GraphicsContext,
        width: Double, height: Double) {
        --snip--
    }

    fun printParams(
        gc: GraphicsContext,
```

```
        radiusStep: Double, numCircles: Int) {
        --snip--
    }

    fun drawCircle(
        gc: GraphicsContext,
        x: Double, y: Double, radius: Double) {
        --snip--
    }
}

fun main() {
    Application.launch(MultiTurnSpiral::class.java)
}
```

We begin by importing the required graphics-related classes, as well as the `cos()` and `sin()` trigonometric functions from Kotlin's `math` library. Then we declare the `MultiTurnSpiral` application class. Much like the rectangle example, we first create and set the graphics elements, then call a custom method, in this case `drawMultiTurnSpiral()`, to coordinate the actual drawing. This approach keeps the overridden `start()` method concise. Notice that we've condensed the creation of the necessary JavaFX nodes by setting the `scene` property of the `primaryStage` object to `Scene(Pane(canvas))` ❶. Nested assignments like this can help eliminate a few lines of code when the intermediate objects aren't needed for other purposes.

Beyond the `start()` method, the application class contains three methods that collectively define and render the spiral pattern. Among them, `drawMultiTurnSpiral()` is the primary method, while the other two, `printParams()` and `drawCircle()`, are helpers. We'll look at each of these methods in detail, starting with `drawMultiTurnSpiral()`.

```
fun drawMultiTurnSpiral(
          gc: GraphicsContext,
          width: Double, height: Double) {

  ❶ // Set key parameters for the spiral.
    val numCircles = 70  // number of circles
    val turns = 2.0      // 360 degrees per turn
    val maxAngle = 360.0 * turns
    // rotation in degrees per step
    val rotationStep = (maxAngle / numCircles)

    // Ensure the circles stay inside the canvas boundaries.
    val maxRadius = minOf(width, height) / 10.0
    // Set the amplification factor.
    val spacingFactor = 2.0
    val radiusStep = (maxRadius / numCircles) * spacingFactor

  ❷ printParams(gc, radiusStep, numCircles)

  ❸ for (i in 0..< numCircles) {
        val angle = i * rotationStep
        val radius = i * radiusStep
```

```
        val x = (width / 2.0) + radius * cos(Math.toRadians(angle))
        val y = (height / 2.0) + radius * sin(Math.toRadians(angle))

        // Draw circles with increasing radii.
      ❹ drawCircle(gc, x, y, radius)
    }
}
```

The `drawMultiTurnSpiral()` method takes three parameters: `gc`, `width`, and `height`, which are the graphics context and the width and height of the canvas, respectively. We first set some key parameters for the spiral ❶. The `numCircles` variable sets the number of circles we'll draw and `turns` dictates the number of rotations we'll make around the center of the spiral. Multiplying `turns` by `360` gives us the maximum angle of rotation we'll achieve, and dividing that value by `numCircles` gives us `rotationStep`, the amount we'll rotate between drawing each circle. Similarly, we set `maxRadius`, the radius of the largest circle, to one-tenth the width or height of the canvas (whichever is lower), then divide this by `numCircles` and multiply the result by `spacingFactor` to get `radiusStep`, the amount by which the radius will grow from one circle to the next. Dividing the minimum of the width and height by `10` gives us some space to play with, and varying the `spacingFactor` allows us to better utilize that space by adjusting the distance between the circles. I've used a `spacingFactor` of `2` to create a visually pleasing pattern, but feel free to experiment with the values to understand how they affect the final figure.

With these parameters set, we call the `printParams()` method ❷, which prints some key parameter values to the canvas. (We'll look at this method shortly.) Then we use a `for` loop ❸ to iterate over the desired number of circles and draw them on the canvas. For each circle, we multiply looping variable `i` by `rotationStep` to calculate the current angle of rotation relative to the positive x-direction, and we multiply `i` by `radiusStep` to get the circle's radius. We then calculate the coordinates of the circle's center (`x`, `y`) by using trigonometric functions, taking the center of the canvas as the fixed point at the center of the spiral. (See Figure 3-7 for how these calculations are derived.) Note that the `cos()` and `sin()` functions expect angles measured in radians rather than degrees, so we call `toRadians()` on the angle. Finally, we pass along the circle's parameters to the custom `drawCircle()` method to actually draw the circle, including its center ❹.

Now let's look at the definitions of the two helper methods, `printParams()` and `drawCircle()`:

```
fun printParams(gc: GraphicsContext, radiusStep: Double, numCircles: Int) {

    val msg1 = "Base radius: " + "%.4f".format(radiusStep) + " pixels"
    val msg2 = "Number of shapes (circles): $numCircles"
    gc.fillText(msg1, 25.0, 555.0)
    gc.fillText(msg2, 25.0, 575.0)
}

fun drawCircle(
        gc: GraphicsContext,
        x: Double, y: Double, radius: Double) {
```

```
    ❶ // Set draw parameters.
      val topLeftX = x - radius
      val topLeftY = y - radius
      val pointSize = 8.0

      with (gc) {
          lineWidth = 2.0
          stroke = Color.LIGHTBLUE
          fill = Color.RED
        ❷ fillOval(x – pointSize / 2, y - pointSize / 2,
                   pointSize, pointSize)
        ❸ strokeOval(topLeftX, topLeftY, radius * 2, radius * 2)
      }
}
```

The `printParams()` method takes in the graphics context and two key parameters of the spiral: the radius step and the number of circles. We create two string templates by using these parameters in the `msg1` and `msg2` variables. In `msg1`, we format `radiusStep`, a floating-point number, to four decimal places. In `msg2`, the number of circles, `numCircles`, is an integer, so no formatting is needed. We then pass the messages along to the `gc.fillText()` method, which displays the text at a specified location on the canvas (we're using coordinates near the bottom-left corner). This method offers a valuable alternative to using the `println()` function, which is limited to displaying text in the console. With `fillText()`, we have the capability to print text directly on the canvas, enhancing the visual representation of the program's output.

The `drawCircle()` method draws an individual circle in the spiral in two ways: as a light blue outline of the full circle and as a smaller red dot to mark the center of the circle. The method takes in the graphics context, the x- and y-coordinates of the circle's center, and the circle's radius. There's a catch, however: the canvas's methods for drawing a circle, `fillOval()` and `strokeOval()`, position the circle not from its center but from the top-left corner of a rectangle that surrounds the circle. We therefore subtract the `radius` from `x` and `y` to get the coordinates for the top-left corner ❶. We also set `pointSize`, which defines the diameter of the small, filled circle marking the circle's center.

For the rest of the method, we use the scoping function `with` to access the properties and methods of the graphics context more easily. We set the line width and stroke color for the outlined circle and the fill color for the central circle. Then we draw the small central circle by using `fillOval()` ❷ and the larger outlined circle by using `strokeOval()` ❸. The first two arguments are the coordinates of the bounding rectangle's top-left corner, and the remaining two are the desired width and height (which for a circle are both twice the radius).

The Result

We're now ready to run the code. It should produce the output shown in Figure 3-8.

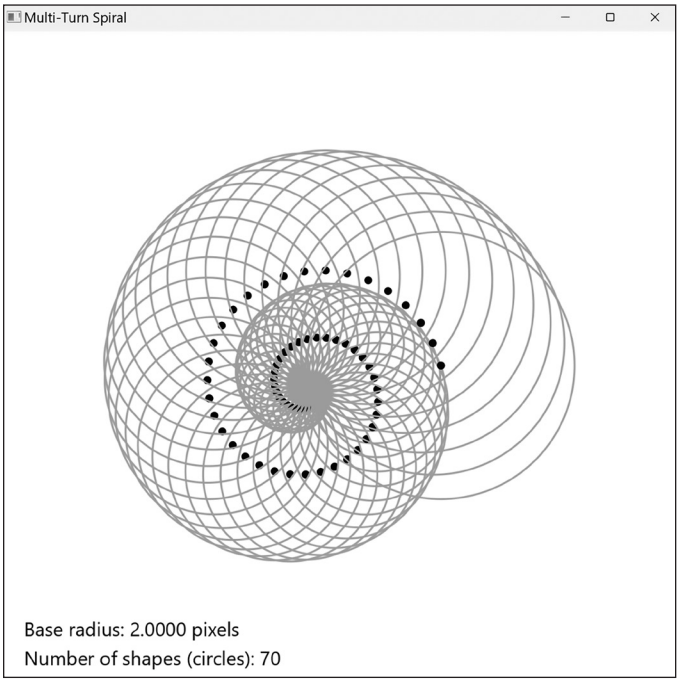


Figure 3-8: A two-turn spiral created with circles of increasing radii

Take a few moments to appreciate the spiral's visual intricacy. By plotting the centers of the circles with a contrasting color, we can clearly see how the successive circles get bigger and farther away from the center of the canvas (recall that the radius was a linear function of `radiusStep`). Ignoring the trajectory of the central dots for a moment, the circles create a visual illusion of a spiral seashell. This is no coincidence: some real-life seashells grow in spirals based on precise mathematical rules.

Animation in JavaFX

Animation is a powerful tool for creating engaging and interactive applications. JavaFX offers various animation options, from simple transitions to complex sequences. You can animate onscreen objects in JavaFX in two main ways: by using the `TranslateTransition` class or by using the `Timeline` and `KeyFrame` classes. In the following projects, we'll explore both of these methods through hands-on examples.

Project 7: Animate a Square

Transition-based animation treats animation as a gradual shift from one state to another. You define the start and end states of a visual object, and JavaFX works out a way to smoothly transition between the two. With transitions, you can make objects move, rotate, scale, fade, and so on, creating simple effects such as sliding, bouncing, flipping, and fading in or out. Transitions are easy to use and require minimal coding, but they're more limited than the timeline and keyframe approach we'll consider in the next project.

To use a transition, you need to create an instance of a transition class, such as `TranslateTransition` to move an object or `RotateTransition` to make it spin. You pass the node you want to animate as an argument to the transition's constructor. Then you set transition properties such as the duration, cycle count, and autoreverse. Finally, you call the transition object's `play()` method to start the animation.

The Code

Here's a simple example that uses transitions to move a square back and forth from one side of the screen to the other:

```
// graphics-related imports
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.scene.shape.Rectangle
import javafx.stage.Stage

// animation-related imports
import javafx.animation.Transition
import javafx.animation.TranslateTransition
import javafx.util.Duration

class TransitionExample : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title = "Transition Example"

        // Create a square.
      ❶ val square = Rectangle(50.0, 50.0, Color.RED)
        square.y = 100.0
        // Create a pane to hold the square.
        val pane = Pane(square)

        // Create a scene and show the stage.
      ❷ val scene = Scene(pane, 300.0, 300.0)
        primaryStage.scene = scene
        primaryStage.show()

        // Create a TranslateTransition class instance
        // and set its properties.
```

```
        ❸ val transition =
              TranslateTransition(Duration.seconds(2.0), square)

          with (transition) {
              fromX = 0.0
              toX = pane.width - square.width
              cycleCount = Transition.INDEFINITE
              isAutoReverse = true
            ❹ play()
          }
      }
}

fun main() {
    Application.launch(TransitionExample::class.java)
}
```

We begin with the import block, which is now organized into imports related to graphics and imports related to animation. The application class, `TransitionExample`, has the same familiar structure we've seen throughout the chapter. I'll highlight only the problem-specific parts.

To animate a square, we first create one by using the `Rectangle` class ❶. We set the side lengths of the square to 50 pixels and its color to red. By default, the square would be positioned at the top-left corner of the window, but we update the square's `y` property to put it 100 pixels lower. We create a `Pane` layout, place the square inside it, and set it as the root node for a 300×300-pixel scene ❷.

To set the transitions, we create an instance of the `TranslateTransition` class named `transition` ❸. In the constructor, we set the duration of the animation to two seconds and assign the `square` object as the node to be animated. We then use `with (transition)` to set the properties controlling the movements of the square. We specify the starting x-position for animation by using the `fromX` property (we'll move the square only from side to side) and the end position by using the `toX` property. For the latter, we use `pane.width - square.width` to ensure the square won't move outside the scene's boundaries. The transition will interpolate the value of the x-position of the square from the `fromX` value to the `toX` value over the duration of the animation. Since we set `cycleCount` to `Transition.INDEFINITE` and `isAutoReverse` to `true`, the transition will reverse direction when it ends and start interpolating back to the `fromX` value—it will continue repeating as long as the window remains open. Finally, we set the transition in motion by using its `play()` method ❹.

The Result

When you run this code, you should see a red square initially positioned next to the left boundary of the scene. It should start moving toward the right boundary, then reverse its course once it hits the right boundary. This back-and-forth movement should continue until you close the window to terminate the program.

Project 8: Animate a Bouncing Ball

The timeline and keyframe approach to animation offers exceptional versatility and is ideal for creating complex and precisely controlled animations. Each `KeyFrame` object defines a specific point in time where you set the values of certain properties, while the `Timeline` class manages the progression between keyframes. In JavaFX, you can find two approaches to this animation method. One is to define the properties of each `KeyFrame` object explicitly. The other is to set the properties of each `KeyFrame` programmatically by using an *action event handler*, a block of code similar to a lambda expression. This block of code is called periodically and contains rules for updating the scene. In this project, we'll look at examples of both methods.

Setting Keyframes Explicitly

In the first example, we'll create an animated circle that grows and shrinks continuously. We'll explicitly define the starting and ending `KeyFrame` objects, representing the circle at its smallest and largest sizes. Then we'll use the `Timeline` class to interpolate back and forth between the keyframes. Here's the code:

```
// graphics-related imports
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.layout.StackPane
import javafx.scene.paint.Color
import javafx.scene.shape.Circle
import javafx.stage.Stage

// animation-related imports
import javafx.animation.KeyFrame
import javafx.animation.KeyValue
import javafx.animation.Timeline
import javafx.util.Duration

class KeyframeAnimationExample : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title =
            "Animation Example: A Growing and Shrinking Circle"
        // Create a circle.
      ❶ val circle = Circle(50.0, Color.BLUE)

      ❷ val root = StackPane(circle) // autocenters child node
        val scene = Scene(root, 600.0, 600.0)
        primaryStage.scene = scene
        primaryStage.show()

        // Create a Timeline for the animation.
        val timeline = Timeline()
        // Define keyframes.
```

```
        val startFrame = KeyFrame(
                         Duration.ZERO,
                         KeyValue(circle.radiusProperty(), 50.0))

        val endFrame = KeyFrame(
                       Duration.seconds(5.0),
                       KeyValue(circle.radiusProperty(), 250.0))

        // Add keyframes to the timeline.
      ❸ timeline.keyFrames.addAll(startFrame, endFrame)

        // Set and play the timeline.
        with (timeline) {
            cycleCount = Timeline.INDEFINITE
            isAutoReverse = true
          ❹ play()
        }
    }
}

fun main() {
    Application.launch(KeyframeAnimationExample::class.java)
}
```

We create a blue `Circle` object with an initial radius of `50.0` pixels ❶ and attach it to a `StackPane` ❷, which automatically centers the node it contains. We manage the animation through a `Timeline` and two `KeyFrame` objects: `startFrame` and `endFrame`. For `startFrame`, we set the initial state of the circle's radius property to `50.0` pixels at the start time of the animation (0 seconds). Notice how we set the radius through an instance of the `KeyValue` class, which we pass to the `KeyFrame` constructor. Any property of an object that should be animated between keyframes must be defined through a separate `KeyValue` object. The second keyframe, `endFrame`, sets the circle's radius property to `250.0` pixels at the end time (five seconds) of the animation. We use the `keyFrames.addAll()` method to attach the two keyframes to the `Timeline` object ❸.

In the final code block, inside the `with` scoping function, we configure the animation to keep repeating by setting the `Timeline` object's `cycleCount` property to `Timeline.INDEFINITE`, and we turn on autoreversing so that the circle will start shrinking once it's grown to its maximum size. Then we initiate the animation by invoking the `play()` method ❹.

If you run this code, an application window should open up where you'll see a blue circle repeatedly grow and shrink, with each cycle lasting five seconds. The animation should continue indefinitely until you close the application window.

FRAME RATE

An animation is nothing more than a series of still images, or *frames*, strung together in rapid succession to create the illusion of motion. The *frame rate* controls how many frames unfold each second. By default, JavaFX uses a frame rate of 60 frames per second, which as of this writing is the maximum possible rate. In the case of the growing and shrinking circle, where one cycle of growing or shrinking lasts five seconds, this means $5 \times 60 = 300$ frames will be created and displayed per cycle. JavaFX automatically generates the intermediate frames between the start and end times, using linear interpolation to calculate the radius of the circle for each frame.

Using an Action Event Listener

We'll now explore a second example of creating animations with a timeline and keyframes. Unlike the previous example, where we specified distinct starting and ending keyframes for each animation cycle, we'll rely on an action event listener block to execute the animation code. Specifically, we'll create a red ball that continually traverses the scene, bouncing off the window boundaries. We'll use the action event listener to set general rules for how the ball should move and when it should bounce. We'll also encapsulate all the animation-related code within a dedicated method instead of putting it all in `start()`. This is a more structured and efficient coding method, which is particularly useful if we want to animate multiple balls by using the same method.

Here's the code for animating a bouncing red ball:

```
// graphics-related imports
import javafx.application.Application
import javafx.scene.layout.Pane
import javafx.scene.Scene
import javafx.scene.paint.Color
import javafx.scene.shape.Circle
import javafx.stage.Stage

// animation-related imports
import javafx.animation.Animation
import javafx.animation.KeyFrame
import javafx.animation.Timeline
import javafx.util.Duration

class BouncingBall : Application() {
    override fun start(primaryStage: Stage) {
        primaryStage.title = "Bouncing Red Ball"
      ❶ val redBall = Circle(250.0, 200.0,
            30.0, Color.RED)
```

```
        val root = Pane(redBall)
        val scene = Scene(root, 500.0, 400.0)
        primaryStage.scene = scene
        primaryStage.show()

        // Call the bouncyBall method.
      ❷ bouncyBall(redBall, scene)
    }

    fun bouncyBall(redBall: Circle, scene: Scene) {
        // displacement parameters
        var dx = 2
        var dy = 2

        // Timeline-KeyFrame with ActionEvent
      ❸ val tl = Timeline()
      ❹ val moveBall = KeyFrame(
            Duration.seconds(0.015),
            {
                // Get min/max boundary coordinates.
                val xMin = redBall.boundsInParent.minX
                val xMax = redBall.boundsInParent.maxX
                val yMin = redBall.boundsInParent.minY
                val yMax = redBall.boundsInParent.maxY

                // Change direction if boundary is hit/crossed.
                if (xMin < 0 || xMax > scene.width) {
                    dx = - dx
                }
                if (yMin < 0 || yMax > scene.height) {
                    dy = - dy
                }
                // Continue to move.
                redBall.translateX += dx
                redBall.translateY += dy
            })

        with (tl) {
          ❺ keyFrames.add(moveBall)
            cycleCount = Animation.INDEFINITE
          ❻ play()
        }
    }
}

fun main() {
    Application.launch(BouncingBall::class.java)
}
```

Inside the `start()` method of the `BouncingBall` application class, we first instantiate a red ball (`redBall`) with a radius of 30 pixels, positioning it at coordinates (250, 200) relative to the top-left corner (0, 0) of the scene ❶. To prevent the ball from constantly occupying the center of the scene, we assign it to a `Pane` instead of a `StackPane`. The rest of the code up to

calling the `show()` method of `primaryStage` is practically the same as in the previous example.

Once we're done with setting the graphics part of the application, we call the custom `bouncyBall()` method, passing `redBall` and `scene` as its arguments ❷. This method encapsulates the process of setting up the `Timeline` and `KeyFrame` objects and maneuvering the ball inside the scene. Inside the method, we first define two displacement parameters, `dx` and `dy`, which set the horizontal and vertical distance (in pixels) that the red ball should travel in each animation frame. We initialize these values to `2`, but if you modify them, the speed of the ball will change. For example, using `4` would double the ball's speed, but this adjustment might introduce a perceptible jitter in the ball's motion.

Next, we create `tl`, an instance of the `Timeline` class ❸, and define a single `KeyFrame` object named `moveBall` ❹. We set the duration of the `KeyFrame` to `0.015` seconds (you can play with this value to get a sense of what happens when the duration is increased or decreased). The rest of the code block, surrounded by braces, is the action event handler block. This block, which we pass as the second argument to the `KeyFrame` constructor, is executed each time the `KeyFrame` is visited along the timeline.

The first few lines inside the action event handler block get the ball's minimum and maximum coordinates in the x- and y-directions based on its current position. This is done by checking the corresponding coordinates of the parent container (a square, in this case) that holds the child (the circular ball). We then use these values to check if the ball has crossed the boundary of the scene during the last update of its position. For example, if `xMin < 0` is `true`, the leftmost edge of the ball will be outside the left boundary of the scene. Similarly, if `xMax > scene.width` is `true`, the rightmost edge of the ball will be beyond the right boundary of the scene. For any such situations, we reverse the ball's direction of movement along the appropriate axis by negating the corresponding displacement parameter (`dx` or `dy`), which creates a bouncing effect. The last step inside the action event block is to update the ball's position by adding the displacement parameters to the coordinates of the ball, which are accessed by using the `redBall` object's `translateX` and `translateY` properties.

We conclude the `bouncyBall()` method by using the `with` scoping function to assign the `moveBall` keyframe to the timeline ❺, set the timeline to cycle indefinitely, and call the `play()` method ❻ to start moving the ball on its bouncy path. Although we can't show the dynamic motion of the ball on the static page of a book, Figure 3-9 gives you a sense of what to expect when you run this code.

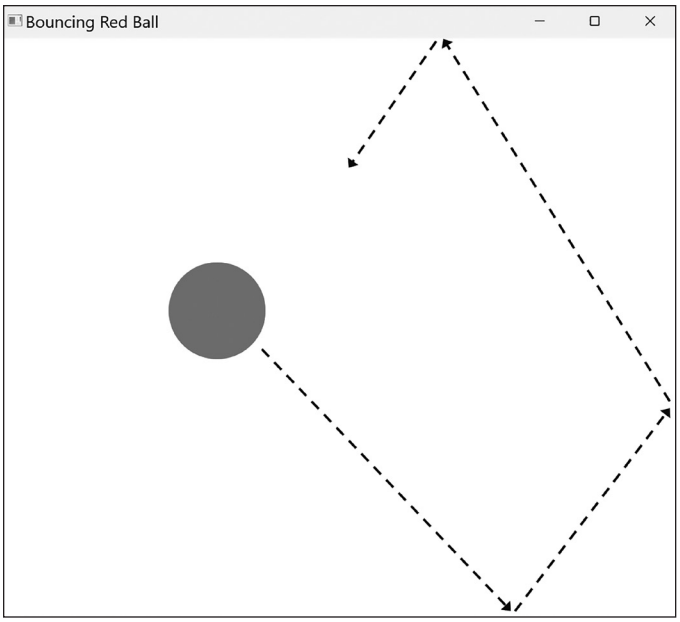


Figure 3-9: Animating a bouncing red ball (shown here in gray)

If you run this code several times and carefully observe the motion of the ball, you'll notice that it actually follows the exact same path every time. In other words, its path is predetermined. Can you explain why that's the case? What could we do to make its path less predictable?

EXERCISE

Enhance the bouncing ball code by adding new features. Here are some ideas to try:

- Add more balls to the screen, each with a different color, speed, and starting location. You can vary these properties randomly by using Kotlin's built-in random number generator function. For example, to generate a random integer between 0 and 10 (inclusive), you could use this code:

```
val randInt = (0..10).random()
```

- Introduce collisions between the balls. That is, instead of the balls simply moving past each other, they should bounce off each other according to the rules of physics. This is a common feature in many games.
- Introduce additional walls inside the canvas, and allow the balls to bounce off these walls as well.

Summary

In this chapter, we explored how to integrate JavaFX with Kotlin to create data visualizations. We covered fundamental JavaFX components such as `Stage`, `Scene`, and various layout containers and nodes. We also discussed using the `Canvas` class and its graphics context to create free-form drawings, and how to implement animations by using transitions or the `Timeline` and `KeyFrame` classes. We practiced these concepts through projects drawing various charts, generating a seashell-like spiral pattern, and even animating a bouncing ball. Throughout the book, we'll continue to draw on the basic JavaFX tools covered here to add visual components to our various projects.

Resources

Balasubramanian, Anirudh. "Crash Course into JavaFX: The Best Way to Make GUI Apps." (Free tutorial.) Accessed June 15, 2024. *https://www.udemy.com*.

Dea, Carl, Gerrit Grunwald, José Pereda, Sean Phillips, and Mark Heckler. *JavaFX 9 by Example*. 3rd ed. New York: Apress, 2017.

JavaFX. The Official Website for the Open JavaFX Project. Accessed June 15, 2024. *https://openjfx.io*.

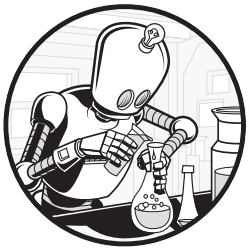
Lowe, Doug. *JavaFX for Dummies*. Hoboken, NJ: John Wiley & Sons, 2015.

PART II

APPLICATIONS IN MATH AND SCIENCE

4

SOLVING MATHEMATICAL PROBLEMS WITH CODE



After our thorough overview of the fundamentals of the Kotlin programming language and the JavaFX graphics tools adapted for use in Kotlin, we're now prepared to tackle a series of math problems in the form of mini projects. The projects will grow in complexity over the chapter, but they require only high school math skills. Our journey will take us from the ancient civilizations of Babylon, Greece, and Egypt to the modern-day world of cryptography.

The main goal of these projects is to enhance your Kotlin programming skills. We'll discuss the context and mathematics behind each problem in detail, but the heart of each project will be developing an appropriate algorithm or problem-solving strategy and then implementing it in well-organized code. In doing so, you'll gain a deeper understanding

of both programming and math, preparing you to solve the more sophisticated problems introduced later in this book.

Project 9: Find the Square Root with the Babylonian Algorithm

We have several methods of finding the square root of a number. In this project, I'll focus on the Babylonian square root algorithm, one of the most widely used methods today.

The Babylonian square root algorithm dates back to around 1800 BCE. It's believed the Babylonians used it for practical purposes, such as land surveying. The algorithm was later refined by the Greeks, who used it to calculate square roots to a high degree of accuracy. The Greek mathematician Heron of Alexandria described the algorithm in his work *Metrica*, written in the first century CE.

Despite its ancient origins, the Babylonian square root algorithm remains a valuable tool for understanding the history of mathematics and the development of numerical methods. The algorithm's enduring usefulness is a testament to the ingenuity of ancient mathematicians and the power of mathematical techniques.

THE ORIGIN OF THE BABYLONIAN ALGORITHM

The Yale Babylonian Collection, part of the Yale University Library in New Haven, Connecticut, houses a Babylonian clay tablet called YBC 7289, which contains the earliest known description of the Babylonian square root algorithm. This tablet has been a significant object of study for scholars of ancient mathematics and the history of science since it was acquired in the early 20th century. The Yale Babylonian Collection also includes a number of other important artifacts, including cuneiform tablets, cylinder seals, and other objects that provide insight into the culture, society, and technology of ancient Mesopotamia.

We can use the Babylonian algorithm to approximate the square root of a positive number in a few simple and iterative steps. The algorithm starts with an initial guess and then refines that guess until it's close enough to the actual square root. Here's how the algorithm works:

1. Start with an initial estimate, `guess`, for the square root of a positive number, `N`. This is customarily set to `N / 2`.
2. Check to see if the absolute value of (`guess * guess - N`) is less than the tolerance value. If yes, then terminate the loop and return the estimated square root.
3. Otherwise, update the guess using the formula `guess = (guess + N / guess) / 2.0`.
4. Repeat steps 2 and 3 until the stopping condition is met.

The Code

The Babylonian algorithm is simple enough that we can write a concise code segment to find a square root. However, it's a good practice to create a separate function for a process like this and then call that function from `main()`. This makes the code more reusable and easier to read.

Here's an example Kotlin function for calculating the square root of a positive number using the Babylonian algorithm:

```
fun babylonianSquareRoot(num: Double): Double {
    val TOL = 0.000001
    var iter = 1
    var guess = num / 2.0

    while(Math.abs(guess * guess - num) > TOL) {
        println("iter: $iter  guess=$guess")
        guess = (guess + num / guess) / 2.0
        iter ++
    }
    return guess
}
```

This `babylonianSquareRoot()` function takes a positive double-precision number `num` as its single argument. It sets the tolerance value `TOL` to `0.000001`, initializes a variable `iter` to `1` to track the number of iterations, and makes a starting `guess` of `num / 2.0`. The function then follows the Babylonian algorithm I described, using a `while` loop to refine the value of `guess` until the result is within the tolerance. To help visualize the convergence process, the intermediate values of `iter` and `guess` are printed at each iteration.

To use this function, call it from the `main()` function and provide the value of the number you want to find the square root of, like so:

```
fun main() {
    println("\n*** Finding Square Root Using Babylonian Algorithm ***\n")
    println("Enter a number (>=1) to find its square root:")
    val num = readln().toDouble()
    println("You have entered: $num\n")
    val squareRoot = babylonianSquareRoot(num)
    println("\nThe estimated square root of $num is: $squareRoot\n")
}
```

In this case, the user is asked to enter the value of a positive number, which is read as a string and converted into a number of type `Double` before its square root is estimated. We're assuming that the user will enter a valid number, which is greater than or equal to 1. If the user enters characters that cannot be converted into a number of type `Double`, the program is terminated with an error message. Also, if the user enters a valid negative number, the algorithm will not converge to a real solution.

In Chapter 1, you learned how to handle such errors or exceptions. Feel free to experiment with this code and to make it error-proof by using a `try...catch` block.

The Result

Without further ado, let's try running the algorithm! If `N` is set to `25`, the code should output the following:

```
*** Finding Square Root Using Babylonian Algorithm ***

Enter a number (>=1) to find its square root:
25
You have entered: 25.0

iter: 1  guess=12.5
iter: 2  guess=7.25
iter: 3  guess=5.349137931034482
iter: 4  guess=5.011394106532552
iter: 5  guess=5.000012953048684

The estimated square root of 25.0 is: 5.000000000016778
```

Of course, the exact value of the square root of 25 is 5. The Babylonian algorithm, like any other numerical algorithm, provides only an approximation. The accuracy of this approximation is determined by the value of tolerance (`TOL`), which can be adjusted to make the approximation more or less precise.

Keep in mind that a more accurate square root approximation will take longer to compute since the algorithm needs to go through more iterations. This sort of trade-off between accuracy and computational time is common.

Project 10: Create Pythagorean Triples with Euclid's Formula

Pythagoras was a Greek philosopher and mathematician who lived in the sixth century BCE. He believed in the idea of a harmonious universe and saw numbers, mathematics, and geometry as key elements in revealing the universe's mysteries. He's best known for the Pythagorean theorem, which states that in a right-angled triangle, the square of the length of the hypotenuse is equal to the sum of the squares of the other two sides (see Figure 4-1). Perhaps you've seen this theorem summarized as $a^2 + b^2 = c^2$.

Pythagorean triples are sets of three positive integers (a, b, c) that satisfy the Pythagorean theorem. A familiar example is $(3, 4, 5)$: $3^2 + 4^2$ equals $9 + 16$, which equals 25, or 5^2. Pythagorean triples are used in many areas of mathematics, science, and engineering, including geometry, number theory, cryptography, physics, and computer graphics. Throughout history, mathematicians have come up with different ways of generating Pythagorean triples. In this project, we'll check out one of the earliest methods, Euclid's formula, and use it to create Pythagorean triples. Here are the steps involved:

1. Choose an arbitrary positive integer k.
2. Choose a pair of positive integers m and n, such that $m > n > 0$.
3. Calculate $a = k(m^2 - n^2)$, $b = 2kmn$, and $c = k(m^2 + n^2)$.
4. The values a, b, and c form a Pythagorean triple (a, b, c).

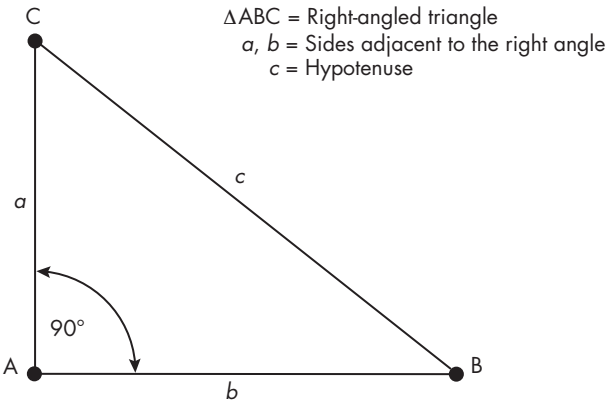


Figure 4-1: The Pythagorean theorem

A Pythagorean triple is considered *primitive* if its members are all coprime, meaning they share no common factors other than 1. For example, (3, 4, 5) and (6, 8, 10) are both Pythagorean triples, but only (3, 4, 5) is primitive, since 6, 8, and 10 have a common factor of 2. Euclid's formula will generate a primitive Pythagorean triple if and only if the two integers m and n are coprime, and one of them is even. If both m and n are odd, then the values of a, b, and c will all be even, and the triple won't be primitive. However, as long as m and n are coprime, dividing the values of a, b, and c by 2 will result in a primitive Pythagorean triple.

The Code

Here's a Kotlin function that generates a Pythagorean triple using Euclid's formula:

```
fun generatePythagoreanTriple(m: Int, n: Int):
    Triple<Int, Int, Int> {
    val a = m * m - n * n
    val b = 2 * m * n
    val c = m * m + n * n
    return Triple(a, b, c)
}
```

The function takes in two integers m and n, then uses them to calculate and return a, b, and c. By not explicitly including a value for k here, we're implicitly assuming $k = 1$.

We can call this function repeatedly from the `main()` function, using a `for` loop to generate Pythagorean triples for an arbitrary number of pairs of successive integers, as shown in the following code:

```
fun main() {
    var m = 2               // value of m
    var n = 1               // value of n
    val numTriples = 10     // number of triples
```

```
    println("\n*** Pythagorean Triples Using Euclid's Formula ***\n")
    println("Number of Pythagorean triples: $numTriples\n")

    // Generate the first "numTriples" triples.
    for (i in 1..numTriples) {
        val pythagoreanTriple =
            generatePythagoreanTriple(m, n)
        print("i=${"%2d".format(i)}    " +
              "m=${"%2d".format(m)}    n=${"%2d".format(n)}  ")
        println("Pythagorean triple: $pythagoreanTriple")
        n++
        m++
    }
}
```

Notice that the first triple is generated using an `m` of `2` and an `n` of `1`. These are the smallest possible values of `m` and `n`. (Recall the stipulation in Euclid's formula that m and n must be positive integers such that $m > n > 0$.) These values are passed on as arguments to the `generatePythagoreanTriple()` function, which returns the elements of the Pythagorean triple as a `Triple` object in Kotlin. Successive inputs are generated by incrementing both `m` and `n` inside the `for` loop that repeats `numTriples` times. Since `m` and `n` are consecutive, one of them will always be even, and they can't possibly share any factors, so every triple will be primitive.

The Result

If you don't change any of the program parameters, the program will produce the following output that shows the first 10 Pythagorean triples:

```
*** Pythagorean Triples Using Euclid's Formula ***

Number of Pythagorean triples: 10

i= 1    m= 2    n= 1  Pythagorean triple: (3, 4, 5)
i= 2    m= 3    n= 2  Pythagorean triple: (5, 12, 13)
i= 3    m= 4    n= 3  Pythagorean triple: (7, 24, 25)
i= 4    m= 5    n= 4  Pythagorean triple: (9, 40, 41)
i= 5    m= 6    n= 5  Pythagorean triple: (11, 60, 61)
i= 6    m= 7    n= 6  Pythagorean triple: (13, 84, 85)
i= 7    m= 8    n= 7  Pythagorean triple: (15, 112, 113)
i= 8    m= 9    n= 8  Pythagorean triple: (17, 144, 145)
i= 9    m=10    n= 9  Pythagorean triple: (19, 180, 181)
i=10    m=11    n=10  Pythagorean triple: (21, 220, 221)
```

In this example, the starting values of `m` and `n` were hardcoded in the `main()` function. You might modify the code to allow the user to input values of `m` and `n` (ensuring $m > n > 0$). That would allow the user to generate a wider range of Pythagorean triples based on their requirements.

Project 11: Identify Prime Numbers with the Sieve of Eratosthenes

Eratosthenes was an ancient Greek scholar who lived in the third century BCE. He was an accomplished mathematician, astronomer, geographer, and poet. In this project, we'll explore one of Eratosthenes's many mathematical discoveries: the *sieve of Eratosthenes*, an intuitive algorithm for identifying all the prime numbers up to a given limit. (We'll explore another of his ingenious discoveries in the next project.) It's remarkable to think that Eratosthenes conceived this strategy more than two millennia ago, during a time when few individuals could read or write, let alone think about algorithms and solve abstract mathematical problems.

> **WHAT IS A PRIME NUMBER?**
>
> A *prime number* is a positive integer greater than 1 that has no positive integer divisors other than 1 and itself. In other words, the number can be divided evenly only by 1 and itself. For example, 5 is a prime number because it can be divided evenly only by 1 and 5. It has no other positive integer divisors. On the other hand, 6 isn't a prime number, because it can be divided evenly by 1, 2, 3, and 6. Instead, we call it a *composite number*. The first several prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, and so on. Prime numbers have many interesting properties and play a central role in number theory and cryptography, among other fields.

The Strategy

To sieve for primes like Eratosthenes, start by creating a list of all integers from 2 up to some limit. Then, starting from 2, iteratively mark off all multiples of each prime number as composite. The unmarked numbers that remain at the end of the process are all prime numbers. Here are the steps to implement this algorithm:

1. Create a list of consecutive integers from 2 through the given limit.
2. Starting with 2 (the first prime number), mark all its multiples as composite.
3. Find the next number in the list that isn't marked as composite. This will be the next prime number.
4. Mark all multiples of the prime number found in step 3 as composite.
5. Repeat steps 3 and 4 until the square of the next prime number exceeds the given limit.
6. The unmarked numbers in the list are all prime numbers.

We can optimize the sieve algorithm by marking multiples of each prime number, starting from its square. For example, when marking multiples of 3, we can start at $3^2 = 9$, since all multiples of 3 less than 9 will already have been marked as composite. In this case, 6 will have been marked while going through the multiples of 2. Similarly, when we get to multiples of 5, we can skip 10, 15, and 20 as being multiples of either 2 or 3 and start marking off composites from 25.

The Code

As before, we'll start by writing a function to implement the algorithm and then use the `main()` function to call this function and list the prime numbers. Here's the Kotlin code for the `sieveOfEratosthenes()` function:

```
fun sieveOfEratosthenes(n: Int): List<Int> {
    // Create a Boolean array with all values set to true.
    val primes = BooleanArray(n + 1) { true }
    // Create a mutable list of integers to save prime numbers.
    val primeNumbers = mutableListOf<Int>()

    // Set 0 and 1 to not be prime.
    primes[0] = false
    primes[1] = false

    // Iterate over all numbers until i^2 > N.
    var i = 2
  ❶ while (i*i <= n) {
        // If i is prime, mark all multiples of i as not prime.
      ❷ if (primes[i]) {
          ❸ for (j in i * i..n step i) {
                primes[j] = false
            }
        }
        i++
    }

    // Collect all prime numbers into a list and return it.
  ❹ for ((index, value) in primes.withIndex())
        if (value) primeNumbers.add(index)

  ❺ return primeNumbers
}
```

The `sieveOfEratosthenes()` function takes an integer `n` as input and returns a list of prime numbers up to `n`. For that, the function creates a Boolean array `primes` with a length of `n + 1` and initializes each element's value to `true`. Over the course of the function, we'll change elements to `false` if their indices aren't prime. The function also creates a mutable list `primeNumbers` of type `Int` to save the prime numbers.

To begin, we set the first two values of the `primes` array to `false` because 0 and 1 aren't prime. We then iterate over the numbers from `2` to the square root of `n` (we do this by ensuring `i*i <= n`) ❶. For each number `i` in this range, if `i` is marked as prime (that is, `true`) in the `primes` array ❷, the function marks all multiples of `i` in the `primes` array as composite (`false`). To reach all the multiples of `i`, we use a `for` loop with a step size of `i` ❸.

To collect the prime numbers, we use a `for` loop ❹ to go over all `primes` elements and add the corresponding index to `primeNumbers` when the value of the element is `true`. Finally, we return the prime numbers to `main()` as a list of integers for postprocessing ❺.

Now that our sieving function is good to go, we can use the `main()` function to retrieve a list of prime numbers up to `n` and print it out. We'll also create a `printPrimes()` helper function to manage the printing.

```
fun main() {
    println("\n*** Find All Prime Numbers Up to 'n' ***\n")
    println("Enter a number > 2 to generate the list of primes:")
    val num = readln().toInt()
    println("You have entered: $num")

    val primeNumbers = sieveOfEratosthenes(num)
    println("\nThe prime numbers <= $num are:")
    printPrimes(primeNumbers)
}

fun printPrimes(primeNumbers: List<Int>) {
    for (i in primeNumbers.indices) {
        if (i != 0 && i % 6 == 0) println()
        print("${"%8d".format(primeNumbers[i])} ")
    }
}
```

The `main()` function is similar to the one we used for the Babylonian square root algorithm in Project 9. It takes a user input for the limit `num`, uses it to create a list of prime numbers with the `sieveOfEratosthenes()` function, and then calls `printPrimes()` to print the list. To make the output look nice, `printPrimes()` organizes the numbers into rows of six and uses string formatting to create neatly aligned columns.

The Result

Here's a look at the program output up to an arbitrary limit `num` of `251`:

```
*** Find All Prime Numbers Up to 'n' ***

Enter a number > 2 to generate the list of primes:
251
You have entered: 251
```

```
The prime numbers <= 251 are:
      2       3       5       7      11      13
     17      19      23      29      31      37
     41      43      47      53      59      61
     67      71      73      79      83      89
     97     101     103     107     109     113
    127     131     137     139     149     151
    157     163     167     173     179     181
    191     193     197     199     211     223
    227     229     233     239     241     251
```

Other methods for generating prime numbers include the sieve of Sundaram, the sieve of Atkin, and trial division. I encourage you to do some online research and experiment with these methods to enhance your Kotlin coding skills and gain additional insight into prime number generation.

Project 12: Calculate Earth's Circumference the Ancient Way

One of Eratosthenes's most famous achievements was calculating Earth's circumference. He accomplished this by measuring the angle of the sun's rays at noon on the summer solstice at two locations, Alexandria and Syene (modern-day Aswan), which were known to be on the same meridian, or longitude. Figure 4-2 shows an abstraction of some of the geometry involved in this brilliant experiment.

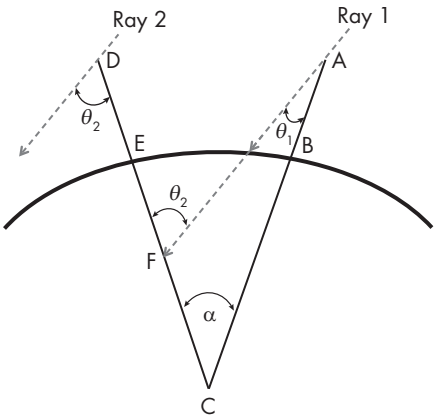


Figure 4-2: Measuring Earth's circumference

In this diagram, we can think of points E and B as two locations on Earth's surface (for example, Alexandria and Syene). We'll assume for the sake of simplicity that Earth is a perfect sphere and that both of these locations are on the same meridian. Let's also assume that *AB* and *DE* are two tall poles or towers that are sufficiently far apart that when extended to Earth's center (C), they create a small but measurable angle α. Two parallel rays coming from the sun just miss the tops of the poles and will hit the ground at slightly different angles. Because Earth's surface is curved,

the angle θ_2 between ray 2 and *DE* will be slightly greater than the angle θ_1 between ray 1 and *AB*. As a result, the shadow of *DE* on the ground will be longer than that of *AB*, even if the poles themselves have the same height.

We also assume that at both locations the poles are positioned vertically relative to the ground surface, which can be thought of as flat in the vicinity of the poles. This last assumption allows us to measure the angle of a ray of light relative to a pole. Assuming that the length of the shadow is *s* and the height of the pole is *h*, the angle θ between the ray and the pole can be expressed as follows:

$$\theta = \tan^{-1}\left(\frac{s}{h}\right) \tag{4.1}$$

Finally, consider triangle *ACF* in Figure 4-2. According to the *exterior angle theorem*, the triangle's exterior angle *AFE* must be equal to the sum of the two interior opposite angles, *ACF* and *CAF*. Meanwhile, *AFE* and θ_2 are equal because these are the *alternate interior angles* of the line *DF* that intersects the two parallel solar rays. Therefore, the following must be true:

$$\angle AFE = \angle ACF + \angle CAF$$

$$\theta_2 = \theta_1 + \alpha$$

Rearranging the latter, we get:

$$\alpha = \theta_2 - \theta_1 \tag{4.2}$$

This final equation is what we need to estimate Earth's circumference. We'll achieve that by using another geometric relationship that connects the length *d* of an arc of a circle to the angle α (in radians) that the arc creates at the center of the circle:

$$\frac{\alpha}{d} = \frac{2\pi}{\text{Circumference}}$$

Solving for the circumference gives us this equation:

$$\text{Circumference} = \frac{2\pi d}{\alpha} \tag{4.3}$$

What Eratosthenes did was quite ingenious. He knew that at noon on the summer solstice, the sun would be directly overhead at Syene (point B in Figure 4-2), so a vertical pole there would cast no shadow, meaning $\theta_1 = 0$, and therefore $\alpha = \theta_2$ per Equation 4.2. In Alexandria (point E), however, the sun would be at an angle, so a pole would cast a shadow on the ground. By measuring the length of this shadow, Eratosthenes was able to calculate the angle between the sun's ray (ray 2) and the pole (*DE*) using Equation 4.1. He found this angle to be about 7.2 degrees, or 0.12566370614 radians.

Eratosthenes was aware of the distance between Alexandria and Syene—he estimated it to be 5,000 stadia (about 800 kilometers). With this information and the angle of the shadow, he calculated Earth's circumference (using Equation 4.3) and arrived at a value of approximately 40,000 kilometers. Once

he determined the circumference, he could also calculate Earth's radius. For any circle, the radius r can be calculated from the circumference as:

$$r = \frac{\text{Circumference}}{2\pi}$$

Filling in the circumference formula from Equation 4.3, Earth's radius R is:

$$R = \frac{d}{\alpha} \tag{4.4}$$

This calculation gave Eratosthenes a value of 6,370 kilometers, which is remarkably close to the actual value of approximately 6,371 kilometers.

The Code

Let's write some code to imitate the method Eratosthenes used to calculate Earth's circumference and radius. We'll make our program more flexible by allowing the sun to not be directly overhead at the first location. To do this, we'll use Equation 4.1 to figure out the shadow angles, Equation 4.2 to get the arc angle, Equation 4.3 to calculate the circumference, and finally, Equation 4.4 to calculate the radius.

```
import kotlin.math.atan

data class Earth(
    val alpha: Double,
    val circumference: Int,
    val radius: Int
)

fun calculateEarthMetrics(s1: Double, h1: Double,
                          s2: Double, h2: Double, d: Double): Earth {
    // Calculate the angles of the shadows.
    val theta1 = atan(s1 / h1)
    val theta2 = atan(s2 / h2)

    // Calculate the angle at the center of Earth.
    val alpha = theta2 - theta1

    // Calculate the circumference and radius.
    val circumference = (2 * Math.PI * d / alpha).toInt()
    val radius = (d / alpha).toInt()

    return Earth(alpha, circumference, radius)
}

fun main() {
    // known values
    val shadow1 = 0.0    // m
    val height1 = 7.0    // m
    val shadow2 = 0.884  // m
    val height2 = 7.0    // m
    val distanceBetweenCities = 800.0 // in km
    val (alpha, circumference, radius) =
```

```
        calculateEarthMetrics(s1=shadow1, h1=height1,
            s2=shadow2, h2=height2,
            d=distanceBetweenCities)

    // Output the estimated circumference and radius.
    println("\n*** Measuring Earth's Circumference and Radius ***\n")
    println("Angle (alpha): ${"%7.5f".format(alpha)} radian")
    println("Circumference: $circumference kilometers")
    println("Radius: $radius kilometers")
}
```

The code segment starts by importing the required math function and defining a data class `Earth` with three properties: `alpha`, `circumference`, and `radius`. This data class allows us to conveniently package up the values estimated inside the `calculateEarthMetrics()` function and return them via a single `Earth` object.

The `calculateEarthMetrics()` function has five named parameters that represent the shadow lengths (`s1` and `s2`) and heights (`h1` and `h2`) for the two locations, and the distance (`d`) between these two locations. Then the function follows the steps described on page 137: calculating `theta1` and `theta2`, using them to calculate `alpha`, and using `alpha` to estimate `circumference` and `radius`. Since these are large numbers (when expressed in kilometers), we convert both `circumference` and `radius` into integers (which is how they were defined in the `Earth` class).

The `main()` function's job is quite simple: call the `calculateEarthMetrics()` function; receive the values of `alpha`, `circumference`, and `radius` by deconstructing the returned object; and print them with appropriate annotations and format.

The Result

For the given parameter values in this example—the same ones Eratosthenes used—the output of our program looks like this:

```
*** Measuring Earth's Circumference and Radius ***

Angle (alpha): 0.12562 radian
Circumference: 40013 kilometers
Radius: 6368 kilometers
```

Feel free to use this tool to experiment with different parameter values. For example, you could try using shadow lengths and angles measured on an exoplanet and find out how large or small the planet is!

Project 13: Code the Fibonacci Sequence

Leonardo of Pisa, commonly known as Fibonacci, was an Italian mathematician born c. 1170. From an early age, he showed a keen interest in mathematics, and his travels to North Africa and the Middle East exposed

him to advanced mathematical concepts that weren't yet known in Europe. Fibonacci's most significant contribution to mathematics was the introduction of the Indo-Arabic numeral system to the Western world, which included the use of zero. This system replaced the previously used Roman numerals and revolutionized arithmetic calculations, making them significantly more efficient.

Fibonacci is widely recognized for introducing the Fibonacci sequence, a series of numbers where each number is the sum of the two preceding numbers. To explain this concept in his book *Liber Abaci*, Fibonacci used a colorful analogy involving a pair of rabbits. Imagine placing a pair of rabbits in an enclosed area. The rabbits can mate when they're one month old and can produce a new pair of rabbits when they're two months old. Therefore, it takes one month for each new pair to mature and an additional month to give birth to a new pair. If the rabbits never die and the mating continues, how many pairs of rabbits will there be after each month?

The solution to this problem forms the Fibonacci sequence. If we start with (1, 1) representing the starting pair over the first two months, the sequence will look like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on. These numbers can be linked to Fibonacci's rabbit example as explained in Table 4-1.

Table 4-1: Fibonacci's Rabbits

Month	Young pairs	Mature pairs	Total pairs	Explanation
0	1	0	1	Start with a newborn pair, no mature pair, and no offspring.
1	0	1	1	The first pair becomes mature and will reproduce at the end of this period.
2	1	1	2	The first pair of offspring is born. One mature pair will reproduce again at the end of this period.
3	1	2	3	The second pair of offspring is born. Two mature pairs will reproduce at the end of this period.
4	2	3	5	Two new pairs are born. Three mature pairs will reproduce at the end of this period.
5	3	5	8	Three new pairs are born. Five mature pairs will reproduce at the end of this period.

Say we want to determine how many rabbit pairs will exist after a certain number of generations. Here are the steps we can take, using the Fibonacci sequence:

1. Set the first two numbers in the sequence. By convention, these are usually 0 and 1 rather than 1 and 1.
2. Add the first two numbers to get the third number in the sequence.
3. Generate the next number by adding the two preceding numbers. This step can be mathematically expressed as $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$.
4. Repeat step 3 until the stopping condition is met.

The Fibonacci sequence has become a classic example of recursive sequences and is used to illustrate many mathematical concepts in various fields. Before getting into how to code this sequence, I'll introduce you to two other related concepts: the golden ratio and the Fibonacci spiral. These concepts will be illustrated in our Fibonacci code.

The Golden Ratio

The *golden ratio*, also known as the golden mean, is a mathematical ratio commonly found in nature, art, and architecture. The ratio is approximately 1.61803398875 and is denoted by the Greek letter φ.

The golden ratio is linked to the Fibonacci sequence: as the sequence continues, the ratio between each successive pair of numbers approaches the golden ratio. Starting with 1 (we can't start with 0, as the ratio of 1 over 0 is infinity), if we calculate and plot this ratio for each successive pair, it will rapidly converge on φ, as shown in Figure 4-3.

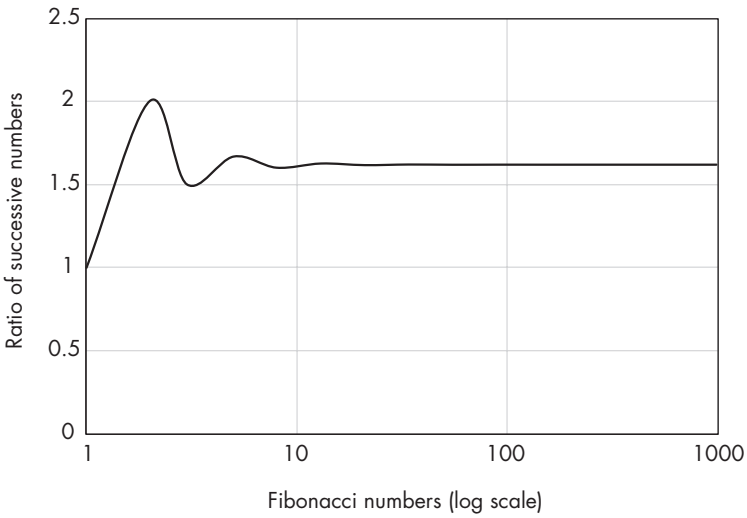


Figure 4-3: Convergence to the golden ratio

Well-known manifestations of the golden ratio in nature include nautilus shells, the arrangements of seeds in a sunflower and scales on a pine cone, and the proportions of the human body (for example, the ratio of the length of the forearm to the hand, and the ratio of the overall height to the height of the navel). The ratio has also been extensively exploited by artists, musicians, photographers, product designers, and architects in their work. In architecture, for example, it might determine the ratio of width to height for a building's facade.

The Fibonacci Spiral

The Fibonacci spiral is a geometric pattern derived from the Fibonacci sequence. It's created by drawing a series of quarter circles inside squares

that are based on the numbers in the Fibonacci sequence. To draw the Fibonacci spiral, follow these steps:

1. Draw a small square with a side length of 1.
2. Draw another square of side length 1 adjacent to the first square, sharing a side.
3. Draw a third square of side length 2 adjacent to the second square, sharing a side.
4. Draw a fourth square of side length 3 adjacent to the third square, sharing a side.
5. Continue this process, drawing squares with side lengths equal to the sum of the two preceding squares, adjacent to the last drawn square, sharing a side.
6. Draw a quarter circle inside each of the squares, connecting the opposite corners of each square. The quarter circles will form a smooth curve: the Fibonacci spiral.

If you follow these steps and draw the spiral for the first eight numbers (starting from 1), the result will look like Figure 4-4.

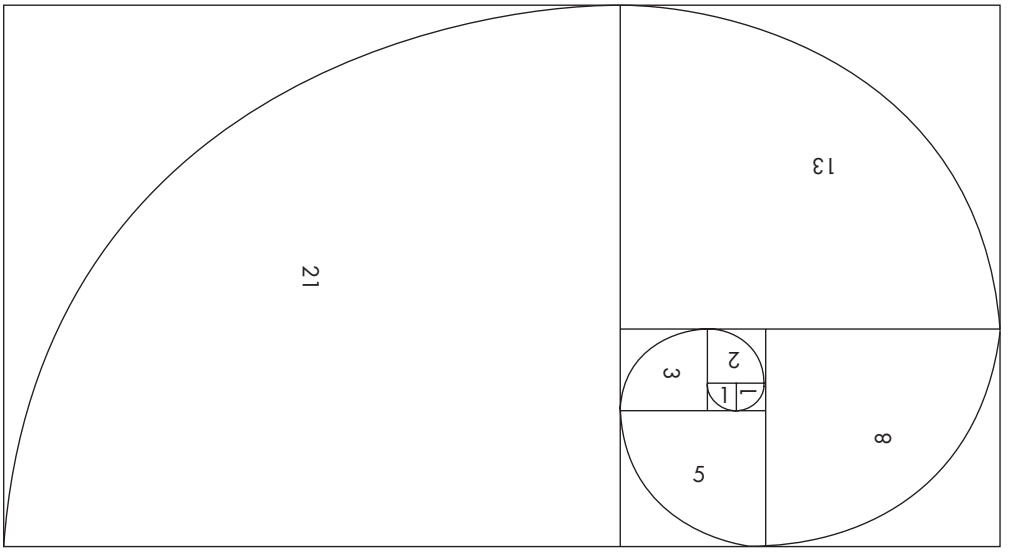


Figure 4-4: A Fibonacci spiral for the first eight numbers

The Fibonacci spiral, often associated with the golden ratio, is a recurring pattern in nature, appearing in various forms such as seashells, leaf arrangements, and even the spirals of distant galaxies! While the golden ratio is not an absolute prerequisite for beauty or efficiency in design, it undeniably holds an enduring charm as a mathematical concept that continues to capture our imagination.

The Code

Generating and printing the Fibonacci sequence up to a certain limit can be accomplished with just a few lines of code. Let's take this project a step further: along with generating the sequence itself, we'll also draw the Fibonacci spiral. This way, we'll be able to practice some of the data visualization techniques covered in the previous chapter and anticipate future projects where we'll gain deeper insight into a problem by visualizing the program output.

Setting Up

To begin, we'll write the global components of the code, including the import block needed for the visualization, the `FibonacciSpiral` application class, and the `main()` function.

```
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.scene.shape.ArcType
import javafx.scene.text.Font
import javafx.stage.Stage

// number of Fibonacci numbers in the list
val N = 9
val fibs = mutableListOf<Int>()

// canvas-related parameters
val canvasW = 1000.0
val canvasH = 750.0

// Scaling parameters: adjust as needed.
val xOffset = 150
val yOffset = 50
val amplify = 25.0

class FibonacciSpiral : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val canvas = Canvas(canvasW, canvasH)
        val gc = canvas.graphicsContext2D
        gc.translate(canvas.width / 2 + xOffset,
            canvas.height / 2 + yOffset)
        root.children.add(canvas)

        val scene1 = Scene(root, canvasW, canvasH)
        scene1.fill = Color.WHITE
        with(stage) {
            title = "Fibonacci Spiral"
```

```
            scene = scene1
            show()
        }

        // code for Fibonacci sequence and spiral
        generateFibonacciNumbers()
        drawFibonacciSpiral(gc)
        printFibonacciSequenceAndRatios()
    }
}

fun main() {
    Application.launch(FibonacciSpiral::class.java)
}
```

The code segment starts with an import block that provides access to a number of JavaFX graphics features that we'll use to draw the Fibonacci spiral on a `canvas` object. See Chapter 3 for a review of these features. Coding in IntelliJ IDEA means you don't need to memorize which library features you need to import; as you use the default template and add code that may require additional graphics elements, the IDE will import those features automatically.

Following the import block, we set up some global parameters. First, we create a variable called `N` to set how far into the Fibonacci sequence we'll go (starting from 0). Then, we create a mutable list of type `Int` named `fibs`, which will store the Fibonacci sequence as we calculate it. We also set several parameters to create a canvas where we'll draw the Fibonacci spiral. To define the size of the canvas, we use the values `canvasW` and `canvasH`, and to set the starting location of the origin of the coordinate system, we use `xOffset` and `yOffset`. For this particular project, I've set the canvas size to 1,000 pixels wide and 750 pixels high, which should be suitable for most screen sizes and resolutions.

It's important to note that the length of a line or the side of a rectangle on the canvas is measured in pixels. To plot the Fibonacci spiral, we'll start with a square of size 1. However, drawing a square of 1 pixel would result in a tiny dot on the screen, which we don't want. To avoid this, we'll use an amplification factor called `amplify` and set it to `25`. Therefore, the first square will be 25 pixels in size, and all subsequent squares will be amplified by the same factor. This ensures that the end result is a Fibonacci spiral that fills the canvas nicely.

In the `FibonacciSpiral` application class, we first create a layout container called `root` of type `Pane()`, which is the most basic layout container in JavaFX to hold and position nodes (user interface components) in a scene. We use `root` to hold the `canvas` on which the spiral will be drawn. Notice how we use the `translate` property of the graphics context `gc` to shift the initial position of the origin from the top-left corner (the default) to a position offset a bit from the middle of the canvas, where we'll draw the first Fibonacci square (see Figure 4-4 to get a sense of where that is). The rest of the class is routine JavaFX: we assign `canvas` to `root`, which is assigned to `scene1`, which connects to `stage`, the primary display window for this application.

Next, we move to the problem-specific segment of the application class, which consists of calls to three separate functions: `generateFibonacciNumbers()`, `drawFibonacciSpiral()`, and `printFibonacciSequenceAndRatios()`. These functions do exactly what their names suggest, and we'll discuss them in detail shortly.

Finally, the `main()` function contains a single line of code that launches a JavaFX application by calling the `launch()` method of the `Application` class, passing it the `FibonacciSpiral` class as an argument.

Generating the Fibonacci Sequence

The `generateFibonacciNumbers()` function generates the Fibonacci sequence as discussed earlier.

```
fun generateFibonacciNumbers() {
    // Add the starting pair.
    fibs.add(0)
    fibs.add(1)

    // Generate the sequence.
    for (i in 2 until N) {
        fibs.add(fibs[i-1] + fibs[i-2])
    }
}
```

First, $F(0)$ and $F(1)$ are set to 0 and 1, respectively, and then the rest of the sequence is generated using $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$. We add all generated numbers to the mutable list `fibs` using its `fibs.add()` method.

Drawing the Fibonacci Spiral

The `drawFibonacciSpiral()` function drives the process of drawing the Fibonacci spiral using the generated sequence of Fibonacci numbers, with support from two other helper functions that label each square with its corresponding Fibonacci number and draw the quarter circles.

```
fun drawFibonacciSpiral(gc: GraphicsContext) {
    for (i in 1 until N) {
      ❶ val side = fibs[i] * amplify
      ❷ with(gc) {
            strokeRect(0.0, 0.0, side, side)
            drawText(i, gc, side)
            drawArc(gc, side)
            // Move to the opposite corner by adding
            // side to both x- and y-coordinates.
            translate(side, side)
            // Rotate the axes counterclockwise.
            rotate(-90.0)
        }
    }
}
```

```
fun drawText(i: Int, gc: GraphicsContext, side: Double) {
    gc.fill = Color.BLACK
    with(gc) {
      ❸ font = when {
            i <= 2 -> Font.font(12.0)
            else -> Font.font(24.0)
        }
        fillText(fibs[i].toString(), side/2, side/2)
    }
}

fun drawArc(gc: GraphicsContext, side: Double) {
    val x = 0.0
    val y = -side
    with(gc) {
        lineWidth = 3.0
        strokeArc(x, y, 2*side, 2*side,
            -90.0, -90.0, ArcType.OPEN)
    }
}
```

The `drawFibonacciSpiral()` function uses a `for` loop that starts with `1` (we cannot draw a square of size `0`) to iterate over the sequence of numbers. In the loop, we retrieve the current number and multiply it by `amplify` to properly scale the squares on the screen ❶. Then we implement the rest of the process inside a `with(gc)` block ❷ where we draw a square, annotate it, and draw an arc. At the end of each cycle, we move the canvas's origin to the next location and rotate the coordinate system counterclockwise by 90 degrees. This way, the squares will spiral outward, as shown in Figure 4-4, but we'll still be able to draw each one with the same `strokeRect(0.0, 0.0, side, side)` call.

In the `drawText()` function, we use the argument `i`, which represents the index of the current Fibonacci number, to set the font size of the text used for annotation ❸. This ensures that the first two numbers fit inside a square of size 25 pixels. We then use the `fillText()` method of the graphics context `gc` to draw the number in the middle of its corresponding square.

The `drawArc()` function sets up the parameter values needed by the `strokeArc()` method of the graphics context. These parameters include the top-left corner of the rectangle, its width and height, the starting angle with respect to the x-axis in degrees, and the length of the arc in degrees. We also specify the arc type as `OPEN`, which means the two endpoints won't be connected with a line.

For drawing the arc, think of the imaginary box inside which the arc will be drawn as a stand-alone object that has its own coordinate system whose origin is at the center of the box. Inside this box, the positive x-axis points east and the positive y-axis points north. (Note that this isn't the same as the default convention used by the JavaFX canvas.) Taking this into account, drawing an arc counterclockwise is considered the positive direction, and this is how the starting angle and arc length are specified. For example, we've specified the start angle as –90 degress and the arc length as –90 degrees (both in the clockwise direction relative to the positive x-axis). Alternatively,

we could have specified the start angle as +180 degress and the arc angle as +90 degrees (both counterclockwise) to produce the same result.

Printing the Sequence

We have one more function that prints the Fibonacci sequence, as well as the ratios between successive terms in the sequence, to illustrate how these values converge on the golden ratio.

```
private fun printFibonacciSequenceAndRatios() {
    println("\n*** Fibonacci sequence and ratios ***\n")
    println("Length of Fibonacci sequence=${fibs.size}")
    println("Generated sequence:")
  ❶ println(fibs)
    println("\nRatio F(n+1)/F(n) [starting from (1,1) pair]:")
    for (i in 2 until fibs.size) {
        println("%5d".format(fibs[i-1]) +
                "%5d".format(fibs[i]) +
                "%12.6f".format(fibs[i].toDouble()/fibs[i-1])
        )
    }
}
```

The function first prints a header message and the length of the generated Fibonacci sequence. Next, it prints the generated sequence itself using `println()` ❶. Finally, a `for` loop calculates and prints the ratios of adjacent numbers in the sequence, using `format()` to show the values with appropriate spacing and precision.

The Result

When you run the code, the text portion of the output should appear as follows:

```
*** Fibonacci sequence and ratios ***

Length of Fibonacci sequence=9
Generated sequence:
[0, 1, 1, 2, 3, 5, 8, 13, 21]

Ratio F(n+1)/F(n) [starting from (1,1) pair]:
    1    1    1.000000
    1    2    2.000000
    2    3    1.500000
    3    5    1.666667
    5    8    1.600000
    8   13    1.625000
   13   21    1.615385
```

Notice how the ratios initially zigzag around the value of 1.61803398875 but quickly approach the golden ratio once we reach the 10th pair in the sequence.

Of course, the app also displays a beautiful Fibonacci spiral drawn on the canvas using JavaFX. It should look exactly like Figure 4-4—that figure was generated with this very code!

EXERCISE

A concept related to the Fibonacci sequence and the golden ratio is Pascal's triangle, named after the French mathematician Blaise Pascal (although it was known to Chinese mathematicians over 500 years earlier). It has many interesting properties and applications in mathematics, including its use in calculating binomial coefficients, which arise in probability theory and other fields.

Pascal's triangle starts with a row containing a single number 1. Each subsequent row has one more number than the row above it. Each number is determined by adding the adjacent pair of numbers directly above it (except for a 1 on either end of each row). The first seven rows of Pascal's triangle are shown here:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

Try writing the Kotlin code to generate Pascal's triangle up to some arbitrary length (say, 10 rows). Display the result as text output formatted to look like a triangle.

Project 14: Find the Shortest Distance Between Two Locations on Earth

We use the Pythagorean theorem to calculate distances between points on the same plane. However, for points on Earth's surface, this method isn't accurate over long distances, because it doesn't consider Earth's curved shape. That's where the *haversine formula* comes in. It calculates the shortest distance between two points on the surface of a sphere using the latitude and longitude coordinates of the points. In the case of Earth, the formula isn't totally accurate, since Earth isn't perfectly spherical, but it still offers a reasonable distance approximation for many practical applications, including in navigation, astronomy, and geography.

The haversine formula revolves around the concept of a *great circle*, the largest circle that can be drawn on a sphere. It's formed by the intersection of the sphere's surface with a plane that passes through the sphere's center.

The great circle divides the sphere into two equal halves, and its circumference matches the circumference of the sphere itself.

Figure 4-5 showcases two prominent great circles: the equator and the prime meridian. The equator acts as a dividing line between the northern and southern hemispheres, while the prime meridian (which passes through Greenwich, England) separates the Eastern and Western Hemispheres on Earth's surface. These two great circles serve as references for latitude and longitude, which together define the locations of points on Earth's surface.

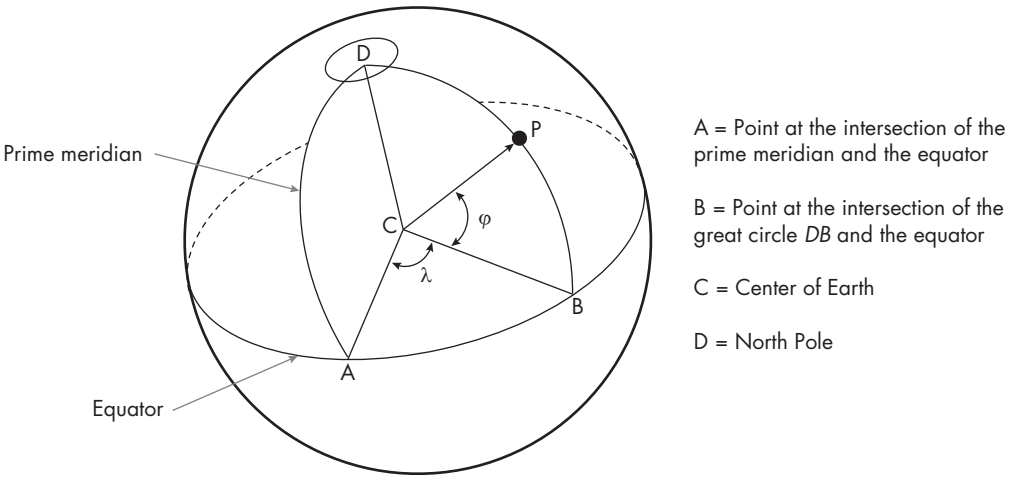


Figure 4-5: The latitude and longitude of a point P

Latitude measures the distance north or south of the equator. It's expressed in degrees, with the equator being 0 degrees latitude, the North Pole 90 degrees north (90°N), and the South Pole 90 degrees south (90°S). In Figure 4-5, the latitude of point P would be denoted as φ°N, as it lies φ degrees north of the equator along a great circle that intersects with P and the North Pole. *Longitude* measures the distance east or west of the prime meridian. The prime meridian itself has a longitude of 0 degrees, and longitude values range from –180 degrees west of the prime meridian to 180 degrees east of the prime meridian. In the case of point P, its longitude would be λ°E, indicating its great circle is λ degrees east of the prime meridian.

NOTE *You may be used to seeing all latitude and longitude values expressed as positive numbers, but for the haversine formula to work, southern latitudes and western longitudes must be negative. Our program will automatically convert coordinates that don't follow this convention.*

Given any two points on the surface of a sphere, you can draw a great circle that intersects with both points, and that great circle will define the shortest path between the two points. If you know the angle θ (in radians)

between the points—that is, the angle formed at the sphere's center by the arc connecting the points—and if you know the radius r of the sphere, you can calculate the distance along the sphere's surface between the two points as follows:

$$d = r\theta \tag{4.5}$$

In the case of Earth, we know the radius R to be about 6,371.009 kilometers, but how do we know the angle between two points on Earth's surface? This is where the haversine formula comes in. It uses the points' latitude and longitude coordinates, and a bit of trigonometry, to determine that angle, which in turn lets us calculate the distance between the points. The formula involves a little-known trigonometric function called the haversine function. The haversine of an angle θ is defined as follows:

$$\text{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) \tag{4.6}$$

The haversine formula calculates a, the haversine of the angle between two points on Earth's surface, as follows:

$$a = \text{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\,\text{hav}(\lambda_2 - \lambda_1)$$

Here (φ_1, λ_1) and (φ_2, λ_2) are the latitude and longitude coordinates of the two points, expressed in radians. To convert from degrees to radians, simply multiply by π and divide by 180.

We now have the haversine of the angle between the two points, but not the angle itself. For that, we can use the a we just calculated and the definition of haversine (Equation 4.6) to solve for the arc angle c:

$$c = 2\sin^{-1}\left(\sqrt{a}\right)$$

Now that we have the angle c, we have everything we need to calculate the distance between the points using Equation 4.5:

$$d = Rc$$

There's one catch, however: d works out to a real number only when $0 \le a \le 1$, but sometimes a can be pushed outside this range due to a floating-point error. To avoid this, we should instead express c as:

$$c = 2\sin^{-1}\sqrt{\max(0.0, \min(a, 1.0))}$$

This constrains the value of a to a range from 0 to 1, preventing any unrealistic results.

The Code

We now have everything we need to write a Kotlin program that calculates the shortest distance between two locations on Earth. For this example, I've hardcoded the locations of two well-known landmarks, Big Ben in London

and the Statue of Liberty in New York, but you can use any locations you want. The code consists of four main segments: an import block and global declarations, the `main()` function, the `printLatLong()` function, and the `haversineDistance()` function. I'll discuss them in the same order.

```
// Import math functions.
import kotlin.math.sin
import kotlin.math.asin
import kotlin.math.cos
import kotlin.math.PI
import kotlin.math.max
import kotlin.math.min
import kotlin.math.pow
import kotlin.math.sqrt

// Define a Location data class.
data class Location(
    val name: String = "",
    var lat: Double,
    val latDir: String,
    var lon: Double,
    val lonDir: String
)

//  global variables and parameters
//  N = north, S = south, E = east, W = west
val L1 = Location(name = "Big Ben", lat=51.5004,
    latDir = "N", lon=0.12143, lonDir = "W")
val L2 = Location(name="Statue of Liberty", lat = 40.689978,
    latDir = "N", lon = 74.045448, lonDir = "W")
val locations = listOf(L1, L2)

val R = 6371.009  // radius of Earth in km

fun main() {
    println("\n*** Measuring Distance Between Two Locations on Earth ***\n")
    printLatLong(category = "input", locations)
    val d = haversineDistance()
    printLatLong(category = "adjusted", locations)
    println("\nThe distance between the two given locations:")
    println("d = ${"%10.2f".format(d)} km")
}

fun printLatLong(category: String, locationsToPrint: List<Location>) {
    when(category) {
        "input" ->
            println("...inputted coordinates...\n")
        "adjusted" ->
            println("\n...adjusted coordinates...\n")
    }
    locationsToPrint.forEach { location -> println(location)}
}
```

```
fun haversineDistance(): Double {
    // Adjust signs based on N-S and E-W directions.
  ❶ for (location in locations) {
        with(location) {
            if(latDir == "S" && lat > 0.0) lat = - lat
            if(lonDir == "W" && lon > 0.0) lon = - lon
        }
    }
    // Calculate the angles in radians.
    val phi1 = L1.lat * PI/180
    val phi2 = L2.lat * PI/180
    val delPhi = phi2 - phi1
    val delLambda = (L2.lon - L1.lon) * PI/180

    // Calculate the distance using haversine formula.
  ❷ val a = sin(delPhi/2).pow(2) +
            cos(phi1) * cos(phi2) *
            sin(delLambda/2).pow(2)
    // Ensure that 0 <= a <= 1 before calculating c.
  ❸ val c = 2 * asin(sqrt(max(0.0, min(1.0, a))))
  ❹ val d = R * c
    return d
}
```

Since the haversine formula needs to use quite a few math functions, we begin by importing those from the `kotlin.math` package. Next, we declare a data class `Location` with five properties: `name` (the name of the location), `lat` (the latitude), `latDir` (the direction of the latitude), `lon` (the longitude), and `lonDir` (the direction of the longitude). We then create two `Location` objects, `L1` and `L2`, representing Big Ben and the Statue of Liberty. We store them together in a list called `locations` so we can efficiently iterate over the locations.

Notice that I've provided the latitude and longitude values as positive numbers in degrees, regardless of which direction they're in. I'm relying on the `latDir` and `lonDir` properties to communicate that extra information. The convention is `N` for north, `S` for south, `E` for east, and `W` for west. In cases where the latitude or longitude of a location is exactly 0, the corresponding direction can be set to `EQ` (equator) or `PM` (prime meridian), although this won't impact the final result. Later, in the `haversineDistance()` function, we ensure that when the `lat` or `lon` direction is `S` or `W`, respectively, the corresponding values are always negative ❶.

The `main()` function prints the provided latitude and longitude values before and after adjustments (if any) using the `printLatLong()` function, makes a single call to the `haversineDistance()` function, and prints the result.

The `printLatLong()` function takes one argument, `category`, which is of type `String`. The `category` parameter is passed to a `when` block to determine which of two messages to print, indicating whether the coordinates have been adjusted for their direction properties. The locations themselves are then printed one at a time using the `forEach()` method of the `locationsToPrint` list. We could have used a regular `for` loop here, but some Kotlin enthusiasts consider `forEach()` to be more idiomatic.

Finally, the `haversineDistance()` function calculates the shortest distance between the two locations on a spherical surface. It first iterates over the locations and negates the latitudes and longitudes if needed ❶, then converts all the latitude and longitude values from degrees to radians. Next, it steps through the equations we discussed on page 150, using the coordinates to calculate a ❷, using a (constrained to between 0 and 1) to calculate the angle c ❸, and using c to calculate and return the haversine distance d between the points ❹.

The Result

When you run the program for the set location and parameter values, the output should appear as follows:

```
*** Measuring Distance Between Two Locations on Earth ***

...inputted coordinates...

Location(name=Big Ben, lat=51.5004, latDir=N, lon=0.12143, lonDir=W)
Location(name=Statue of Liberty, lat=40.689978, latDir=N, lon=74.045448,
lonDir=W)

...adjusted coordinates...

Location(name=Big Ben, lat=51.5004, latDir=N, lon=-0.12143, lonDir=W)
Location(name=Statue of Liberty, lat=40.689978, latDir=N, lon=-74.045448,
lonDir=W)

The distance between the two given locations:
d =   5575.08 km
```

As mentioned on page 148, the haversine calculation assumes that Earth is a perfect sphere, which is not true. In fact, Earth is an oblate spheroid, slightly flattened at the poles and bulging at the equator. To get around this wrinkle, you could use Vincenty's formula, which takes into account the oblate spheroidal shape of Earth by considering Earth's equatorial and polar diameters. Which formula to use really depends on the nature of the problem, as both methods have their strengths and their weaknesses.

EXERCISE

Every time you fly to a distant location for business or pleasure, you might wonder about the minimum flight distance between the departing and arriving cities. Now that you know how to calculate the haversine distance, you can find the answer if you know the latitude and longitude coordinates of the locations. Try finding the shortest distance between the locations listed here. (You can look up the coordinates for these locations online; for the last two cases, you should already know what they are.)

(continued)

- Tokyo, Japan, and Sydney, Australia
- Paris, France, and Lima, Peru
- Dublin, Ireland, and Ankara, Turkey
- The North Pole and South Pole
- Two points where the equator and the prime meridian intersect, located on opposite sides of the globe

As an added step, use the calculated distances and an average speed of air travel to estimate the flight times between these locations.

Project 15: Do Encryption with the Hill Cipher

In today's interconnected world, we constantly share sensitive data like personal information, financial details, and confidential messages. What's to stop unauthorized parties from accessing that information? The answer is *encryption*, a set of techniques for scrambling our data into gibberish that can be deciphered only with the right key. Encryption protects our privacy, safeguards against hackers and cybercriminals, and secures our online transactions.

There are a variety of encryption algorithms in use today. In this project, we'll focus on a particular algorithm called the Hill cipher, developed by American mathematician Lester S. Hill in 1929. According to this method, the *plaintext* (text in plain English or any other language) is divided into blocks of fixed size and represented as vectors. These vectors are then multiplied by a square matrix called the *encryption key*, modulo a specified number, to obtain the *ciphertext* (encrypted text). For decryption, the ciphertext vectors are multiplied by the inverse of the encryption key matrix, modulo the same specified number.

Hill's encryption method can be vulnerable to attacks if we don't choose the encryption key matrix carefully. While it's no longer employed as the sole encryption mechanism, it can still be incorporated into more sophisticated methods and remains a valuable concept to grasp. Plus, exploring Hill's method provides an excellent opportunity to apply and enhance our coding skills in the crucial field of cryptography.

How It Works

The Hill cipher revolves around concepts from linear algebra and modulo operations. I don't expect you to have an in-depth knowledge of these areas of mathematics, but you may wish to review these topics to gain a better insight into how the Hill cipher actually works, as well as its strengths and weaknesses. Here are brief definitions of the key terms that we'll use in this project:

Vector

A one-dimensional sequence of values. For example, [1, 3, 5] is a row vector with three elements.

Matrix

A two-dimensional collection of values, arranged in rows and columns. For example, a 3×3 matrix has three rows and three columns, and a total of nine elements (numbers) that can be real or complex.

Determinant

A single value calculated using the elements of a matrix. The matrix must be square, meaning it has the same number of rows and columns. Say we have the following square matrix A:

$$A = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

Its determinant, denoted by $\det(A)$, $\det A$, or $|A|$, can be calculated as follows:

$$\det(A) = aei + bfg + cdh - ceg - bdi - afh$$

Identity matrix

A square matrix, often denoted as I, where all the elements along a diagonal from the top left to the bottom right have a value of 1 and all other elements have a value of 0. A 3×3 identity matrix looks like this:

$$I = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Inverse matrix

For a given matrix A, its inverse A^{-1} is another matrix such that multiplying the two matrices results in the identity matrix (that is, $AA^{-1} = I$). A matrix must be square to have an inverse, although not all square matrices have one.

Modulo

An operation represented by the symbol % that finds the remainder when one number is divided by another. For example, 5 % 2 is 1. Modulo (*mod* for short) is a multipurpose operator used in various applications, such as determining divisibility, cycling through a range of values, and handling periodic patterns. Hill's algorithm relies on the modulo operation to keep the encrypted and decrypted texts within the same alphabet as the plaintext. Thus, the size of the alphabet serves as the *base* or *modulus* (the number after the % operator) for these operations, guaranteeing valid ciphertext and plaintext representations.

Modular multiplicative inverse (MMI)

For a given integer a and a modulus m, the MMI is positive integer x such that $ax \% m = 1$. The value of x must be less than the modulus. For example, the MMI for 5 modulo 11 is 9, because $(5 * 9) \% 11 = 1$, and 9 is less than 11.

Armed with these definitions, let's now dive into the core encryption and decryption steps employed by the Hill cipher and highlight some of our Kotlin implementation.

For Encryption

1. Define the alphabet. Choose which letters are to be used for writing plain and encrypted messages. For messages written in English, the alphabet size should be at least 26 to include all lowercase letters. We'll also include a period, a space, and a question mark, giving us an alphabet of 29 characters total. The size of the alphabet serves as the modulus.
2. Choose a block size. During encryption and decryption, the message is divided into small blocks of characters, each of the same length. In this exercise, we'll have three characters per block.
3. Generate the encryption key matrix. For the purposes of this project, I've generated the encryption key matrix for you, but if you're curious, it must adhere to these rules:
   a. The matrix must be square and have the same dimension as the block size chosen in step 2. In this case, with a block size of 3, we need a 3×3 matrix.
   b. The determinant of the matrix can't be 0.
   c. The determinant must not share a factor, other than 1, with the modulus from step 1.
4. Prepare the plaintext. Divide the plaintext message into blocks based on the chosen block size. If the last block is smaller than the fixed size, pad it with filler characters. We'll use spaces for padding to ensure that the message remains the same after decryption, with no extra visible characters.
5. Create vectors from the plaintext. Each block of the plaintext must be converted into a numerical vector with the same length as the block size. To assign numerical values to characters, we'll save the alphabet in a single `String` object. We can then map each character in the plaintext to that character's corresponding index in the string. For example, `a` will be mapped to `0`, `b` to `1`, and so on. This way, the block `cab` will become `(2, 0, 1)`, a vector of size 3.
6. Encrypt the message. For each block, carry out the following steps:
   a. Multiply the block's plaintext vector by the key matrix, modulo 29, to generate a ciphertext vector.

b. Convert the numerical values in the ciphertext vector back to text characters using the reverse mapping scheme.

c. Add the ciphered characters to a mutable list of characters, which will become the encrypted message (ciphertext) once all the blocks have been processed.

For Decryption

1. Generate the decryption key matrix. To decipher the encrypted text, we must first create the inverse of the encryption key matrix modulo the specified number. This process involves multiple linear algebraic steps and modulo operations; for simplicity, I've provided the inverse matrix. If you want to work with a different set of key and inverse matrices, you can look up the online tools that will generate those for you.
2. Prepare the ciphertext. Divide the ciphertext into blocks of the specified size used during encryption (3 for this project). This time, no padding is needed, since the padding was already done during encryption.
3. Create vectors from the ciphertext. Use the same character-numbering scheme to convert the ciphertext blocks into ciphertext vectors of size 3.
4. Decrypt the message. For each block, carry out the following steps:

   a. Multiply the ciphertext vector by the decryption key matrix, modulo 29, to generate a deciphered vector.

   b. Convert the numerical values in the deciphered vector back to text characters using the reverse mapping scheme.

   c. Add the deciphered characters to a mutable list of characters, which will become the decrypted message (plaintext) once all the blocks have been processed.

Finally, keep in mind that it's customary to pick an encryption key matrix made up of only integers, preferably falling between 0 and the modulus.

The Code

We're ready to implement the Kotlin code for Hill's encryption and decryption method. The code is organized in a top-down manner, starting with global declarations, continuing with the `main()` function, and ending with a series of short helper functions. We'll review everything in sequence.

Variables and Data Structures

We begin by declaring the variables and data structures needed to implement Hill's method.

```
/*  --- Hill's method for encrypting and decrypting texts --- */

// Declare the key matrix and its inverse.
// keyInv is based on mod 29.
val key = arrayOf(
```

```
    intArrayOf(13, 11, 6),
    intArrayOf(15, 21, 8),
    intArrayOf(5, 7, 9)
)

val keyInv = arrayOf(
    intArrayOf(1, 12, 8),
    intArrayOf(20, 0, 6),
    intArrayOf(0, 3, 20)
)

val dim = key.size
const val alphabet = "abcdefghijklmnopqrstuvwxyz .?"

data class Block(
    val t1: Char,
    val t2: Char,
    val t3: Char,
)

val indexVector = IntArray(dim)
val processedVector = IntArray(dim)
val blocks = mutableListOf<Block>()
val processedText = mutableListOf<Char>()
```

First, we create the matrices for encryption and decryption (`key` and `keyInv`). For this project, we'll accept these as given, but you can use online tools to create a different encryption key matrix that meets the required conditions and calculate the corresponding inverse matrix. The size of these square matrices is captured in the parameter `dim`, which is later used as the block size for processing messages. We also define a string called `alphabet` that stores all the valid letters that can be used in the plaintext and ciphertext.

Next, we introduce a data class called `Block`, which we'll use to store the text blocks generated while processing the message. These blocks will be stored as a mutable list named `blocks`. We also create a few other collections to temporarily hold and manipulate the vectors created during encryption and decryption operations, along with a mutable list named `processedText` to store the final list of characters. Since the encryption and decryption processes are very similar, we'll be able to use these variables and collections during both processes to store the intermediate and final values.

The main() Function

The `main()` function calls a series of helper functions to coordinate the overall encryption or decryption process.

```
fun main() {
    println("\n*** Cryptography with Hill's Method ***\n")
    runValidation()
    println("\nEnter 1 for encryption or 2 for decryption:")

  ❶ when(val choice = readln().toInt()) {
        1 -> {
```

```
            println("You have chosen encryption\n")
            getText()
            encrypt()
            printProcessedText(choice)
        }
        2 -> {
            println("You have chosen decryption\n")
            getText()
            decrypt()
            printProcessedText(choice)
        }
        else -> println("\nInvalid choice...exiting program\n")
    }
}
```

In `main()`, we first call the `runValidation()` function, which uses matrix multiplication (mod 29) to ensure that the encryption and decryption matrices are valid. We then prompt the user to choose which operation to carry out: encryption (enter **`1`**) or decryption (enter **`2`**). Based on the choice, we use a `when` block ❶ to implement the steps to encrypt or decrypt a message.

For both choices, we start with the `getText()` function, which takes in the message to be encrypted or decrypted from the user as a text string and divides it into blocks. We then call `encrypt()` or `decrypt()`, depending on the choice made earlier. Finally, we display the result with help from the `printProcessedText()` function.

The Helper Functions

There are several helper functions called from within the `main()` function. We'll turn to those next, starting with the functions that help validate the matrices.

```
fun runValidation() {
    println("key matrix dimension:")
    println("${key.size}  x  ${key[0].size}\n")

    // validation of key and keyInv
    val productMatrix = multiplyMatricesMod29(key, keyInv,
        r1=dim, c1=dim, c2=dim)
    displayProduct(productMatrix)
}

fun multiplyMatricesMod29(firstMatrix: Array <IntArray>,
                          secondMatrix: Array <IntArray>,
                          r1: Int,
                          c1: Int,
                          c2: Int): Array <IntArray> {
    val product = Array(r1) { IntArray(c2) }
    for (i in 0 until r1) {
        for (j in 0 until c2) {
```

```
            for (k in 0 until c1) {
                product[i][j] += (firstMatrix[i][k] *
                                  secondMatrix[k][j])
            }
          ❶ product[i][j] = product[i][j] % 29
        }
    }
    return product
}

fun displayProduct(product: Array <IntArray>) {
    println("[key * keyInv] mod 29 =")
    for (row in product) {
        for (column in row) {
            print("$column    ")
        }
        println()
    }
}
```

The `runValidation()` function displays the size of the key matrices. It then calls `multiplyMatricesMod29()` to do the validation check and shows the results with `displayProduct()`. The matrices are considered valid if one is the inverse of the other, modulo 29. If this is the case, the product of the two matrices, modulo 29, should be an identity matrix where all elements are zeros, except for ones along the diagonal from the top left to the bottom right.

In `multiplyMatricesMod29()`, we test this out, using three nested `for` loops to multiply the encryption and decryption key matrices, taking modulo 29 of each resulting value before putting it in the `product` matrix ❶. See the "Multiplying Two Matrices" box for details about the math behind this process.

MULTIPLYING TWO MATRICES

To multiply two matrices, we must first ensure that their shapes (number of rows and columns) are compatible: the number of columns in the first matrix must be equal to the number of rows in the second matrix. As a result, the product matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

Once this condition is met, we work through the rows of one matrix and the corresponding columns of the other, multiplying and summing the values. Say we have the following two 3×3 matrices, *A* and *B*:

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}, \quad B = \begin{vmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{vmatrix}$$

Here are the steps to find their product matrix *C*:

1. Multiply the elements in the first row of matrix *A* by the elements in the first column of matrix *B* and add the results together: (1 × 9) + (2 × 6) + (3 × 3) = 9 + 12 + 9 = 30. This is the value at *C*[1,1], the first row and first column of the product matrix.
2. Multiply the elements in the first row of matrix *A* by the elements in the second column of matrix *B* and add the results together: (1 × 8) + (2 × 5) + (3 × 2) = 8 + 10 + 6 = 24. This is the value at *C*[1,2], the first row and second column of the product matrix.
3. Multiply the elements in the first row of matrix *A* by the elements in the third column of matrix *B* and add the results together: (1 × 7) + (2 × 4) + (3 × 1) = 7 + 8 + 3 = 18. This is the value at *C*[1,3], the first row and third column of the product matrix.
4. Repeat steps 1 through 3 for the second and third rows of matrix *A* to get the values in the second and third rows of the product matrix *C*.

   After performing all these calculations, we get this result:

$$A = \begin{vmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{vmatrix}$$

The `displayProduct()` function neatly formats and prints the contents of the product matrix. As you'll later see in the example output, the result should indeed be an identity matrix.

Here's the `getText()` function, which we call from `main()` at the start of the encryption or decryption process:

```
fun getText() {
    println("Enter text for processing:")
    var text = readln().lowercase()
    val tmp = " " // Use a space for padding.

  ❶ when(text.length % 3) {
        1 -> text = text + tmp + tmp
        2 -> text += tmp
    }
    for (i in text.indices step 3)
        blocks.add(Block(text[i], text[i+1], text[i+2]))
}
```

The function uses `readln()` to take in the plaintext or ciphertext from the user. We convert all the characters to lowercase since we have only lowercase letters in our alphabet. We then check if the input string is divisible by `3` ❶ and pad it with spaces if not. Finally, we use a `for` loop with a step size of `3` to break the text into three-character blocks. Each one is stored in a `Block` object and added to the `blocks` mutable list.

The remainder of the helper functions do the work of actually encrypting and decrypting the text.

```
fun encrypt() {
    for (block in blocks) {
        getIndexBlock(block)
        encryptIndexBlock()
        addToProcessedText()
    }
}

fun decrypt() {
    for (block in blocks) {
        getIndexBlock(block)
        decryptIndexBlock()
        addToProcessedText()
    }
}

fun getIndexBlock(block: Block) {
    val (x,y,z) = block
    indexVector[0] = alphabet.indexOf(x)
    indexVector[1] = alphabet.indexOf(y)
    indexVector[2] = alphabet.indexOf(z)
}

fun encryptIndexBlock() {
    for (j in 0 until  3) {
        var sum = 0
        for (i in 0 until  3) {
            sum += indexVector[i] * key[i][j]
        }
        processedVector[j] = sum % 29
    }
}

fun decryptIndexBlock() {
    for (j in 0 until  3) {
        var sum = 0
        for (i in 0 until  3) {
            sum += indexVector[i] * keyInv[i][j]
        }
        processedVector[j] = sum % 29
    }
}

fun addToProcessedText() {
    processedVector.forEach { i ->
        processedText += alphabet[i]
    }
}
```

```
fun printProcessedText(choice: Int) {
    when(choice) {
        1 -> println("\nHere is the encrypted text:")
        2 -> println("\nHere is the decrypted text:")
    }
    print(processedText.joinToString(""))
}
```

The `encrypt()` and `decrypt()` functions both iterate through the `Block` objects in the `blocks` list and call a series of helpers to process them. The first helper called is the `getIndexBlock()` function, which looks up each character's index in the `alphabet` string, thereby converting each character to an integer. The values are stored in the `indexVector` array.

Next, we call `encryptIndexBlock()` or `decryptIndexBlock()`, which converts the plaintext vector into a cipher (encrypted) vector or vice versa by multiplying the vector by the appropriate matrix (`key` or `keyInv`), modulo 29. Multiplying a vector by a matrix is much like multiplying two matrices, but in this case, we need only two levels of `for` loops. The result goes in the `processedVector` array.

Our last encryption and decryption helper is the `addToProcessedText()` function, which takes each number from the `processedVector` array, looks up the corresponding character from the `alphabet` string, and adds that character to `processedText`, a mutable list. In the end, once all the vectors are processed, this list contains the final encrypted or decrypted text. Back in `main()`, we call the `printProcessedText()` function, which concatenates all the characters stored in the `processedText` list into a single string for easy printing.

The Result

Here's a sample run of the program in encryption mode:

```
*** Cryptography with Hill's Method ***

key matrix dimension:
3  x  3

[key * keyInv] mod 29 =
1    0    0
0    1    0
0    0    1

Enter 1 for encryption or 2 for decryption:
1
You have chosen encryption

Enter text for processing:
Code is like humor. It is bad code when you have to explain it.

Here is the encrypted text:
tsgsiomjjnhtvwpqxs.ahk?ru gbn tsgbtynurosksdoqfb a?ujsmtexvjcji
```

First, notice the validation check: the product of the two matrices is indeed an identity matrix, with ones running along the diagonal and zeros everywhere else. Then notice the final output, where the program has turned the readable plaintext into unreadable gibberish. The process also works in reverse: if you choose the decryption option (enter **2**) and input the encrypted text, the program will instantly convert the ciphertext back to the original plaintext.

Currently, the final result is displayed in all lowercase letters. I invite you to improve the `printProcessedText()` function so that the final result is capitalized as needed before printing. If you're thorough, you'll soon realize that implementing a complete set of capitalization rules isn't as simple as it sounds.

Project 16: Simulate a One-Dimensional Random Walk

So far in this chapter, the projects have all been *deterministic*, meaning there's a unique solution for a given set of input parameters. If we were to run the code multiple times with the same input, the output would remain unchanged. In this project, we'll explore a different kind of problem, one that's *stochastic* in nature. In a stochastic problem, the output for a given set of inputs isn't predetermined. We may be aware of various possible outcomes, or a range within which the output will fall, but the specific value generated by an individual instance of the experiment is determined purely by chance. To illustrate this concept, we'll probe the idea of a random walk.

A *random walk* is a process made up of a series of *random steps*, actions with multiple possible outcomes. We know the probability of each potential outcome, but the actual outcome is determined randomly. For example, rolling a die is a type of random step. Assuming the die is fair, each of its six sides will have the same likelihood of landing face up (one-sixth, or approximately 16.67 percent). Therefore, when we actually roll the die, we can't know for sure what number we'll get. Our guesses will be correct only 16.67 percent of the time.

Random walks can be described using a *mathematical space* with a certain number of dimensions, depending on the nature of the random step. Let's say we're considering the movement of a heavily inebriated person who has just come out of a pub. The street in front of the pub runs east–west. This person is totally disoriented and is taking random steps along the street in both directions. We can mathematically describe the distance the person travels over time as the sum of individual steps along the x-direction (the x-axis being the east–west line). We could record each step toward the east as +1 and a step in the opposite direction as –1 (assuming all steps cover the same distance). This is an example of a one-dimensional random walk—we need only the x-axis to describe it mathematically.

Now suppose the person has been drinking in the middle of an open field and has started to wander randomly in different directions. The person's steps can now have both an x-component (east or west) and a y-component (north or south). In this case, to measure the distance

traveled from the center of the field, we'll have to track the person's movements in a two-dimensional space, which will make this a two-dimensional random walk problem.

A well-known example of a random walk is *Brownian motion*, named for Robert Brown, a 19th-century Scottish botanist. Using a microscope, Brown was observing grains of pollen immersed in water when he noticed that the grains were constantly moving in random directions. In fact, we can find similar movements whenever very small particles are injected in a fluid medium, such as dust or smoke particles in the air or the movement of particles in a colloidal suspension such as milk or paint. Brown's observation was an important scientific discovery that remained unexplained for more than half a century until 1905, when Albert Einstein explained that Brownian motion was caused by the continuous bombardment of the pollen grains by the surrounding water molecules.

In this exercise, we'll build and simulate a 1D random walk model in Kotlin. This will allow us to gain a deeper insight into how particles or objects move in one dimension through random steps. In particular, by repeating the simulation many times and plotting the results, we'll be able to identify patterns and explore the statistical properties that underlie this dynamic behavior.

A One-Dimensional Model

Imagine a single particle moving randomly along a line in small steps. For simplicity's sake, we'll assume that the particle's step size remains constant and that steps are made at steady time intervals (we do not need to use time as an explicit variable in our model). Physicists often call this scenario a *free diffusion* problem in one dimension. The process is schematically shown in Figure 4-6.

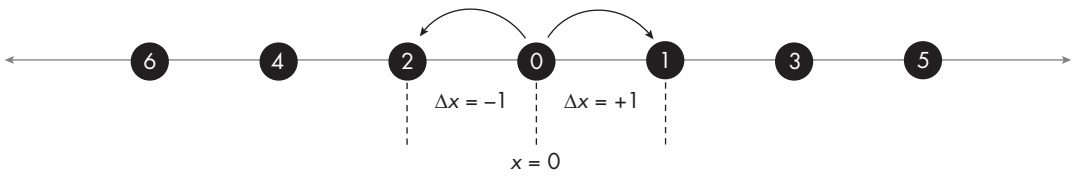


Figure 4-6: A random walk in one dimension, starting at x = 0

The particle starts at location $x = 0$ and moves in discrete steps of length $\Delta x = \pm 1$. The direction of the particle's movement is random, so its next position after 0 can be either 1 (with a displacement of +1) or 2 (with a displacement of –1). The probability p of choosing either direction is equal, so $p = 0.5$. Notice that at any given location, the particle can change its direction, so it's possible for the particle to take several random steps and end up back where it started.

The question we're interested in exploring is this: After making an arbitrary number of steps, n, how far on average will the particle have moved from its starting position? To answer this question, we'll need to simulate

many random walks—say, 500 walks of 1,000 steps each—and analyze the results. We can't simply take the average of the cumulative distances traveled in the different simulations, however; the particle can drift in both the positive and negative x-directions, so the net-positive and net-negative distances would likely cancel each other out, giving us an average distance of roughly 0. Instead, we'll use the *root-mean-square (RMS) distance*, which is calculated in three steps:

1. Square all the distances from all the simulations after a given number of steps *n*. This converts any negative numbers to positive numbers.
2. Add all the results from step 1 and divide by the number of simulations to find the mean (average) of the squared distances.
3. Take the square root of step 2's result to arrive at the RMS distance.

Based on past research conducted on one-dimensional random walks, we know that the RMS distance exhibits a nonlinear relationship with the number of steps taken; in theory, the RMS distance after the *n*th step should equal the square root of *n*. To validate this notion, we'll compute the RMS distance (the *simulated* RMS) and the square root of *n* (the *theoretical* RMS) and plot them both against the number of steps, *n*. Hopefully, the two plots will be similar. We'll also plot the mean distance traveled at each time step, which should remain close to 0.

In a separate graph, we'll visualize the trajectories of the 500 random walks themselves. This should help illustrate the random nature of the outcomes and give further support to our theories about the cumulative and RMS distances traveled.

The Code

I'll present the code segments for this project in a top-down sequence, starting with some general setup code. Since we want to visualize the random walks from different simulations and examine the relationship between the cumulative, mean, and RMS distances with the number of steps, we'll use the JavaFX template that we developed and used in Chapter 3.

```
// import block
import javafx.application.Application
import javafx.scene.Node
import javafx.scene.Scene
import javafx.scene.chart.LineChart
import javafx.scene.chart.NumberAxis
import javafx.scene.chart.XYChart
import javafx.scene.control.ScrollPane
import javafx.scene.layout.Background
import javafx.scene.layout.BackgroundFill
import javafx.scene.layout.CornerRadii
import javafx.geometry.Insets
import javafx.scene.layout.VBox
```

```
import javafx.scene.paint.Color
import javafx.stage.Stage
import kotlin.math.sqrt

// data class
❶ data class State(
    var step: Double,
    var dist: Double
)

// global parameters
val numStep = 1000
val numSim = 500

❷ // Create lists needed for plotting line charts.
val xList  : List<State> = List(numStep) { State(0.0, 0.0) }
val avgList: List<State> = List(numStep) { State(0.0, 0.0) }
val rmsList: List<State> = List(numStep) { State(0.0, 0.0) }
val expList: List<State> = List(numStep) { State(0.0, 0.0) }

val states1 = mutableListOf<List<State>>()
val states2 = mutableListOf<List<State>>()

class RandomWalk1D : Application() {
    override fun start(primaryStage: Stage) {
      ❸ val root = VBox()
        /*----------------------------------------*/
      ❹ root.styleClass.add("color-palette")
        root.background = Background(BackgroundFill(Color.WHITE,
             CornerRadii.EMPTY, Insets.EMPTY))
        /*----------------------------------------*/
      ❺ val scroll = ScrollPane()
        scroll.setContent = root
        val scene = Scene(scroll, 550.0, 850.0, Color.WHITE)
        primaryStage.title = "1D random Walk Simulation"
        primaryStage.scene = scene
        primaryStage.show()

        // ----- Random walk simulation starts here. -----
        // Call random walk function.
        randomWalk1d()
        // Get the theoretical RMS values.
        calcRMS1d()
        // Create line charts.
        createRWChart1(root)
        createRWChart2(root)
    }
}

fun main() {
    Application.launch(RandomWalk1D::class.java)
}
```

The code segment starts with the import block. Since this project will use the XY charting features of JavaFX instead of the canvas feature, the import block is somewhat different from what we needed for Project 13, and it includes a few extra lines of code to import the `Background`, `BackgroundFill`, `CornerRadii`, and `Insets` features, which we'll use to set the chart background to white.

Next, we declare a simple data class `State` ❶ for holding individual data points during the simulation. Its `step` property represents the number of steps taken since the beginning of the random walk, and `dist` is the cumulative distance traveled after that many steps. We then declare two global parameters: `numStep`, to specify the maximum number of steps per simulation, and `numSim`, to set the maximum number of simulations.

We'll accumulate data in a number of lists ❷, each of size `numStep` and type `State`, as follows:

`xList` Stores the cumulative distance traveled after each step for a particular simulation

`avgList` Stores the arithmetic average (mean) of the cumulative distances traveled across all simulations after each step

`rmsList` Stores the RMS distance calculated across all simulations after each step

`expList` Stores the theoretical (exponential) RMS distance after each step

All these lists are initialized to `(0.0, 0.0)`, meaning all simulations start at step number 0 and position 0. In addition to these lists, we also create two mutable lists, `states1` and `states2`, which we'll use for charting purposes.

Inside the `RandomWalk1D` application class, we use a `VBox` container ❸ to hold the chart objects, as we'll generate two sets of charts that will be placed vertically inside the `VBox`. Notice the additional lines of code for setting the background of the container to white programmatically ❹, without using the cascading style sheets needed for more extensive customizations. We've also introduced the `ScrollPane` feature ❺, which will allow us to scroll the chart window to view the top or the bottom chart, as needed. We can also enlarge the window to make both charts visible at the same time.

After setting up the graphics window, we call three custom functions that will run the simulation and help visualize the results. The first call is to the `randomWalk1d()` function, which simulates `numSim` one-dimensional random walks over `numStep` steps. Here's how it works:

```
fun randomWalk1d() {
    // Create local arrays.
  ❶ val s = Array (numSim) {DoubleArray(numStep)}
    val sumX = DoubleArray(numStep)
    val sumX2 = DoubleArray(numStep)

    // Walk numStep steps numSim times.
    for (i in 0 until numSim) {
        var draw: Int
        var step: Int
```

```
        for (j in 1 until numStep) {
          ❷ draw = (0..1).random()
            step = if (draw == 0) -1 else 1
          ❸ s[i][j] = s[i][j-1] + step
            sumX[j] += s[i][j]
            sumX2[j] += (s[i][j] * s[i][j])
            xList[j].step = j.toDouble()
            xList[j].dist = s[i][j]
        }
      ❹ states1.add(xList.map {it.copy()})
    }

    // Create average (mean) and RMS for distances traveled.
    for (j in 0 until numStep) {
        avgList[j].step = j.toDouble()
        avgList[j].dist = sumX[j] / numSim
        rmsList[j].step = j.toDouble()
        rmsList[j].dist = sqrt(sumX2[j] / numSim)
    }
  ❺ states2.addAll(listOf(avgList, rmsList))
}
```

The function body starts by creating three local arrays of type `DoubleArray`. The first, `s`, is a two-dimensional array that stores the cumulative distance traveled at each step of each simulation ❶. The others are one-dimensional arrays, `sumX` and `sumX2`, to save the running sums of the cumulative distances at each step and the sums of squared distances at each step, respectively. We'll use these values to get the mean and RMS distances.

The random walks are implemented inside a nested `for` loop. The outer loop controls the number of simulations, and the inner loop makes the particle take `numStep` steps in succession. During each step, a local variable `draw` is randomly set to either `0` or `1` with equal likelihood ❷. Based on the outcome, `step` (referred to as Δx in Figure 4-6) is set to `-1` or `1`, which is then added to the cumulative distance traveled up to the previous step of the simulation ❸. These cumulative distances are used to create the elements of `xList`, which is then copied and passed on to `states1` once per simulation ❹. Notice how we're reusing the memory allocated for `xList` during each simulation by overwriting the values of its elements. In the end, `states1` has all the data we need to visualize the random walks themselves.

Once we're done with the random walks, we use the resulting lists, `sumX` and `sumX2`, to create the `avgList` and `rmsList` inside another `for` loop by dividing the elements of `sumX` and `sumX2` by `numSim`. Here `sumX[j]` is the sum of all the elements in column `j` of the `s[i][j]` matrix, where `i` represents the simulation number and `j` represents the number of steps taken so far. (Likewise, `sumX2[j]` is the same, squared.) Finally, `avgList` and `rmsList` are passed on as elements of `states2` ❺, which we defined earlier as a list of lists.

The second function call inside the application class is `calcRMS1d()`. It generates the theoretical RMS distance at each step:

```
fun calcRMS1d() {
    // Create the theoretical (exponential) rms/list.
```

```
    for (j in 0 until numStep) {
        expList[j].step = j.toDouble()
      ❶ expList[j].dist = sqrt(j.toDouble())
    }
    states2.add(expList)
}
```

We know from the theoretical analysis of the one-dimensional random walk problem that the RMS distance is a nonlinear function of the number of steps n, which can be expressed as $x_n = \sqrt{n}$, where x_n is the RMS distance for the nth step (n is equivalent to looping variable `j` in the code). We use this relationship in the `calcRMS1d()` function to calculate the theoretical RMS distances and update `expList` ❶. We'll use this list to create a side-by-side plot of the theoretical and simulated RMS distances to see how closely they follow each other.

In the last two lines of the application class, we make two successive calls to the `createRWChart1()` and `createRWChart2()` functions, shown here:

```
fun createRWChart1(root: VBox) {
    val xyChart1 =
        singleXYChart(states1,
            title = "Random Walk 1D Experiment",
            xLabel = "Steps",
            yLabel = "Cumulative distance traveled")
    root.children.add(xyChart1)
}

fun createRWChart2(root: VBox) {
    val xyChart2 =
        singleXYChart(states2,
            title = "Random Walk 1D Experiment",
            xLabel = "Steps",
            yLabel = "Mean and RMS distance traveled")
    root.children.add(xyChart2)
}
```

Other than the chart labels, the only difference between these two functions is that the first one uses `states1` and the second one uses `states2` to generate the respective charts. Both of these functions call the `singleXYChart()` function (which we discussed and used in Chapter 3) to draw the line charts and stack them inside a scroll pane.

The Result

When you run the full code on your device, you should see a single scrollable window pop up with two separate charts. Let's first consider the visualization of the random walks themselves, shown in Figure 4-7.

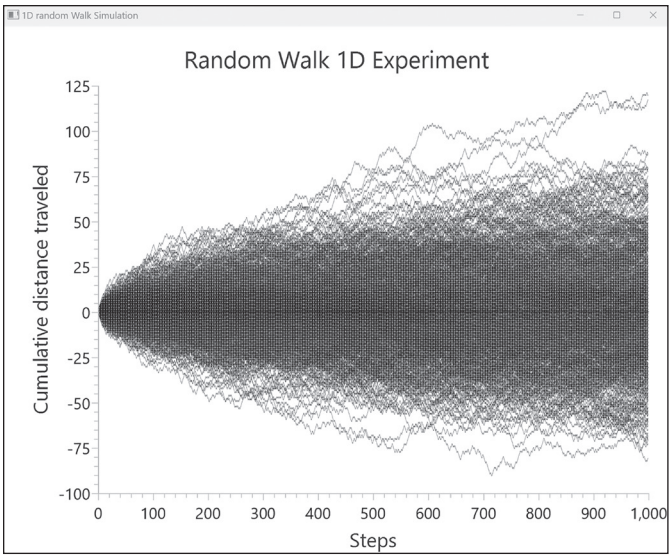


Figure 4-7: The random walk paths from 500 simulations of 1,000 steps

This chart shows all 500 random paths generated by the `randomWalk1d()` function (one per simulation, based on our global `numSim` parameter). These paths show a number of key features of one-dimensional random walks:

- Each random path is unique, evident from the tangled web of lines moving across the chart.
- Most random walks tend to stay close to their starting position, even after many steps. We can see this in the darker band along the x-axis.
- The paths are equally dispersed on both sides of $x = 0$, as expected. You could confirm this by creating histograms of the cumulative distances traveled at different numbers of steps. (I'll leave this for you to try out as an exercise.)
- For any given number of steps, if we add the cumulative distances from all simulations, the sum will be close to zero because positive and negative distances will cancel each other. For the same reason, the arithmetic mean will also be close to zero.
- The RMS distance increases with the number of steps, as confirmed by the gradual widening of the band that envelops all random paths. The RMS distance is therefore a better measure for the average distance traveled than the arithmetic mean, as we don't care about the direction of the movement.

All these points collectively provide the answer we were seeking at the start of this project. A particle moving randomly in one dimension will likely trace a path that will initially stay close to its starting position.

However, if we follow the particle for a long time, it may gradually move farther away. Again, we can't predict exactly how far a particular particle will move, but if we measure the RMS distance from many different particles, we'll see that the RMS distance increases with the number of steps. Our other chart, shown in Figure 4-8, helps us explore this last point further.

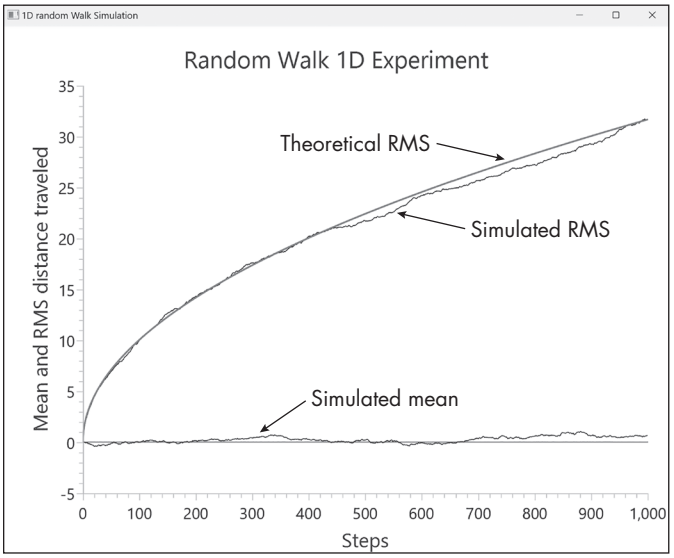


Figure 4-8: The mean and RMS distances traveled for a given number of steps

Figure 4-8 charts three lines. First, we have the line labeled "Simulated mean," generated from the data in `avgList`. This line stays very close to zero, confirming one of the key points made based on Figure 4-7: that the arithmetic average or mean for any number of steps will be zero if we have a sufficiently large number of observations. Second, we have the "Simulated RMS" line, generated from the data in `rmsList`, which clearly shows the RMS distance increasing (although at a decreasing rate) with the number of steps. Third, the smooth "Theoretical RMS" line represents the theoretical RMS distances from `expList`, calculated by simply taking the square root of the number of steps. Again, we can visually confirm that the simulated RMS values are very close to the theoretically expected RMS values.

The minor discrepancies we see between the simulated and theoretical RMS values are to be expected. The simulated RMS values will approach the theoretical values as the number of simulations approaches infinity. I invite you to run the code again, this time setting `numSim` to `5000`. Make sure that you comment out the call to the `createRWChart1()` function before doing that. The default implementation of JavaFX is memory and computation intensive, and trying to plot 5,000 lines, each with 1,000 data points, can take a while depending on your processor and memory configuration. However, if you do this experiment as suggested, you'll see that with the added

random walks, the simulated and theoretical RMS lines become virtually the same. If you go further by setting `numSim` to `50000`, you'll see only one line.

EXERCISE

Now that you've seen how to model a random walk in one dimension, try extending your random walk app to model molecular diffusion in two dimensions. The effect should be similar to adding a drop of food coloring to the center of a dish of water and watching the color spread.

Hint: To be able to track distances in 2D, the `State` data class has to be changed to replace `dist` with two values for the x- and y-components—say, `distX` and `distY`. At each step in the random walk, treat these values separately. For the x-component, for example, assume that a particle can either stay still, move toward the positive x-direction, or move toward the negative x-direction. You can code this as `stepX = (-1..1).random()`, meaning `stepX` is equally likely to be `-1`, `0`, or `1`. Repeat the same for the y-component.

Define how large you want the dish to be (set an appropriate diameter), and decide what happens when a particle hits the boundary of the dish. (Perhaps the particle should bounce off the wall.) Update the particle positions inside a nested `for` loop, and update the series for charting using the `ScatterChart` feature of JavaFX. Start with 300 particles. To show particle positions dynamically (which looks really cool), run the random walk function and update the data points as an animation, using `Timeline` and `KeyFrame`, as discussed in Chapter 3.

Summary

In this chapter, we used Kotlin code and custom algorithms to solve math-related problems. The problems weren't just theoretical; they also had practical applications in fields like mathematics, geodesy, navigation, and cryptography. Throughout our journey, we employed various mathematical concepts, operations, and tools, including basic arithmetic, math and trigonometric functions, the Pythagorean theorem, the Fibonacci sequence, the haversine formula, modulo operations, and linear algebra. We also probed the realm of stochastic processes, exploring the generation and utilization of random numbers to simulate random phenomena.

Along the way, we used many core features of Kotlin, such as variables and collections, data classes, and conditional and iterative structures like `if`, `when`, `for`, and `while`. We also discovered the convenience of functions and lambdas, along with the rich set of mathematical and graphics library functions at our disposal.

Resources

Ayars, Eric. "Stochastic Methods." In *Computational Physics with Python*, 131–139. August 18, 2013. Accessed June 15, 2024. *https://belglas.files.wordpress.com/2018/03/cpwp.pdf*.

Dutka, Jacques. "Eratosthenes' Measurement of the Earth Reconsidered." *Archive for History of Exact Sciences* 46, no. 1 (1993): 55–66. Accessed June 15, 2024. *http://www.jstor.org/stable/41134135*.

Eisenberg, Murray. "Hill Ciphers and Modular Linear Algebra." November 3, 1999. Accessed June 15, 2024. *https://apprendre-en-ligne.net/crypto/hill/Hillciph.pdf*.

Harder, Douglas. "Project H.1: Sieve of Eratosthenes." University of Waterloo. Accessed June 15, 2024. *https://ece.uwaterloo.ca/~ece150/Programming_challenges/H/1/*.

Kereki, Federico. "A Modern Look at Square Roots in the Babylonian Way." *Cantor's Paradise*. December 7, 2020. Accessed June 15, 2024. *https://medium.com/cantors-paradise/a-modern-look-at-square-roots-in-the-babylonian-way-ccd48a5e8716*.

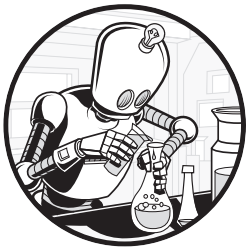
"Pythagorean Triples." Prime Glossary. Accessed June 15, 2024. *https://t5k.org/glossary/page.php?sort=PrmPythagTriples*.

Reich, Dan. "The Fibonacci Sequence, Spirals and the Golden Mean." Department of Mathematics, Temple University. Accessed June 15, 2024. *https://math.temple.edu/~reich/Fib/fibo.html*.

Van Brummelen, Glen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry*. Princeton, NJ: Princeton University Press, 2013.

5

MODELING AND SIMULATION



In this chapter, we'll venture into several mini science projects that will further strengthen your coding abilities in Kotlin. In addition, the mini projects of this chapter will enhance your problem-solving skills by showing how to apply basic laws of nature and mathematics to solve problems that are very difficult to answer without the help of a computer.

Since visualization is a key part of scientific investigation, we'll also use the graphics templates we developed in Chapter 3 to visualize our program outputs.

Project 17: Predict the Flight of a Cannonball

Imagine an old castle surrounded by thick defensive walls. The walls have notches where vigilant soldiers stand ready to fire cannons on any

encroaching enemy. Typically, a castle is located on high ground to allow for better visibility over the surrounding area, and the cannons themselves are placed at a higher elevation relative to the base of the castle to maximize their firing range. Within that range, the soldiers can hit targets at a variety of distances by adjusting parameters of the cannon. Our question is, how can the soldiers aim a cannon to hit a target on the ground x distance away?

Assuming no air resistance for now and assuming that the impact of wind is negligible, the distance a cannonball travels is affected by three factors: the cannon's height above ground level, its angle of fire, and the exit velocity of the cannonball. The soldiers can't easily change the height of the cannon or the exit velocity of the cannonball, but they can adjust the angle of fire, so that's where we'll focus our efforts. Figure 5-1 diagrams the nature of the problem.

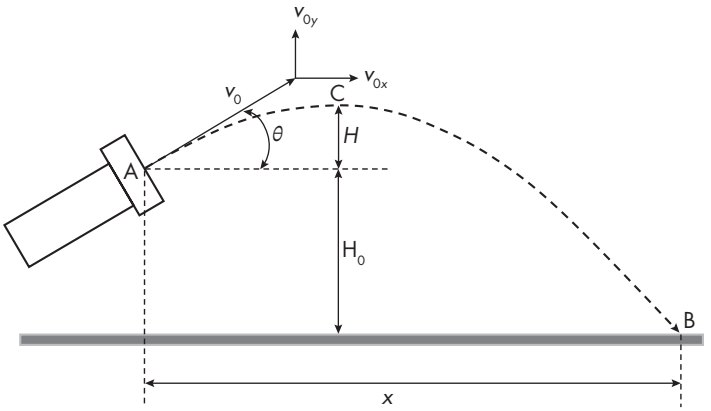


Figure 5-1: The path of a cannonball

The cannonball is fired from point A, which has an elevation of H_0 relative to the ground surface, and the goal is to hit point B. The exit velocity of the cannonball is v_0, and its angle of fire is θ.

Once the cannonball is fired, it becomes a *projectile*, an object in motion that has only one force acting upon it: gravity. To arrive at the correct value for θ, we need to know a little bit about the science of projectiles. First, let's assume that the cannonball is moving in only two dimensions: horizontally and vertically. We'll place the origin of this two-dimensional coordinate system at point A. Given the initial velocity of the ball v_0, which is projected at an angle θ, we can write the corresponding x- and y-components of the initial velocity as:

$$v_{0x} = v_0 \cos\theta$$

$$v_{0y} = v_0 \sin\theta$$

These velocity components tell us how fast the ball will be traveling in the horizontal and vertical directions solely due to its initial velocity. Again, disregarding the effects of air or wind resistance, no other force is acting

on the ball horizontally. Therefore, we could say that the velocity in the x-direction, v_x, remains constant:

$$v_x = v_{0x} = v_0 \cos\theta$$

In the vertical direction, however, the acceleration due to gravity will slow the ball's rise and eventually bring it down to the ground. Thus, we can express the resultant velocity in the y-direction, v_y, as:

$$v_y = v_{0y} - g\,t = v_0 \sin\theta - gt$$

Here g is the acceleration due to gravity, and t is time.

Using these expressions for v_x and v_y, we can write the *displacement equations* for the projectile, which express its x- and y-coordinates, or how far the ball has traveled horizontally and vertically, as of a given time t:

$$x = v_0 \cos\theta\, t \tag{5.1}$$

$$y = (v_0 \sin\theta)\, t - \frac{1}{2} g t^2 \tag{5.2}$$

If we now assume that the time to reach the target at point B is t_B, then the horizontal distance to the target is:

$$x_B = v_0 \cos\theta\, t_B \tag{5.3}$$

Over the same time period t_B, the projectile will also move vertically. It will pass through point C (shown in Figure 5-1, the ball's highest elevation, at $H + H_0$) and then free-fall to the ground while continuing to move horizontally at velocity v_x. When the projectile reaches point B (meaning it hits the target), its net vertical displacement y_B will be $-H_0$, which can be expressed using Equation 5.2 as:

$$-H_0 = (v_0 \sin\theta)\, t_B - \frac{1}{2} g t_B^{\,2} \tag{5.4}$$

We don't actually know how long it will take the cannonball to reach the target, however, so we need a way to express t_B in terms of the other variables. Equation 5.4 is a *quadratic equation*, where the highest power of a variable is 2. It can be solved for the positive value of t_B as:

$$t_B = \frac{1}{g}\left(v_0 \sin\theta + \sqrt{(v_0 \sin\theta)^2 + 2gH_0}\right) \tag{5.5}$$

We can now use Equation 5.5 to substitute for t_B in Equation 5.3, which gives us the following equation for x_B as a function of θ:

$$x_B = \frac{v_0 \cos\theta}{g}\left(v_0 \sin\theta + \sqrt{(v_0 \sin\theta)^2 + 2gH_0}\right) \tag{5.6}$$

This equation establishes a relationship between the distance to the target x_B and the firing angle θ, but it still doesn't allow us to calculate θ for a given x_B. What we need is an equation that expresses θ in terms of x_B, not the other way around.

We have two options to resolve this situation. One is analytical, involving rearranging Equation 5.6 and deriving an expression for θ in terms of x_B. It can be done, but it would require a significant amount of mathematical juggling. The other option is to use a *numerical method*, which involves solving the problem through an iterative process following an algorithm. A numerical method is also an approximate method, meaning it won't produce the exact same solution as the theoretical one. However, we can choose the desired degree of precision, and often that's close enough. In fact, for many real-world problems of this nature, we may not have any known theoretical solution, in which case a numerical method is the only viable option for solving the problem. This is why we'll use a numerical method to answer our cannonball question.

The Strategy

We need to take one additional step to make the problem amenable to a numerical method: getting x_B onto the right-hand side of Equation 5.6 and turning it into a function of θ. In other words, we'll define function $f(\theta)$ as:

$$f(\theta) = x_B - \frac{v_0 \cos\theta}{g}\left(v_0 \sin\theta + \sqrt{(v_0 \sin\theta)^2 + 2gH_0}\right) \tag{5.7}$$

To solve our cannonball problem, we now need to find a value of the independent variable θ that, for a given value of x_B, will make the right-hand side of Equation 5.7 equal to zero. In mathematical terms, this value is known as the *root* of the function. We've turned the projectile problem into a root-finding problem.

We could use a number of methods to find the root for Equation 5.7. In this case, we'll use a simple method called *bisection*, as illustrated in Figure 5-2.

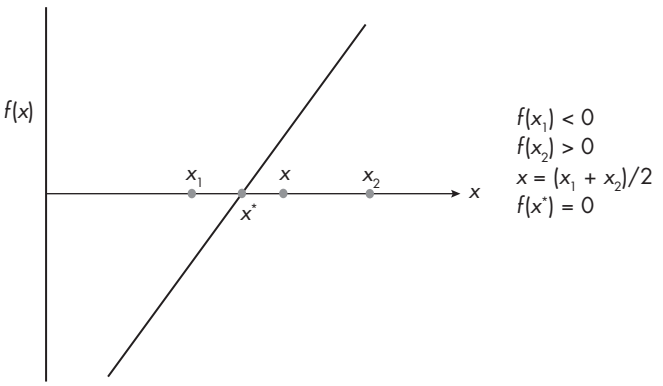


Figure 5-2: The elements of the bisection method for root finding

Let's assume we're trying to find the root of a simple linear function $f(x)$, which is a continuous function of variable x, as shown in Figure 5-2. The root is the value of x that makes the function equal zero. Here are the simple bisection steps for finding the root x^* so that $f(x^*) = 0$:

1. Locate two values x_1 and x_2 such that they are on the opposite side of the root, meaning $f(x_1) < 0$ and $f(x_2) > 0$.
2. Find the midpoint x between x_1 and x_2 such that $x = (x_1 + x_2)/2$.
3. If the absolute value of $f(x)$ is less than some tolerance factor (for example, 0.0000001), then x is the root, so we're done.
4. Otherwise, if $f(x) * f(x_2) > 0$, set $x_2 = x$, else set $x_1 = x$.
5. Repeat steps 2 through 4 until the condition in step 3 is met.

This method is guaranteed to converge to the root if $f(x)$ is continuous within the initial range $[x_1, x_2]$, which also includes the root or roots. Without further ado, let's start coding!

The Code

The bisection method involves only a few lines of code, as shown here:

```
❶ // Import Kotlin math functions.
  import kotlin.math.sin
  import kotlin.math.cos
  import kotlin.math.PI
  import kotlin.math.sqrt
  import kotlin.math.pow
  import kotlin.math.abs

❷ // Set global parameters.
  val v0 = 70             // m/s
  val g = 9.8             // m/s2
  val h0 = 25             // m
  val target = 400        // m
  val TOL = 1.0e-7

  private val f = :: projectile

❸ // The interval [x1, x2] needs to enclose the root.
  val x1 = 10.0    // in degrees
  val x2 = 30.0    // in degrees

  fun main() {
      println("\n*** Firing angle for hitting a target ***\n")

    ❹ if (f(x1) * f(x2) < 0) {
          println("...Initial guesses are valid...")
          val root = bisection(x1, x2)
          val rootFormatted = String.format("%.2f", root)
          println("The firing angle to hit the target is:" +
                  "\n$rootFormatted degrees")
      } else {
          println("\n...Initial guesses are not valid...\n")
      }
  }
```

```
fun projectile(angle: Double): Double {
  ❺ val x = angle * PI / 180.0
    return target - (v0 * cos(x) / g) *
            (v0 * sin(x) + sqrt((v0 * sin(x)).pow(2) + 2 * g * h0))
}

fun bisection(_x1: Double, _x2: Double): Double {
    var x1 = _x1
    var x2 = _x2
  ❻ var x = (x1 + x2) / 2.0

  ❼ while (abs(f(x)) >= TOL) {
        if (f(x) * f(x2) > 0 ) {
            x2 = x
        } else x1 = x
        x = (x1 + x2) / 2.0
    }
    return x
}
```

We start by importing the `math` library functions ❶ and providing the global parameter values ❷. Notice that we import only the math functions we need, separately, instead of using `import kotlin.math.*` to import all math functions. It's good coding etiquette to use only what you need, as it helps keep the namespace clean and reduces the chance of introducing bugs, especially when the code is reused.

In this example, we've set the target distance to 400 m, the cannon's height to 25 m, and the cannonball's initial velocity to 70 m/s. Next, we guess initial values for `x1` and `x2` for the bisection method ❸ and set the function parameter `f` to `projectile`. Treating `projectile` as a parameter allows us to reuse the code to find the roots of other functions.

The `main()` function uses an `if...else` block to confirm that the initial `x1` and `x2` guesses are located on opposite sides of the root ❹. If true, it calls the `bisection()` function to find the root. If the test fails, the program prints a message indicating that the initial guesses weren't valid.

The next code block defines the `projectile()` function, which calculates the right-hand side of Equation 5.7 for a given angle, after first converting the angle to radians ❺ as required for the `sin()` and `cos()` functions.

The final code block is the `bisection()` function that implements the steps outlined earlier. The function takes the two initial guesses as arguments, finds the midpoint between them ❻, and then uses a `while` loop ❼ to make smaller and smaller bisections until the midpoint is within the specified tolerance range. The function returns the root as `x` (of type `Double`).

The Result

If you run the program, the output should look like this:

```
*** Firing angle for hitting a target ***

...Initial guesses are valid...
The firing angle to hit the target is:
21.91 degrees
```

Finally, we have the answer to the cannonball question! For the given initial velocity and height of the cannon above the ground, the soldiers need to fire the cannon at an angle of 21.91 degrees to hit a target 400 m away.

Let's think about this solution a bit more. Is this the only possible way to hit the target? Figure 5-3 shows how $f(\theta)$ varies with θ (for the given parameter values).

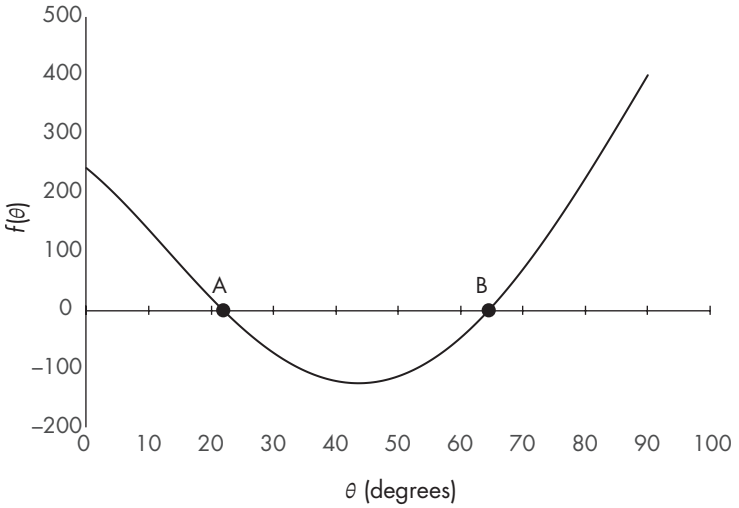


Figure 5-3: The relationship between f(θ) and θ

The function follows a parabolic curve, passing through the x-axis twice, so it has two different roots for $\theta \geq 0$. From the given initial guesses, we've found the first root at point A, but another root is found at point B, where $\theta = 64.51$ degrees. This is a much steeper angle than the first solution, $\theta = 21.91$ degrees. Figure 5-4 shows the paths, or *trajectories*, the cannonball will take for both firing angles.

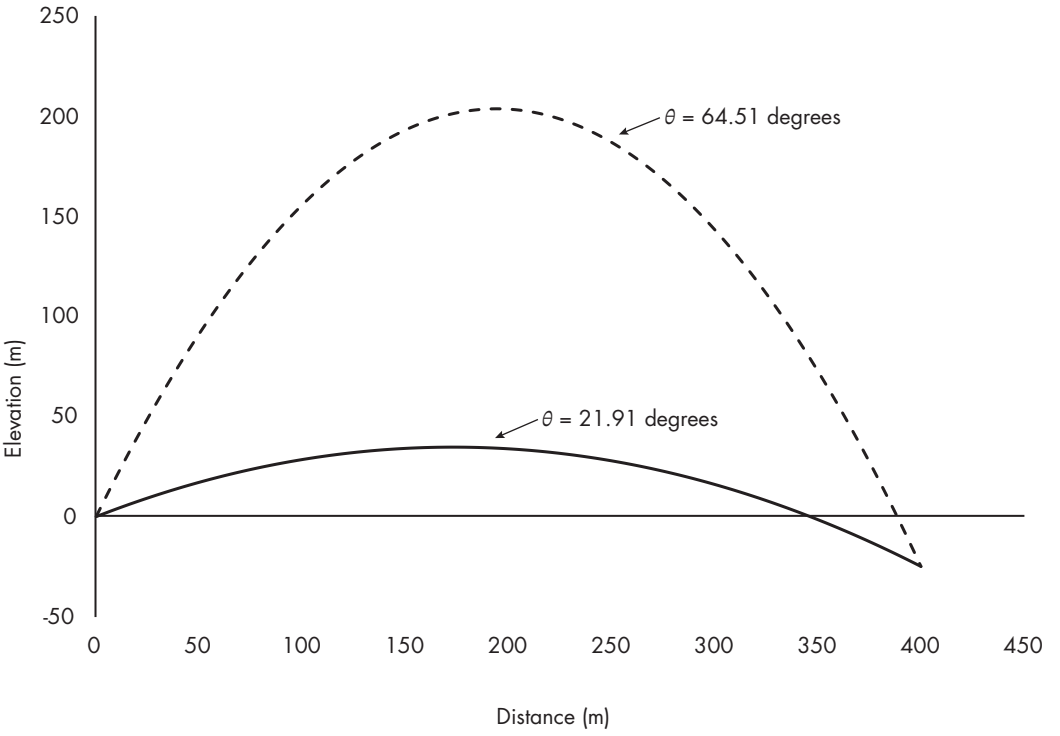


Figure 5-4: Two trajectories for the cannonball solutions

While cannonballs fired at either angle will hit the target, the ball will reach a much higher elevation and take longer to hit the target when $\theta = 64.51$ degrees. Which option do you think the soldiers will prefer, and why?

EXERCISE

Using your understanding of projectiles and the equations we've discussed, write a program to generate the data required to plot Figures 5-3 and 5-4. Then try creating these figures by using the data and the visualization tools presented in Chapter 3.

Project 18: Design a Fountain with Water Jets

In this project, we'll continue to explore the path of a projectile, but instead of chasing a cannonball, we'll follow a jet of water. Water jets have many familiar applications, such as washing cars, watering lawns, and fighting

fires. We'll explore a more artistic example: running the water jet through a nozzle assembly to create a decorative fountain.

A *nozzle* is a narrow opening through which a jet of fluid comes out at a high velocity (the fluid can be a gas or a liquid). For a given pressure, the narrower the opening of the nozzle, the faster the velocity of the jet. In the nozzle assembly used in fountains, several nozzles of different sizes or with adjustable openings can be placed at different angles. Designers vary these parameters to create beautiful patterns with the water jets. In this case, we'll try to adjust the nozzle to shoot jets into different levels of a multilevel fountain. Figure 5-5 illustrates the problem.

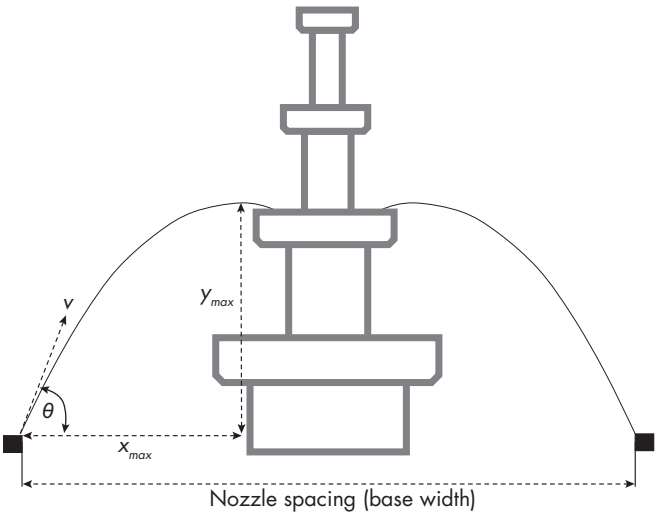


Figure 5-5: Pointing a nozzle toward a multilevel water fountain

The objective of this exercise is to estimate the velocity of a jet v and its angle of ejection θ, given the desired highest point of the jet's trajectory, (x_{max}, y_{max}). This high point is set so that the jet will just pass over the edge of the water basin at a particular level, as shown in Figure 5-5. The fountain has four levels, so we have four sets of (x_{max}, y_{max}), shown in Table 5-1.

Table 5-1: Fountain Parameters

Level	x_{max} (meters)	y_{max} (meters)
1	2.25	1.5
2	2.55	3.0
3	2.85	4.25
4	3.0	5.5

The values in this table are defined relative to the nozzle at point (0, 0). We'll use them to calculate the corresponding values of v and θ. Further,

we'll draw the trajectories of the jets by using the canvas feature of JavaFX so that we can visualize the solution relative to the water fountain.

Notice that Figure 5-5 shows only a cross section of the fountain system, which is circular in shape in three dimensions. The nozzles are placed at diametrically opposite positions along the base of the fountain system, which is made up of the ground-level basin (not shown), the nozzles, and the four-level middle structure. We'll use the spacing between the nozzles as the base width of this fountain system.

The Strategy

We need to use three specific equations to solve this problem, all of which can be derived from the projectile equations discussed in Project 17 (search online for "projectile motion" or "equations of motion" to learn about the derivations, or see the resources listed at the end of the chapter).

$$t_{max} = \frac{v\sin\theta}{g} \tag{5.8}$$

$$x_{max} = \frac{v^2\sin\theta\cos\theta}{g} \tag{5.9}$$

$$y_{max} = \frac{v^2\sin^2\theta}{2g} \tag{5.10}$$

We've already defined the variables in these equations, except for t_{max}, which is the time needed for a water particle leaving the nozzle to reach the highest point (x_{max}, y_{max}). We can use Equations 5.9 and 5.10 to find the expressions for θ and v as:

$$\theta = \arctan\left(\frac{2y_{max}}{x_{max}}\right) \tag{5.11}$$

$$v = \frac{\sqrt{2gy_{max}}}{\sin\theta} \tag{5.12}$$

We'll use these equations to calculate the angles of ejection and velocities for the four levels of the fountain, based on the data provided in Table 5-1. We'll also use Equation 5.8 to calculate the trajectories to be drawn with the JavaFX library.

The Code

Our project uses the same basic JavaFX template that's discussed in detail in Chapter 3. We begin with some overall setup.

```
// Import JavaFX features.
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
```

```
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.stage.Stage

// Import required math functions.
import kotlin.math.sin
import kotlin.math.cos
import kotlin.math.tan
import kotlin.math.atan
import kotlin.math.pow
import kotlin.math.PI
import kotlin.math.sqrt

// Set global parameters and variables.
val baseWidth = 6.5  // m
val xMaxJet = doubleArrayOf(2.25, 2.55, 2.85, 3.0)
val yMaxJet = doubleArrayOf(1.5, 3.0, 4.25, 5.5)
val angle = DoubleArray(4)
val vel = DoubleArray(4)
val g = 9.8  // m/s2

// Set canvas properties.
val xMin = -0.5
val xMax = baseWidth + 0.5
val yMin = -0.5
val yMax = 6.0
val xRange = xMax - xMin
val yRange = yMax - yMin
val canvasW = 700.0
val canvasH = (canvasW/ xRange) * yRange
```

The first half of the import block imports the necessary graphics features from the JavaFX library, and the second half imports the required math functions from the `kotlin.math` library.

Next, we define the global parameters and variables. The base width of the fountain system has been set to 6.5 m. The coordinates for the highest points of the parabolic jets (`xMaxJet` and `yMaxJet`) are set by using type `DoubleArray`. We also create arrays to hold the angles and velocities that we'll be calculating. As we're imagining this fountain to be on Earth, the acceleration due to gravity, `g`, has been set to 9.8 m/s^2. Finally, we set the required dimensions for the canvas based on the width and height of the fountain, with some margins on the sides.

The remainder of the project code has three core segments: the `ShapeOfWater` application class and two helper functions, `getAngleAndVel()` and `getTrajectories()`. We'll look at these in turn.

The ShapeOfWater Application Class

The `ShapeOfWater` application class manages the canvas and calls the helper functions.

```
// the primary application class
class ShapeOfWater : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val canvas = Canvas(canvasW, canvasH)
        val gc = canvas.graphicsContext2D
        gc.translate(0.0, canvas.height)
        gc.scale(1.0, -1.0)
        root.children.add(canvas)
        val scene = Scene(root, canvasW, canvasH)
        scene.fill = Color.WHITE
        stage.title = "Shape of Water"
        stage.scene = scene
        stage.show()

        // problem-specific section
        getAngleAndVel()
        getTrajectories(gc)
        println("\nNozzle velocities:")
        for (v in vel) print(String.format("%.2f  ", v))
        println("\nNozzle angles:")
        for (theta in angle)
             print(String.format("%.2f  ", theta))
    }
}

fun main() {
    Application.launch(ShapeOfWater::class.java)
}
```

The class follows the standard JavaFX block we developed in Chapter 3, which we'll use throughout this book. Notice the two function calls. The first call to `getAngleAndVel()` calculates the θ and v for the four sets of coordinates provided for the four levels of the fountain. Calling `getTrajectories()` calculates the corresponding trajectories for the water jets and draws them on the canvas.

The class declaration ends with a few lines of print statements to display the calculated values of θ and v (as `angle` and `vel`) for the four pairs of (x_{max}, y_{max}) provided. The values are rounded to two decimal places for ease of reading.

The getAngleAndVel() Function

The `getAngleAndVel()` function uses Equations 5.11 and 5.12 to calculate θ and v for the four water jets.

```
fun getAngleAndVel() {
    var index = 0

  ❶ xMaxJet.zip(yMaxJet) { x, y ->
        val theta = atan(2 * y / x)
     ❷ angle[index] = theta * 180/ PI
       vel[index] = sqrt(2 * g * y) / sin(theta)
```

```
        index += 1
    }
}
```

We use `zip()` with a lambda expression ❶ to work through the pairs of (x_{max}, y_{max}) values, calculating `theta` and `vel` for each and storing those values in the appropriate arrays. We could have also used a `for` loop to iterate over the coordinates, and you might try that as an exercise. The `atan()` function gives us angles in radians, but we convert the values to degrees before storing them in the `angles` array to make the output more intuitive ❷.

The getTrajectories() Function

The `getTrajectories()` function calculates the trajectories of the jets based on the data from `getAngleAndVel()` and visualizes the results.

```
fun getTrajectories(gc: GraphicsContext) {
    // Generate trajectories by iterating over time.
    vel.zip(angle) {v, _theta ->
        val theta = _theta * PI / 180
      ❶ val tmax = 1.1 * v * sin(theta)/g
        val delt = tmax/50

        // Calculate trajectory coordinates.
        var t = 0.0
      ❷ while (t <= tmax) {
            val x = v * cos(theta) * t
            val y = x * tan(theta) -
                (g / (2 * (v * cos(theta)).pow(2))) * x.pow(2)

            // Draw points on canvas.
          ❸ gc.fillOval(canvasW * (x - xMin)/ xRange,
                        canvasH * (y - yMin)/ yRange, 3.0, 3.0)
          ❹ gc.fillOval(
                canvasW * ((baseWidth - x) - xMin)/ xRange,
                canvasH * (y - yMin)/ yRange, 3.0, 3.0)
            t += delt
        }
    }
}
```

First, we calculate t_{max} for each (x_{max}, y_{max}) pair by using Equation 5.8. We deliberately extend t_{max} by 10 percent so that when the trajectory is drawn up to that time, we'll see the jet starting to bend downward toward the water basin ❶. Next, we use a `while` loop ❷ to iterate over 50 even time steps between 0 and t_{max}. Using `while` allows us to work with a real-valued (noninteger) increment per time step. We couldn't do that with a `for` loop, whose iterator can only be an integer.

Inside the `while` loop, we use the original displacement Equations 5.1 and 5.2 introduced in Project 17 to calculate the successive locations of a water particle at each time step, giving us the entire trajectory for the water

jet from the nozzle to the basin. Instead of saving the values in an array for future use, we use them instantly to draw small circles on the canvas ❸. By repeating the process for all (θ, v) pairs and for all time steps, we create the two-dimensional profile of the water jets. Notice that we take advantage of the symmetric nature of the problem to draw a second set of jets from another nozzle on the opposite side of the ground-level basin ❹. For this extra set of jets, we only need to adjust the x-coordinates by subtracting `x` from `baseWidth`.

The Result

If you run the downloadable version of the code, the complete 2D solution should resemble Figure 5-6. This version of the code includes two additional functions: `drawFountain()` and `drawNozzles()`. The former is used to draw the four-level fountain from the given arrays of x_{max} and y_{max} for the water jets, while the latter is used to display the positions of the nozzles as shown in Figure 5-5. I encourage you to study these two functions and understand the logic used to calculate the dimensions of the fountain and the positions of the nozzles.

To change the shape of the trajectories, you can adjust the values of x_{max} and y_{max} for the water jets. The shape of the fountain will be automatically adjusted. However, it's important to ensure that the values of x_{max} and y_{max} are consistent and reasonable. For example, the x_{max} and y_{max} for level 2 should be greater than those of level 1, and similar conditions apply to levels 3 and 4. Additionally, all values must be nonnegative.

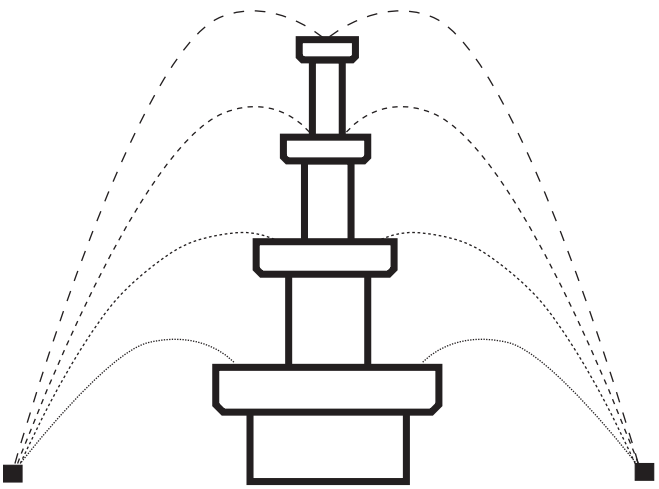


Figure 5-6: Water jet profiles around a multilevel fountain

The values of the nozzle velocities (in m/s) and angles (in degrees) are directly printed onscreen as text output when you run the code. The output should look like this:

```
Nozzle velocities:
6.78  8.33  9.63  10.76
Nozzle angles:
53.13  66.97  71.46  74.74
```

It's not difficult to use this 2D solution to build a real-world 3D fountain. Since we previously imagined the fountain system to be circular in shape, all we need to do is repeat the solution at certain intervals along the perimeter of the circle that passes through the nozzle assemblies. You can also play with the code to generate your own unique fountain designs.

EXERCISE

Using the idea and techniques discussed in Project 18, design a water fountain so that the nozzles are placed along the outer edges of the water tanks of the same four-level structure. Make the nozzles point away from the center of the structure and then draw the water jets. You can decide how far and high those jets should travel before coming down, but be realistic and pay attention to aesthetics. The shape and size of the jets should match the size of the fountain. A water fountain is an object of beauty, after all.

Project 19: Track a Pendulum's Motion and Phase

A *pendulum* is a body or weight hung from a fixed point with the help of a string, chain, or rod. When the pendulum is moved from the stable equilibrium of its vertical position, it can freely swing under the influence of gravity. In ideal conditions, when no friction or air resistance interferes, the pendulum will continue to swing indefinitely once it's set in motion. Clockmakers have long used pendulums to steadily drive mechanical clocks—picture, for example, the swinging pendulum of an antique grandfather clock. Other examples of real-life pendulums include swings, wrecking balls, and church bells.

Pendulums exhibit *simple harmonic motion (SHM)*, defined as the motion of an object that is *periodic*, or repeating, and that is driven by two opposing forces: an *inertial force* that moves the object away from its equilibrium position and a *restoring force* that tries to return the object to its equilibrium position. The maximum displacement of the object relative to the mean or equilibrium position is called the *amplitude*, and the number of cycles the object goes through per unit time is called the *frequency*. The inverse of frequency, which is the time it takes to complete one cycle or oscillation, is called the *period*.

The simple harmonic motion of a pendulum provides a starting point for understanding the more complex periodic motions found in nature,

from the vibration of an electron to the orbit of a planet about a star. This project introduces the equation for the motion of a simple pendulum. It shows you how to use a numerical method to solve this equation and calculate the state of the pendulum—that is, its velocity and displacement angle—at any given time once the pendulum is set in motion. For added realism, we'll factor in the effect of air resistance to show how it gradually dampens the pendulum's motion. Of course, we'll codify our model pendulum in Kotlin and visualize the results by using the JavaFX tools we learned in Chapter 3. Our visualization will consist of three charts: one showing the change of the pendulum's velocity over time, one showing the change of its displacement angle over time, and one showing the relationship between the two.

The Motion of a Simple Pendulum

A simple pendulum has a single weight, or *bob*, of mass m that hangs from a thin string of length l fixed at point A. It's assumed that the mass of the string is negligible compared with the mass of the bob. It's also assumed that the string doesn't stretch due to the weight of the bob and that the bob can move back and forth indefinitely without any friction or loss of energy (meaning the total energy of the system is conserved, once set in motion). As the pendulum swings, its angular displacement (θ) changes, as measured relative to the pendulum's stable equilibrium position (the vertical line passing through point A). Figure 5-7 shows these components of a simple pendulum.

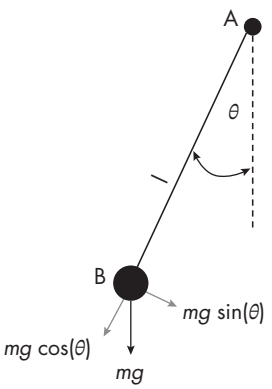


Figure 5-7: Components of a simple pendulum

At any point in time, the force of gravity, mg, acts on the bob to set it in motion. The gravitational force has two components. The first, $mg\cos(\theta)$, acts along the line of the string and is countered by the tension in the string. The second, $mg\sin(\theta)$, acts perpendicularly to the string. This is the part that isn't balanced by any other force (assuming no air resistance) and is responsible for the harmonic motion of the pendulum.

Let ω denote the rate of change of angular displacement, also called the *angular velocity*:

$$\omega = \frac{d\theta}{dt} \tag{5.13}$$

The right-hand side of Equation 5.13 is called the derivative of θ with respect to time t. Mathematically speaking, the time derivative of a variable gives us the instantaneous rate of change of that variable with respect to time.

We can now express the equation of motion for this simple pendulum in terms of its angular acceleration as follows:

$$\frac{d\omega}{dt} = -\frac{g}{l}\sin\theta \tag{5.14}$$

An analytical solution to Equation 5.14 would give us an expression for the angular displacement θ in terms of time t and other parameters. However, the analytical solution isn't easy to derive unless we make an additional assumption that the maximum displacement θ_{max} is relatively small—say, less than 15 degrees. This would allow us to replace $\sin(\theta)$ with θ, since the two are equivalent for small angles.

We don't want to be limited to just this special case, however. Instead, we'll use the Euler-Cromer numerical method to incrementally solve Equation 5.14 for a small time step. This method maintains the conservation of energy inherent in simple harmonic motion, and it yields *stable* solutions (a solution is *stable* when the numerical approximation doesn't deteriorate and move away from the theoretical solution over time).

The Strategy

To apply the Euler-Cromer method, we need to change Equation 5.14 from its continuous-time form to its discrete-time form:

$$\omega_{t+1} = \omega_t - \frac{g}{l}\sin\theta_t \delta t \tag{5.15}$$

Here δt is a small, discrete time step. Next, we'll incorporate the impact of air resistance on the pendulum. Air resistance slows down the angular velocity as a decelerating force, which can be assumed to be proportional to the angular velocity. Assuming γ to be the proportionality constant, we can modify Equation 5.15 as:

$$\omega_{t+1} = \omega_t - \frac{g}{l}\sin\theta_t \delta t - \gamma\omega_t \delta t \tag{5.16}$$

Finally, we'll write another expression for updating the angular displacement θ, which follows directly from the definition of angular velocity ω in Equation 5.13 as:

$$\theta_{t+1} = \theta_t + \omega(t)\delta t \tag{5.17}$$

Here, we are assuming that over a small time interval δt, θ is changing from θ_t to θ_{t+1}. In reality, ω also continues to change over this small time period. We recognize this fact by denoting ω as a function of time—that is, $\omega(t)$—in Equation 5.17. However, in discrete-time representation, we can use only one discrete value of ω, and the Euler-Cromer method requires that it be ω_{t+1} to ensure numerical stability. Thus, the final numerical form for the equation of motion of a simple pendulum is:

$$\theta_{t+1} = \theta_t + \omega_{t+1}\,\delta t \tag{5.18}$$

By starting with initial values for θ and ω, and by repeatedly using Equations 5.16 and 5.18, we can calculate the state of a simple pendulum and plot the changes of its angular displacement and velocity over time.

The Code

The program for modeling our simple pendulum has three main code segments: the main application segment coordinating the model, a `simplePendulumWithDrag()` function implementing the equations we've discussed, and a `singleXYChart()` function to draw each plot. We'll review these segments in detail in the same order, starting with the main application segment.

```
// import block
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.control.ScrollPane
import javafx.scene.layout.VBox
import javafx.scene.paint.Color
import javafx.scene.chart.*
import javafx.stage.Stage
import java.text.DecimalFormat
import kotlin.math.PI
import kotlin.math.sin
import kotlin.math.sqrt

// data classes
data class XYChartData(val x: Double, val y: Double)
data class PendulumData(val theta: Double, val omega: Double,
                        val t: Double)

// problem definition and global parameters
val theta0 = -PI/6  // angular displacement at t = 0, rad
val omega0 = 0.0    // angular velocity at t = 0, rad/s
val l = 0.4         // length, m
val g = 9.81        // acceleration due to gravity, m/s2
val n = 100         // intervals
val gamma = 0.6     // drag coefficient

class SimplePendulum : Application() {
    override fun start(primaryStage: Stage) {
        val root = VBox()
        val scroll = ScrollPane()
```

```
        scroll.content = root
        val scene = Scene(scroll, 550.0, 600.0, Color.WHITE)
        primaryStage.title = "Simple Pendulum"
        primaryStage.scene = scene
        primaryStage.show()

        // Generate pendulum state data.
    ❶ val state: List<PendulumData> =
            simplePendulumWithDrag(theta0, omega0,
                  l, g, n, gamma)

        // Create (x, y) series for plotting.
        val list1 = mutableListOf<XYChartData>()
        val list2 = mutableListOf<XYChartData>()
        val list3 = mutableListOf<XYChartData>()

    ❷ for (item in state) {
            val (theta, omega, t) = item
            list1 += XYChartData(t, theta)  // t along x-axis
            list2 += XYChartData(t, omega)  // t along x-axis
            list3 += XYChartData(theta, omega)
        }

        // Call singleXYChart() to generate plots.
        val xyChart1 =
            singleXYChart(list1,
                title = "Angular Displacement over Time",
                xLabel = "Time (sec)",
                yLabel = "Angular displacement (rad)")
        val xyChart2 =
            singleXYChart(list2,
                title = "Angular Velocity over Time",
                xLabel = "Time (sec)",
                yLabel = "Angular velocity (rad/sec)")
        val xyChart3 =
            singleXYChart(list3,
                title = "Phase-Space Plot (omega vs. theta)",
                xLabel = "Angular displacement (rad)",
                yLabel = "Angular velocity (rad/sec)",
                sort = "NONE")

        // Add the charts to the root (VBox) object.
    ❸ root.children.addAll(xyChart1, xyChart2, xyChart3)
    }
}

fun main() {
    Application.launch(SimplePendulum::class.java)
}
```

As usual, the main application segment starts with the boilerplate import block required for a JavaFX-based app. This time, we're also going to use a class called `ScrollPane` that can hold a graphics element and provide a scrollable view of it. This will help us view all three charts in the same

window. The import block also includes the math functions we'll need for this project.

The second code block declares two data classes. We'll use the `XYChartData` class to hold (x, y) data points for creating the line charts and the `PendulumData` class to store the state of the pendulum (angular displacement θ and velocity ω) for a specific time t. Each `XYChartData` data point will be fetched from the values in a particular `PendulumData` instance.

Next, we provide problem-specific and global parameter values. For example, we've set the initial displacement to `-PI/6` (–30 degrees) and the initial velocity to `0`. The length of the pendulum, measured in meters, is `0.4`, and `n = 100` means we'll determine the value of the time step (δt) by dividing the period of oscillation into 100 intervals. We've also set the coefficient for air resistance to `0.6` (a value of `0` would mean no air resistance).

Inside the `SimplePendulum` application class, the problem-specific part of the code starts with a call to `simplePendulumWithDrag()` ❶ that takes in the following arguments: the initial displacement θ_0, the initial velocity ω_0, the length of the pendulum l, the acceleration due to gravity g, the number of time intervals per period n, and the air resistance coefficient γ. As you'll see shortly when we look at the inner workings of this function, it returns a list of `PendulumData` instances, one for each time step, which we store as `state`.

Next, we create three mutable lists to hold the `XYChartData` instances that we'll need to draw the line charts illustrating the pendulum's state variables. We build these lists by extracting the elements from each `PendulumData` instance as needed for the three charts ❷. Charts 1 and 2 both show time on the x-axis and displacement and velocity on the y-axis, respectively; chart 3 shows displacement on the x-axis and velocity on the y-axis.

We generate the three charts by making successive calls to the `singleXYChart()` function. The function takes five arguments, four of which have default values assigned in the function definition. Thus, we only need to provide a list of type `XYChartData` (for example, `list1`) and optionally supply values of the other named parameters. Notice that for `xyChart3`, we specify `sort = "NONE"` so that we can correctly plot the (velocity, displacement) pairs without changing their sequence. This is a requirement when we consider the air resistance that alters the cyclic relationship between the two state variables.

We plot the charts by attaching them to `root` ❸, which we previously assigned to a `ScrollPane`. This scheme allows us to view all three charts inside the same window by scrolling up and down or sideways as needed. Finally, the `main()` function's only job in a JavaFX application is to launch the main application class—in this case, `SimplePendulum`.

The simplePendulumWithDrag() Function

In the `simplePendulumWithDrag()` function, we use the Euler-Cromer method to calculate the values of the pendulum's state variables for a specified number of time steps, given the initial conditions and global parameter values.

```
fun simplePendulumWithDrag(
        theta0 : Double,
        omega0 : Double,
```

```
        l: Double, g: Double, n: Int,
        gamma: Double = 0.0): List<PendulumData> {

    // Set local variables, parameters, and list.
    val alpha = g / l
    // Calculate period for small displacement.
❶   val T = 2 * PI * sqrt(l/g)
❷   val dt = T / n
    val Nmax = 4 * n
    val df = DecimalFormat("##.####")

    var omegaOld: Double; var omegaNew: Double
    var thetaOld: Double; var thetaNew: Double
    var timeOld: Double;  var timeNew: Double
    val pList = mutableListOf<PendulumData>()

    // Initialize for t = 0.
    thetaOld = theta0
    omegaOld = omega0
    timeOld = 0.0
    pList += PendulumData(theta0, omega0, 0.0)

    // Calculate and save state variables.
❸   for (k in 1..Nmax) {
        omegaNew = omegaOld -
            (alpha * sin(thetaOld) + gamma * omegaOld) * dt
        thetaNew = thetaOld + omegaNew * dt // Euler-Cromer
        timeNew = timeOld + dt
      ❹ pList += PendulumData(thetaNew, omegaNew, timeNew)

      ❺ omegaOld = omegaNew
        thetaOld = thetaNew
        timeOld = timeNew
    }

    println("\n*** Simple Pendulum Simulation ***\n")
    println("length l: $l m")
    println("theta0: ${df.format(theta0*180/PI)} degrees")
    println("omega0: ${df.format(omega0*180/PI)} rad/sec")
    println("gamma: ${df.format(gamma)}")
    println("dt: ${df.format(dt)} sec")
    println("Nmax: $Nmax intervals")
    println("Simulation length: ${df.format(Nmax*dt)} sec")

    return pList
}
```

Our first order of business is to determine the value of the time step `dt` (δt). For that, we need to estimate the pendulum's period of oscillation T ❶, using the following formula:

$$T = 2\pi\sqrt{\frac{l}{g}}$$

Technically, this formula is valid only for initial displacement angles under 15 degrees, but it provides a good starting approximation. Dividing T by the number of time intervals per period `n` gives us the length of each time interval ❷.

Notice that we also arbitrarily set the maximum number of time steps, `Nmax`, to `4 * n` intervals to ensure that the simulation period includes at least three full cycles of the pendulum's oscillation (the period of a pendulum gets longer as the initial displacement or amplitude gets larger). Feel free to try out other values for these parameters, but keep in mind that a larger `n` (and thus a smaller `dt`), or a longer simulation period, will increase the number of computations and require more memory to store all the intermediate values of the state variables.

After initializing the local variables, parameters, and the mutable list `pList` that saves the state variables (including time), we implement the Euler-Cromer method by using a `for` loop ❸. Inside this loop, we use Equations 5.16 and 5.18 to calculate `omegaNew` and `thetaNew`, along with the updated time `timeNew`. We store these in an instance of `PendulumData` and add it to `pList` ❹. Then we assign these "new" state variables back to the "old" state variables (for example, setting `omegaOld = omegaNew` ❺) for the next time through the loop. In all, these steps are repeated for each time interval, for a total of `Nmax` times.

The code segment includes several print statements that summarize the global values and parameters used to solve the problem. Then the segment ends by returning `pList` to the calling function.

The singleXYChart() Function

The `singleXYChart()` function creates a line chart with JavaFX. In this case, using the JavaFX charting tools is a more natural choice for creating our visualization than using the canvas feature, as we did in Project 18. This time we want to be able to read off the values of the state variables directly from the charts, and this would not be easy to do from scratch using the canvas (we'd essentially have to write our own charting library).

Most of the code in this segment is boilerplate JavaFX code used for creating a line chart. Since this process was discussed in detail in Chapter 3, we'll focus only on the problem-specific elements in this code.

```
fun singleXYChart(data: List<XYChartData>,
          title: String  = "",
          xLabel: String = "x-axis",
          yLabel: String = "y-axis",
          sort: String = "default"): LineChart<Number, Number> {

    // Define axes.
    val xAxis = NumberAxis()
    val yAxis = NumberAxis()
    xAxis.label = xLabel
    yAxis.label = yLabel
```

```
    // Create LineChart.
    val lineChart = LineChart(xAxis, yAxis)
    lineChart.title = title
  ❶ lineChart.createSymbols = false
  ❷ lineChart.isLegendVisible = false
  ❸ if (sort == "NONE")
    lineChart.axisSortingPolicy = LineChart.SortingPolicy.NONE

    // Define series.
    val series = XYChart.Series<Number, Number>()

    // Populate series with data.
  ❹ for (item in data) {
        val (x, y) = item
        series.data.add(XYChart.Data(x, y))
    }

    // Assign series with data to LineChart.
    lineChart.data.add(series)

    // Return LineChart object.
    return lineChart
}
```

The first thing to notice is that the function takes five arguments, four of which are named arguments. We've provided default values for the named arguments in the function definition, and these defaults will be used unless we supply problem-specific values when calling the function.

To keep the chart clean and simple, we turn off markers ❶ and the legend ❷. We also turn off the default sorting of x-values when the `sort` parameter is set to `"NONE"` ❸. As you saw in the main application code, we use this option when creating the chart of angular velocity over displacement, because we need to maintain the order of (ω, θ) pairs to correctly capture the cyclic nature of this relationship. Finally, we extract the data points for the line chart from `data` (which is a list of `XYChartData` instances) by deconstructing each `item` in `data` into a set of `x` and `y` values ❹.

The Result

The program generates a few lines of text output and a scrollable window that contains our line charts. The text output should look like this:

```
*** Simple Pendulum Simulation ***

length l: 0.4 m
theta0: -30 degrees
omega0: 0 rad/sec
gamma: 0.6
dt: 0.0127 sec
Nmax: 400 intervals
Simulation length: 5.075 sec
```

This output provides a reminder of the parameter values used and especially shows the time interval δt that we've calculated internally based on the estimated period of the pendulum. You need to monitor this value if accuracy is a concern.

Regarding the content of the chart window, the first line chart shows how the angular displacement varies with time, and the second chart shows the same for angular velocity (see Figure 5-8).

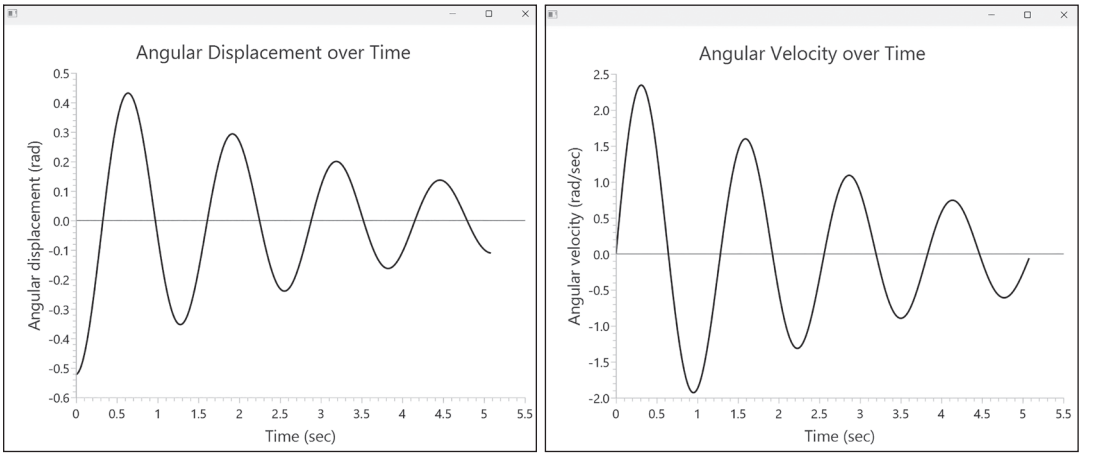


Figure 5-8: The angular displacement (left) and angular velocity (right) of a pendulum facing air resistance

The charts show the pendulum's periodic pattern decaying, or dampening, due to air resistance; otherwise, the maximum displacement and the maximum velocity would remain unchanged. Notice the phase difference between θ (displacement) and ω (velocity): ω lags θ by a fixed distance along the time axis. Specifically, whenever θ is at its most extreme, ω is 0, and vice versa. For example, at $t = 0$, θ is at its maximum displacement, $-\pi/6$ (to the left of the equilibrium position), but $\omega = 0$; then, at approximately $t = 0.32$, the pendulum passes through its equilibrium position ($\theta = 0$) and ω reaches its maximum magnitude. This is because at this location, all of the pendulum's potential energy becomes kinetic energy. Beyond that point, ω begins to drop as the kinetic energy converts back into potential energy, until ω becomes 0 again at the point of maximum displacement on the opposite side.

This relationship can be shown directly by plotting ω against θ, which is exactly what we've done in our third plot. This kind of visualization is known as a *phase-space plot*; for a dynamic system in two dimensions, it plots velocity $v(t)$ over displacement $x(t)$, the common denominator being time, t. Also called a *phase portrait*, a phase-space plot helps in studying complex system behaviors and uncovering relationships between the state variables that might otherwise remain undetected. Figure 5-9 shows our phase-space plot for the pendulum, with θ on the x-axis and ω on the y-axis.

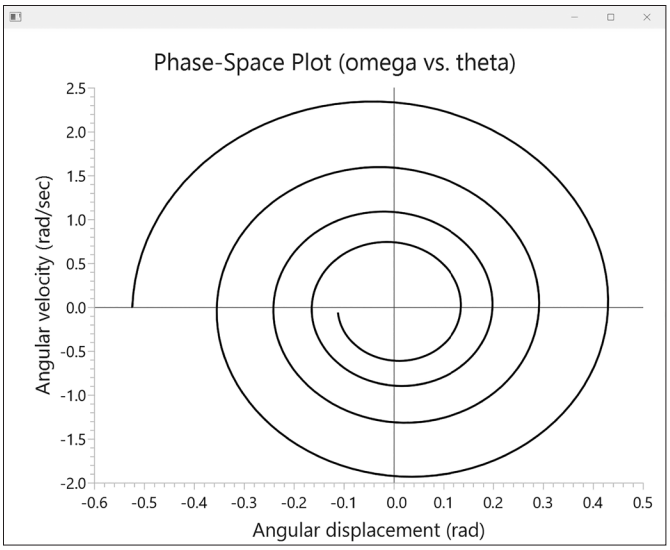


Figure 5-9: A phase-space plot of a simple pendulum with damped motion

The main point to notice in the plot is that while ω and θ are still related through the parameter t, both are decreasing due to air resistance. With each time step, the pendulum gets a bit slower and its swing a bit narrower—hence the distinctive spiral shape of the plot. If you're wondering what the pattern might look like for an undamped pendulum that isn't experiencing any air resistance, you can simply set the drag coefficient γ to 0 and run the program again to investigate.

EXERCISE

For many periodic systems, the Euler-Cromer method, which is *first-order convergent* (the global error is proportional to the step size), isn't suitable, because it will require too many computations to produce results of acceptable accuracy. The Runge-Kutta (R-K) method offers significant computational efficiency and accuracy over the Euler-Cromer method. Study the R-K method and implement the algorithm in Kotlin to solve the simple pendulum problem. (Detailed instructions can be found in any introductory resource on numerical analysis, including those provided at the end of this chapter.) You'll notice that the R-K method will allow you to use larger time steps to achieve the same level of accuracy as the Euler-Cromer method.

Project 20: The Physics of Coffee Cooling

Now we'll turn our attention from the physics of motion to the physics of heat transfer. Imagine you've picked up a cup of coffee from your favorite coffee shop on your way to work in the morning. It's a 25-minute drive from the coffee shop to your office, during which the coffee will get colder due to heat loss. You like to have milk in your coffee, but the milk is kept refrigerated, so adding it will further lower the temperature of the coffee. You could add milk right away at the coffee shop or wait to add milk from your office refrigerator. Which option should you use to keep your coffee as warm as possible?

We'll write a Kotlin application to plot the coffee's change in temperature over time in both scenarios. For that, we'll first need to review Newton's law of cooling, which defines how an object cools due to the loss of heat energy to its surroundings. We also have to consider the science of mixing two different liquids so we can figure out how to calculate the change of temperature when we add cold milk to hot coffee.

Newton's Law of Cooling

Newton's law of cooling states that the rate of change of the temperature of an object is proportional to the difference between the object's temperature and the ambient temperature. It can be mathematically expressed as follows:

$$\frac{dT}{dt} = -k(T - T_a) \tag{5.19}$$

T is the temperature of the object at time t, T_a is the ambient temperature, which is assumed to remain constant, and k is the heat transfer coefficient. For situations where T is greater than T_a, the greater the value of k, the quicker the object will cool.

The value of the coefficient k depends on the material and surface properties of the object, as well as on the temperature of the object itself. When the primary method of heat loss is conduction, and convective and radiative losses are negligible, k can be assumed to remain constant for a small range of temperature variations. Our coffee cooling problem meets these requirements well: most of the heat loss happens conductively through the wall of the cup, convection is minimized by keeping a lid on the cup, and electromagnetic radiation in this temperature range is negligible compared to the other two factors.

Let's now look at the solution of Equation 5.19, which is a first-order ordinary differential equation. For the initial temperature T_0 (at $t = 0$) > T_a, the solution can be expressed as:

$$T(t) = T_a + (T_0 - T_a)\ e^{-kt} \tag{5.20}$$

Equation 5.20 will allow us to calculate the temperature of the coffee (with or without milk) given the initial coffee temperature T_0, the ambient temperature T_a, the elapsed time t, and the heat transfer coefficient k.

The Effect of Mixing Liquids

When two liquids of different temperatures are mixed together, the hotter liquid gets cooler by releasing heat energy, and the colder liquid gets warmer by absorbing the released heat. This exchange of heat energy will continue until both liquids attain the same temperature.

Mathematically, the amount of heat Q absorbed or released by one of the liquids, and its corresponding change in temperature ΔT, are related as follows:

$$Q = C\Delta T, \text{ where } C = \rho V s \tag{5.21}$$

The term C in Equation 5.21 is called the *thermal mass* of the object, which is the product of its density ρ, volume V, and specific heat capacity s. In SI units, Q is measured in joules (J), temperature T in degrees Celsius (°C), ρ in kg/m^3, V in m^3, and s in J/(kg °C). As a result, the unit for C will be J/°C. For the coffee cooling problem, since we'll be working with small quantities of liquids, we'll use grams per milliliter (g/mL) for density, mL for volume, and J/(g °C) for specific heat capacity.

Say we're mixing two liquids such that C_1, T_1, and T_f are the thermal mass, initial temperature, and final temperature for liquid 1, and C_2, T_2, and T_f are the same for liquid 2. (Notice that the final temperature for both liquids is the same.) If the mixing happens quickly, with no heat exchange with the surrounding environment, and if the liquids don't chemically react and generate or absorb heat, the principle of conservation of energy suggests that the net change in energy in the system will be zero—that is, $Q_1 + Q_2 = 0$. After substituting for Q_1 and Q_2 using Equation 5.21 and simplifying the results, we get the following expression for the final temperature:

$$T_f = \frac{C_1 T_1 + C_2 T_2}{C_1 + C_2} \tag{5.22}$$

You can use Equation 5.22 to calculate the temperature of the coffee after adding milk, either at the coffee shop or when you arrive at the office. Finally, we have all the science worked out to solve our problem and enjoy the optimal cup of coffee.

The Strategy

To identify the right time for adding milk, we need some data on the properties of the coffee and milk, as well as the ambient temperature of the environment. This data is summarized in Table 5-2.

Table 5-2: Coffee Cooling Problem Data

Item	Value and unit
Initial temperature of black coffee	92°C
Volume of black coffee	250 mL
Density of black coffee	1 g/mL
Specific heat capacity of coffee	4.19 J/(g °C)
Initial temperature of refrigerated milk	4°C
Volume of milk added to the coffee	25 mL
Density of milk	1.035 g/mL
Specific heat capacity of milk	3.89 J/(g °C)
Length of the drive from the coffee shop to the office	25 minutes
Heat transfer coefficient for coffee (assumed to be the same with or without milk)	0.0116/minute
Ambient temperature everywhere	20°C

Armed with the science of coffee cooling and the data presented in Table 5-2, we can now solve the problem by following these steps:

Option 1: Add milk later

1. Calculate the temperature of black coffee after 25 minutes, starting with the initial temperature of 92°C.
2. Calculate the final temperature of the coffee after adding 25 mL of milk (assuming that mixing and heat exchange happen instantly).

Option 2: Add milk at the coffee shop

1. Calculate the initial temperature of the coffee when milk is added immediately at the coffee shop (assuming that mixing and heat exchange happen instantly).
2. Calculate the final temperature of the coffee after 25 minutes.

By comparing the final temperatures from options 1 and 2, we'll know which option will keep the coffee warmer.

The Code

We'll develop this program as a JavaFX application so we can plot the coffee's temperature profile over time for both options. For this, we'll use the same template as in Project 19, with some changes to the problem-specific parts.

Let's start with the problem definition and global parameters, along with the problem-specific components in the main application class, `MixCoffeeAndMilk`.

```
// import block
import javafx.application.Application
import javafx.scene.Scene
```

```
import javafx.scene.control.ScrollPane
import javafx.scene.layout.VBox
import javafx.scene.paint.Color
import javafx.scene.chart.*
import javafx.stage.Stage
import java.text.DecimalFormat
import kotlin.math.exp
import kotlin.math.ln
import java.text.DecimalFormat

// data classes
❶ data class State(
    val time: Double,
    val Temp: Double
)

// problem definition and global parameters
val coffeeT0 = 92.0      // degrees Celsius
val coffeeV = 250.0      // mL
val coffeeS = 4.190      // J/(gm C) - assumed same as water
val coffeeD = 1.0        // gm/mL - assumed same as water
val coffeeK = 0.0116     // 1/min

val milkT0 = 4.0         // degrees Celsius
val milkV = 25.0         // mL
val milkS = 3.890        // J/(gm C)
val milkD = 1.035        // gm/mL

val T_ambient = 20.0     // degrees Celsius
val timeMax = 25.0       // min (length of drive)
❷ val timeStep = 0.25    // min
val df = DecimalFormat("#.##")

// application class
class MixCoffeeAndMilk : Application() {
    override fun start(primaryStage: Stage) {
        val root = VBox()
        val scroll = ScrollPane()
        scroll.content = root
        val scene = Scene(scroll, 550.0, 600.0, Color.WHITE)
        primaryStage.title = "Coffee Cooling Profile"
        primaryStage.scene = scene
        primaryStage.show()

        // Execute steps for coffee cooling process.
        println("\n  *** Coffee Cooling Problem ***  \n")

        // step 1:
      ❸ val state1 =
            newtonCooling(T0 = coffeeT0, Ta = T_ambient,
                          k = coffeeK,tMax = timeMax, dt = timeStep)
        printTimeAndTemp(state1.last(), 1)
```

```
        // step 2:
    ❹ val finalT1 =
          tempAfterMixing(d1 = coffeeD, v1 = coffeeV, s1 = coffeeS,
                          T1 = state1.last().Temp,
                          d2 = milkD, v2 = milkV,
                          s2 = milkS, T2 = milkT0)
        println("step 2: final temp with milk: " +
                "${df.format(finalT1)} degrees Celsius\n")

        // step 3:
    ❺ val initT2 =
          tempAfterMixing(d1 = coffeeD, v1 = coffeeV, s1 = coffeeS,
                          T1 = coffeeT0, d2 = milkD, v2 = milkV,
                          s2 = milkS, T2 = milkT0)
        println("step 3: initial temp with milk: " +
                "${df.format(initT2)} degrees Celsius")

        // step 4:
    ❻ val state2 =
          newtonCooling(T0 = initT2, Ta = T_ambient, k = coffeeK,
                        tMax = timeMax, dt = timeStep)
        printTimeAndTemp(state2.last(), 4)

        // step 5:
    ❼ val state3 =
          newtonCooling(T0 = finalT1, Ta = T_ambient,
                        k = coffeeK,tMax = timeMax, dt = timeStep,
                        start = timeMax)

    ❽ val state4 =
          newtonCooling(T0 = state2.last().Temp,
                        Ta = T_ambient, k = coffeeK,
                        tMax = timeMax, dt = timeStep, start = timeMax)

        val states =
            listOf(state1, state2, state3, state4)

        createCoolingChart(root, states = states)
    }
}

fun main() {
    Application.launch(MixCoffeeAndMilk::class.java)
}
```

The import block for this code segment is similar to that of Project 19, except that we need fewer math functions. In addition to initializing the values from Table 5-2, we declare a data class called `State` ❶ to save the temperature at a given time. This will help us organize the data to be plotted. Among the parameters, notice that we set `timeStep` to `0.25` minutes ❷. This way, we'll track and plot the coffee's cooling progress in 15-second intervals. This time step may seem surprisingly long compared to the very short time intervals from Project 19, but it's adequate for this problem because the

temperature drops smoothly and slowly. Besides, we're using an analytical solution and not a numerical approximation; thus, this choice doesn't affect the accuracy of the calculations.

The problem-specific part of the code in the main application class follows the same order as the steps outlined in "The Strategy" on page 211, except that we've added an extra step (step 5) to further track the coffee as it continues to cool at the office (more on that shortly). The code here relies on two main helper functions: the `newtonCooling()` function, which calculates how the coffee cools over time, and the `tempAfterMixing()` function, which calculates the immediate temperature after adding the milk. We deploy these functions as follows:

- For step 1, we call `newtonCooling()` and save the result as `state1` ❸. This produces a list of type `State` containing the data points needed to show how the black coffee gets colder over time between the coffee shop and the office.
- For step 2, we call `tempAfterMixing()` to get the final temperature of the coffee when milk is added after arriving at the office ❹. This is how warm the coffee will be before you take the first sip if you chose option 1 (add milk later).
- For step 3, we call `tempAfterMixing()` to calculate the temperature of the coffee after adding milk at the coffee shop, before the drive to work ❺.
- For step 4, we call `newtonCooling()` and save the resulting list of `State` data points as `state2` ❻. The temperature property of the last element of this list will tell us how warm the coffee will be when you arrive at the office after choosing option 2 (add milk first).

During each step, we display some output from the simulation either by using the `println()` function or by calling the custom `printTimeAndTemp()` function, defined here:

```
fun printTimeAndTemp(datapoint: State, step: Int) {
    val (endTime, endTemp) = datapoint

    println("step $step: end time: ${df.format(endTime)} minutes")
    println("step $step: end temp: ${df.format(endTemp)} " +
            "degrees Celsius")
}
```

We have everything we need to solve the problem after step 4, and the results are printed on the console. Let's have a look at the results first, then come back to step 5 and the code for visualizing the cooling process.

```
  *** Coffee Cooling Problem ***

step 1: end time: 25 minutes
step 1: end temp: 73.87 degrees Celsius
step 2: final temp with milk: 67.75 degrees Celsius
```

```
step 3: initial temp with milk: 84.29 degrees Celsius
step 4: end time: 25 minutes
step 4: end temp: 68.1 degrees Celsius
```

The final temperature of the coffee is 67.75°C when you add milk at the office versus 68.1°C when you add milk at the coffee shop. Though the difference isn't that significant (just 0.35°C), you'll be better off adding the milk at the coffee shop.

Of course, you could figure this out for yourself without any math or code by investing in a good-quality thermometer and brewing two cups of coffee. The benefit of building a mathematical model of the process, however, is that it not only tells us what the temperature will be after 25 minutes but also how the system will get there, starting from an initial state and considering interventions such as adding milk. Once we have the model, we can play with the parameter values and generate answers to many other questions, without making more coffee, adding more milk, and taking many temperature measurements.

For example, say you get pulled into a meeting right after you arrive at work. You have just enough time to add milk to your coffee (if you haven't added it already), but you don't have time to enjoy said coffee until the meeting ends—another 25 minutes later. What will the temperature of the coffee be at that point, a full 50 minutes after you bought it? Since we already have a mathematical model for how the coffee cools over time, this question is quite easy to answer.

This brings us to step 5, where we use the final temperatures after the first 25 minutes as the initial temperatures for running the simulation for a further 25 minutes. For this, we use an additional named parameter `start` in the `newtonCooling()` function. This parameter lets us offset the time values by `25` instead of starting the simulation time over at `0`. We call the function twice, generating lists `state3` ❼ (for option 1—milk added at the office) and `state4` ❽ (for option 2—milk added at the coffee shop).

Once all four states are calculated, we create a list of type `State` by using the `listOf()` method and pass this list to the `createCoolingChart()` function for plotting the temperature profiles for the entire 50-minute period.

Calculating the Temperature Changes

Now let's review our two helper functions for calculating the temperature changes in the coffee. The `newtonCooling()` function tracks the temperature change of the coffee over a period of time. The `tempAfterMixing()` function calculates the instantaneous temperature change when the milk is mixed in.

```
fun newtonCooling(T0: Double, Ta: Double, k:Double,
                  tMax: Double, dt: Double,
                  start: Double = 0.0): List<State> {
    val state = mutableListOf<State>()
    var t = 0.0

    while (t <= tMax) {
      ❶ val temp = Ta + (T0 - Ta)*exp(-k * t)
      ❷ state += State(t+start, temp)
```

```
        t += dt
    }
    return state
}

fun tempAfterMixing(
    d1: Double, v1: Double, s1: Double, T1: Double,
    d2: Double, v2: Double, s2: Double, T2: Double
    ): Double {

    return (d1 * v1 * s1 * T1 + d2 * v2 * s2 * T2) /
           (d1 * v1 * s1 + d2 * v2 * s2)
}
```

In the `newtonCooling()` function, we use Equation 5.20 to calculate the temperature of the coffee at point `t` in time ❶. We do this in a `while` loop, incrementing `t` by `dt` (set to `timeStep`) each iteration, until we get to `tMax`, giving us time and temperature data points in 15-second intervals, up to 25 minutes, which we store in a list called `state` ❷.

Notice that we add `start` to each `t` before storing it in the list. When `start` isn't set during the function call, it defaults to `0` and has no effect. However, when we call the function to create `state3` and `state4` (as part of step 5, discussed earlier), we set `start = timeMax` so the timestamps will range from 25 to 50. This allows us to plot all the data in a single chart.

The `tempAfterMixing()` function takes in the separate density, volume, specific heat capacity, and premixing temperatures for the milk and coffee and returns the final equilibrium temperature once the milk is mixed in, using Equation 5.22.

Plotting the Temperature Profiles

We're now ready to plot the two temperature profiles of the cooling coffee. For that, we'll define the `createCoolingChart()` function.

```
fun createCoolingChart(root: VBox, states: List<List<State>>) {

    val xyChart =
        singleXYChart(states,
            title = "Temperature of Coffee over Time",
            xLabel = "Time",
            yLabel = "Temperature (degrees Celsius)")

   root.children.add(xyChart)
}
```

We use this short function mainly to preprocess the information needed by the `singleXYChart()` function, also used in Project 19. This helps keep the body of the main application class less cluttered. Since we've discussed how to use `singleXYChart()` previously, I'll skip that part here, except to mention that we're now passing a list of lists rather than one single list as a collection of data points. Inside the `singleXYChart()` function, we therefore

create four different series (from the list of lists, `states`) and plot them on the same chart instead of creating four separate charts. See Chapter 3 for a review of how to plot a single series versus multiple series in the same chart.

The last line of this function adds the chart object `xyChart` returned by the `singleXYChart()` function to the `root` node for display. Figure 5-10 shows the result.

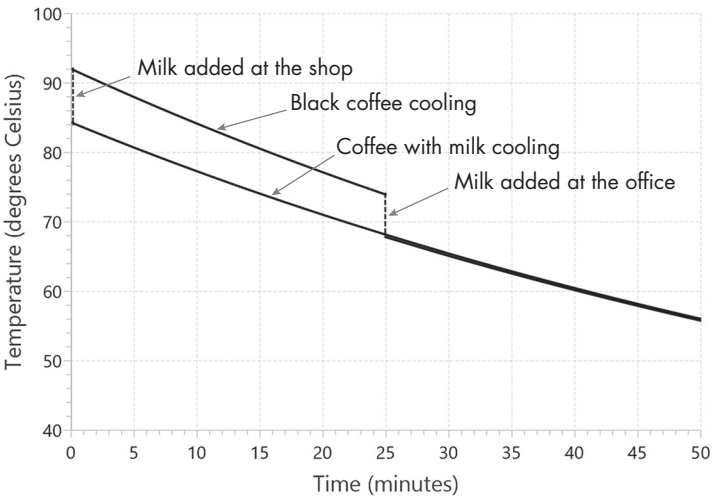


Figure 5-10: The temperatures of black coffee and coffee with milk over time

The plot spans the entire 50-minute period of the simulation. We can see the sudden temperature drops when the milk is added either at the coffee shop (at time 0) or at the office (at time 25). After the latter, the temperature profiles appear to be almost identical. When we zoom in, however, as in Figure 5-11, we find a slight difference (less than 1°C).

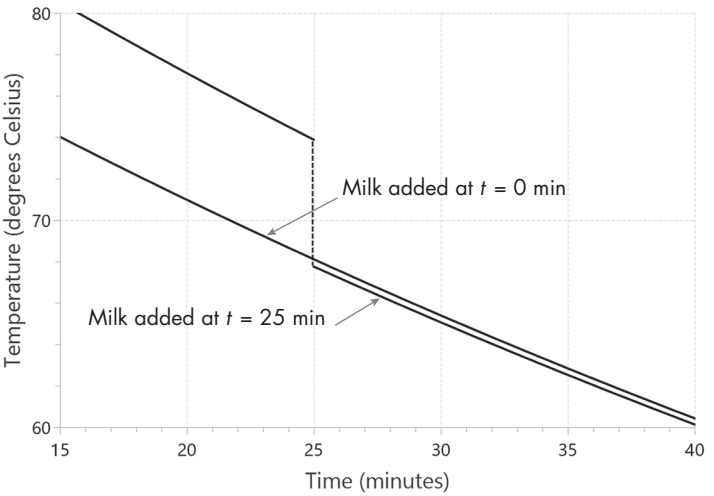


Figure 5-11: A closer look at the coffee cooling problem for t > 25 minutes

The gap between the two lines indicates that the coffee with milk added at the shop will always be a tad warmer. If we continue this experiment for a long time, will these temperature profiles still stay separate from each other? Why or why not?

EXERCISE

In this exercise, you'll apply Newton's law of cooling to help with a forensic investigation into the death of a person whose body was found in a locked apartment. This is what you know so far: The body was found at 2:00 PM, when the core body temperature was recorded as 25°C. A second body temperature reading two hours later (in the same room) yielded 23.6°C. The temperature inside the apartment was set to a constant 20°C. Assuming the person died in the same room where the body was found, estimate the approximate time of death.

Hint: Start by writing a new function `getHTC()` to estimate the heat transfer coefficient k based on the two recorded body temperatures postmortem. Use the following equation:

$$k = \frac{1}{\Delta t} \ln\left(\frac{T_1 - T_a}{T_2 - T_a}\right)$$

Here Δt is the elapsed time between the two temperature readings, T_1 is the first reading, T_2 is the second reading, and T_a is the ambient temperature. Next, write a second function using the same equation, but solve it for Δt when k is given. This time, T_1 is the average body temperature of a live human (37°C), and T_2 is the first temperature reading taken at 2:00 PM.

If you do the calculations correctly, you should get $\Delta t = 7.45$ hours, which would place the time of death at approximately 6:33 AM. If this was your finding, congratulations! You have a bright future ahead of you as a forensic investigator.

Project 21: Simulate a Binary Star System

In this project, we'll explore the moves made by "star couples"—not the latest objects of celebrity gossip, but rather the stars we can see when we look up at the night sky. Many of the bright objects in the universe that appear to the naked eye like a single star are in fact two stars in close proximity that "dance" or orbit around a common center of rotation. Depending on the stars' mass, orbital velocity, and distance from each other, these *binary star systems* can create interesting and unique orbital patterns. We'll visualize these patterns by creating a binary star system Kotlin app.

The Science of Binary Star Systems

Binary star systems are governed by Newton's laws of gravity and motion. Let's consider the model binary system in two dimensions shown in Figure 5-12.

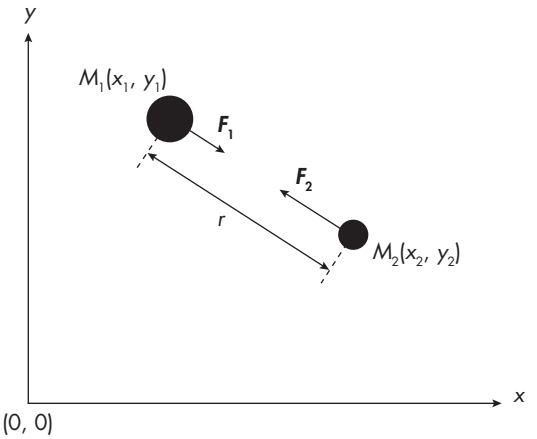


Figure 5-12: A two-body system under gravity

Our binary system is made up of two astronomical bodies, shown as dark circles in the figure. The first body, located at (x_1, y_1), has a mass of M_1, and the second body, located at (x_2, y_2), has a mass of M_2. The distance from the center of M_1 to the center of M_2 is r, which we can calculate using the Pythagorean theorem:

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.23}$$

The law of gravity says that these two bodies will be pulled toward each other. In this example, $\boldsymbol{F_1}$ and $\boldsymbol{F_2}$ are the gravitational forces acting on body 1 and body 2, respectively. Notice that these forces are *vectors*, meaning they have both magnitude and direction. (We indicate vectors with **boldface** type.) According to the law of gravity, the force acting on body 1 ($\boldsymbol{F_1}$) can be expressed as:

$$\boldsymbol{F_1} = G\frac{M_1 M_2}{r_2}(\hat{\boldsymbol{r}}) = F(\hat{\boldsymbol{r}}) \tag{5.24}$$

In Equation 5.24, G is a gravitation constant, F is the magnitude of the force vector, and $\hat{\boldsymbol{r}}$ is a unit vector pointing from body 1 toward body 2. Further, Newton's second law of motion says that the force $\boldsymbol{F_1}$ acting on body 1 of mass M_1 will create an acceleration $\boldsymbol{a_1}$ along the direction of the force such that:

$$\boldsymbol{F_1} = M_1\, \boldsymbol{a_1} \tag{5.25}$$

We can combine Equations 5.24 and 5.25 to write expressions for the x- and y-components of acceleration $\boldsymbol{a_1}$ as:

$$a_{1x} = \left(\frac{F}{M_1}\right)\left(\frac{x_2 - x_1}{r}\right) \tag{5.26}$$

$$a_{1y} = \left(\frac{F}{M_1}\right)\left(\frac{y_2 - y_1}{r}\right) \tag{5.27}$$

Finally, Newton's third law of motion requires that $\boldsymbol{F_1} = -\boldsymbol{F_2}$. Thus, we can add a negative sign to the right-hand side of Equations 5.26 and 5.27 and replace M_1 with M_2 to calculate the acceleration components acting on body 2 due to $\boldsymbol{F_2}$ as:

$$a_{2x} = -\left(\frac{F}{M_2}\right)\left(\frac{x_2 - x_1}{r}\right) \tag{5.28}$$

$$a_{2y} = -\left(\frac{F}{M_2}\right)\left(\frac{y_2 - y_1}{r}\right) \tag{5.29}$$

To simulate the orbits of a binary system, we'll first use these equations for the x- and y-components of acceleration to calculate the respective velocity components of the stars, then use those velocities to update the positions of the stars of our binary system.

The Strategy

In this section, we'll develop a simple algorithm to track the stars in a binary system once they're set in motion. First, however, we need to consider what units of measurement are appropriate to the cosmic scale of our astronomical calculations. We'll use the following:

- For mass, we'll use $M_\odot$, which is the mass of our sun. One *solar mass* is approximately 2×10^{30} kg, about 333,000 times the mass of Earth.
- For distance, we'll use the *astronomical unit (AU)*, which is the average distance between the sun and Earth. One AU equals 149,597,870.7 km.
- For time, we'll use the *solar year (yr)* as the standard unit, which is equal to 365 days, 5 hours, 48 minutes, and 46 seconds.

Given these basic units, the unit for velocity will be AU/yr, and the unit for acceleration will be AU/yr^2. This, in turn, gives us a gravitational constant G of $4\pi^2$ AU^3 yr^{-2} $M_\odot^{-1}$.

Now we're ready to move on to the algorithm, which is similar to the algorithm used in Project 19. Here are the steps we'll follow to calculate the positions of the binary stars:

1. Get the initial position and velocity vectors (in terms of x- and y-components) for the binary stars (including their masses and the gravitation constant, G). Use the system of units we just discussed.
2. Calculate the distance r between the stars at a given time t, using Equation 5.23.
3. Calculate x- and y-components of the accelerations, using Equations 5.26 through 5.29.

4. Choose a small enough time step dt and update the velocity and position vectors, using the Euler-Cromer rule. The components of the velocity and position vectors along the x-axis can be calculated as:

$$v_{x,\,t+1} = v_{x,\,t} + a_{x,\,t}\,dt \tag{5.30}$$

$$x_{t+1} = x_t + v_{x,\,t+1}\,dt \tag{5.31}$$

5. Repeat step 4 for the components along the y-axis for both stars.
6. Repeat steps 2 through 5 until the stopping condition is met. We'll set the maximum number of iterations as the stopping condition.

Of course, the whole point of making all these calculations is to be able to sit back and enjoy watching the dance steps of the stars. After each cycle through the algorithm, we'll animate the stars' motion by plotting the new positions by using the canvas feature of JavaFX. We'll also give each star a trail to better illustrate the orbital paths. A significant part of the code for this app is devoted to displaying and managing the moving objects on the canvas.

The Code

We'll review the code segments for this project in the same order as they appear in the app. Let's start with the imports and declarations that come before the `SimulateBinarySystem` application class.

```
// animation-related tools
import javafx.animation.KeyFrame
import javafx.animation.Timeline
import javafx.util.Duration

// graphics-related tools
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.stage.Stage

// math functions
import kotlin.math.*

// data class
data class Star(
    val mass: Double,
    val size: Double,
    var x: Double,
    var y: Double,
    var vx: Double,
    var vy: Double,
    var xOld: Double = 0.0,
    var yOld: Double = 0.0,
```

```
    var trailCount: Int = 0,
    val color: Color = Color.GOLD
)

// problem definition and global declarations
// initial state of the binary system
val stars = listOf(
    Star(mass = 0.73606507, size = 40.0, x = -35.0, y = 0.0,
         vx = 0.0, vy = 0.3045865, color = Color.BLACK),
    Star(mass = 0.50121162, size = 25.0, x = 51.4, y = 0.0,
         vx = 0.0, vy = -0.447307098, color = Color.BLACK)
)
val G = 4 * PI * PI

// Set canvas/animation parameters.
val canvasW = 800.0
val canvasH = 800.0
val durationMillis = 4.0
val frameCountMax = 50_000

// parameters related to star trails
val TRAIL_CODE = "YES"
val trails = Array(2) { ArrayList<Pair<Double,Double>>() }
val trailMAX = 6500
val trailSize = 2.0
val scaleFactor = 4
```

The first three lines of the import block load important tools (`KeyFrame`, `Timeline`, and `Duration`) for creating frame-by-frame animations with JavaFX. The remaining imports are similar to what we've used previously in this chapter.

After the imports, we declare a `Star` data class to hold the properties needed for representing a star, including its mass, x- and y-coordinates, and velocity components. The `size` property, specified in pixels, is the diameter of the circle that will represent a particular star; `xOld` and `yOld` will hold a copy of the star's current position vector before updating it (needed to generate its trails); `trailCount` sets the number of small dots that will follow a star to mark its trail; and `color` defines the color used to draw the star and its trail. Because this book is printed in black and white, all figures have been generated with the default color (`BLACK`), but feel free to experiment with any color you like when you run the code on your device.

The problem definition and global declaration section starts with creating `stars` as a list of two `Star` objects. We initialize each with its `mass`, `size`, current x- and y-coordinates (`x`, `y`), components of the velocity vector along the positive x- and y-axes (`vx`, `vy`), and color for drawing the star on the canvas. Since we're not going to draw the stars to proper scale, I'm just using two arbitrary sizes that might look reasonable given the size of the canvas. You could make them proportional to their masses if you have reason to believe that the stars have similar densities. Also, notice that we've set the gravitational constant `G` to $4\pi^2$ because we're using the astronomical units mentioned earlier.

In the next part of the code, we first set the width (`canvasW`) and height (`canvasH`) of the canvas to 800 pixels (without any scaling, this would be equivalent to 800 AU in both directions). The `durationMillis` parameter sets the duration per animation frame to 4.0 milliseconds. The `frameCountMax = 50_000` means the app will stop simulating the binary system after 50,000 frames. If `dt` = 1 year, that will be equivalent to observing the system for 50,000 years! You may not need to simulate the system for such a long time (you can terminate the app at any time by closing the animation window). However, it's a good idea to run a simulation for several rounds of full orbits (when the expected orbit is either a circle or an ellipse) to confirm that the simulated orbits are stable. Once this is confirmed, you can play God with the parameters and see how the movements of the stars go crazy!

The final code segment in this block sets parameters for the trails, allowing us to visualize the path traced by the stars during live simulations. By default, the `TRAIL_CODE` is set to `YES`, which means that trails for both stars will be drawn on the canvas alongside their parent stars. Each trail is composed of small dots or tracers that follow the parent star. The dots' coordinates are saved in a two-dimensional array called `trails`, where each element represents x- and y-coordinates stored as a `Pair` of type `<Double, Double>`. To ensure that the full orbital path is traced out by the trails when the orbit is stable, we set the number of trail elements to `6500`. Try to keep this number to a minimum, however; otherwise, too many dots in the trails can slow down the animation and force you to adjust the duration of the frame.

The SimulateBinarySystem Application Class

The core part of the app is the `SimulateBinarySystem` application class, where we combine Kotlin with JavaFX to drive the key parts of the app and run the animation.

```
class SimulateBinarySystem : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val canvas = Canvas(canvasW, canvasH)
        val gc = canvas.graphicsContext2D
      ❶ gc.translate((canvas.width)/2.0, (canvas.height)/2.0)
      ❷ gc.scale(1.0, -1.0)
        root.children.add(canvas)

        val scene = Scene(root, canvasW, canvasH)
        //scene.fill = Color.WHITE
        stage.title = "Binary System Simulation"
        stage.scene = scene
        stage.show()

        // -----------simulation block-----------
        // Set the background and initial positions.
      ❸ initialPositions(gc)

        // Start animation.
      ❹ val t = Timeline()
        var frameCount = 0
```

```
        val dt = 1.0
        val iterMax = 1

      ❺ val k = KeyFrame(Duration.millis(durationMillis), {
            for (i in 1..iterMax)
                updateStarPositions(stars, dt)
            drawStars(gc)
            if (TRAIL_CODE == "YES")
                updateAndDrawTrails(gc)
            frameCount += 1
            // Check the stopping condition.
          ❻ if (frameCount >= frameCountMax) {
                println("maximum limit for frameCount reached")
                t.stop()
            }
        })
      ❼ t.keyFrames.add(k)
        t.cycleCount = Timeline.INDEFINITE
        t.play()
    }
}

fun main() {
    Application.launch(SimulateBinarySystem::class.java)
}
```

The first section of this code segment is boilerplate JavaFX code that we've used before. Notice that this time we're setting the origin of the coordinate system to the center of the canvas ❶. Also, the positive y-axis direction is set to point upward ❷.

Inside the simulation block, we first call the `initialPositions()` function ❸, which draws the x- and y-axes on the canvas, places the stars on the canvas based on their position vectors, and initializes an array of type `Pair` called `trails` if the `TRAIL_CODE` is set to `YES`. Since this function is very short, I'll show it and the `drawAxes()` function, which is called from within the `initialPositions()` function, right here:

```
fun initialPositions(gc: GraphicsContext) {
    drawAxes(gc)
    stars.forEachIndexed {index, star ->
        gc.fill = star.color
        gc.fillOval(
            scaleFactor * star.x - star.size/2,
            scaleFactor * star.y - star.size/2,
            star.size, star.size)

        // Place the tracers to initial star position.
        if (TRAIL_CODE == "YES") {
            for (i in 1..trailMAX) {
                trails[index].add(Pair(star.x, star.y))
            }
        }
    }
}
```

```
fun drawAxes(gc: GraphicsContext) {
    // Draw the x- and y-axes.
    with(gc) {
        setLineDashes()
        lineWidth = 0.25
        stroke = Color.BLACK
        strokeLine(-canvasW/2,0.0, canvasW/2,0.0)
        strokeLine(0.0,-canvasH/2, 0.0,canvasH/2)
    }
}
```

Notice that we use a `forEachIndexed` structure for iteration instead of a standard `for` loop in the `initialPositions()` function. This is because we need the `index` property of the stars to create the respective trails in the second part of the function. Each star's trail consists of `trailMAX` number of dots, or tracers, which for now we initialize to the same starting position as the star itself.

Regarding the `drawAxes()` function, notice how all the graphics commands involving the graphics context `gc` have been grouped together using the scoping function `with()`. This saves us from having to type `gc` multiple times.

Getting back to the application class, the animation of the binary star system is implemented by using a combination of `Timeline` and `KeyFrame` with a `{lambda}` expression. We first create a `Timeline` variable `t` ❹ and set a few other local parameters. The parameter `frameCount` is initially set to `0` and later incremented by `1` per frame. The `dt` parameter is the time step used in updating the velocity and position vectors. We've set it to `1`, meaning each animation frame represents 1 year (this may seem too large, but it's all relative; the period for this star system is 722 Earth years long). Notice that the `iterMax` parameter is set to `1`, which means the position and velocity vectors are updated only once per frame. If we wanted, we could use a different `dt` and `iterMax` combination to carry out `iterMax` updates of the position and velocity vectors before changing frames.

Inside the `KeyFrame` class instance ❺, the `{lambda}` expression specifies what happens between the frame updates at a frequency specified by the `Duration.millis()` function. First, we call `updateStarPositions()` to calculate each star's acceleration, velocity, and position. We then call `drawStars()` to update the positions of the stars on the canvas, and finally, we call `updateAndDrawTrails()` to trace out the trails for both stars. (We'll review these functions separately.) Before exiting the `KeyFrame` block, we check for the stopping condition: if `frameCount` (the number of frames displayed) reaches `frameCountMax`, the timeline ends, and the animation is terminated ❻.

The final three lines complete the `Timeline` implementation. First, we supply the `KeyFrame` instance (`k`) to the timeline ❼. Next, we set `cycleCount` to `INDEFINITE`, which means continue the timeline until a stopping condition is met inside the `KeyFrame` or elsewhere in the program. Finally, `t.play()` starts the animation and sets the stars in motion.

The updateStarPositions() Function

The `updateStarPositions()` function implements the physics of binary stars in motion by using the formulas and algorithms we've discussed. The function takes in two arguments—a list of type `Star` called `stars` and the time step `dt`—and doesn't return anything, since the function makes its updates directly to the globally accessible properties of the stars.

```
fun updateStarPositions(stars: List<Star>, dt: Double) {
    val rx = stars[1].x - stars[0].x
    val ry = stars[1].y - stars[0].y
    val r = sqrt(rx * rx + ry * ry)

  ❶ val force =
        G * stars[0].mass * stars[1].mass / (r * r)

    var sign = 1
    for (star in stars) {
      ❷ // Update the acceleration, velocity, and position of stars.
        val acceleration = force / star.mass
        val ax = acceleration * rx / r
        val ay = acceleration * ry / r
        star.vx += sign * ax * dt
        star.vy += sign * ay * dt

      ❸ // These will be needed for updating trails.
        star.xOld = star.x
        star.yOld = star.y

        star.x += star.vx * dt
        star.y += star.vy * dt
      ❹ sign = -1
    }
}
```

The function starts by calculating the distance between the two stars by using Equation 5.23. We then use the scalar part of Equation 5.24 to calculate the magnitude of the gravitational force that both stars will experience ❶. Next, we iterate over the `stars` list and update the acceleration, velocity, and position for both stars per Equations 5.26 through 5.31 ❷. Notice that we save the current position vectors in the `xOld` and `yOld` properties of the stars before updating `x` and `y` ❸. We'll need these so the stars' trails will lag the stars themselves by one time step. Also notice how the variable `sign` switches from `1` to `-1` at the end of the first iteration ❹. This inverts the direction of the force acting on the second star to ensure that $\boldsymbol{F}_1 = -\boldsymbol{F}_2$.

The drawStars() Function

The `drawStars()` function simply updates the positions of the stars on the canvas to make them move. It takes in one parameter, the graphics context, and returns nothing.

```
fun drawStars(gc: GraphicsContext) {
  ❶ gc.clearRect(-canvasW/2, -canvasH/2, canvasW, canvasH)
  ❷ drawAxes(gc)

    // Connect the centers of the stars.
  ❸ with (gc) {
        lineWidth = 0.5
        stroke = Color.BLACK
        setLineDashes(2.0,4.0,4.0,2.0)
        strokeLine(
            scaleFactor*stars[0].x,
            scaleFactor*stars[0].y,
            scaleFactor*stars[1].x,
            scaleFactor*stars[1].y)
    }

    // Draw the stars using updated positions.
  ❹ for (star in stars) {
        gc.fill = star.color
        gc.fillOval(
            scaleFactor * star.x - star.size/2,
            scaleFactor * star.y - star.size/2,
            star.size, star.size)
    }
}
```

The function starts by clearing the canvas ❶. Then we redraw the objects in three stages. First, we call the `drawAxes()` function, which we've already reviewed, to draw the axes ❷. Since this happens first, the other objects (for example, the stars and the trails) will be drawn on top of the axes.

Next, we connect the centers of the stars by using a dashed line ❸. The argument sequence `(2.0, 4.0, 4.0, 2.0)` means the dash lengths will be set in cycles of 2, 4, 4, and 2 pixels. This line helps identify the center of rotation (the point where it intersects the x-axis) and will grow and shrink like a spring with the movements of the stars, making the simulation more interesting. For a circular and concentric orbit, this line also helps visually confirm that at any moment the binary stars are placed in diametrically opposite positions about the center of rotation.

Finally, we use a `for` loop to iterate over the stars and draw them in their new positions ❹. Notice the use of `scaleFactor` while drawing the stars. This is a global variable that allows us to change the scale of the simulation on the fly without going through more complicated rescaling schemes that we've used elsewhere in the book. For this problem, since the orbit shapes and sizes can vary significantly during the simulations, I suggest that you adjust this scale factor based on your own setup. I used `scaleFactor = 4` for the first simulation (for stable circular orbits) and `scaleFactor = 1` for the second simulation (for stable elliptical orbits). In the latter case, the distance between the stars r varies significantly during the simulation, and we need to allocate more space on the canvas to fully outline the orbits.

The updateAndDrawTrails() Function

The two-in-one `updateAndDrawTrails()` function updates the trails and draws them at the same time. It takes one parameter, `gc`, and doesn't return anything, just like the `drawStars()` function.

```
fun updateAndDrawTrails(gc: GraphicsContext) {
    // Update the trails.
    stars.forEachIndexed { index, star ->
      ❶ if (star.trailCount >= trailMAX) star.trailCount = 0
      ❷ trails[index][star.trailCount] =
                      Pair(star.xOld, star.yOld)
        star.trailCount += 1
    }

    // Draw the trails.
    trails.forEachIndexed { index, trail ->
      ❸ gc.fill = stars[index].color
      ❹ for (point in trail) {
            gc.fillOval(
                scaleFactor * point.first - trailSize / 2,
                scaleFactor * point.second - trailSize / 2,
                trailSize, trailSize
            )
        }
    }
}
```

The first block in this function iterates over the `stars` list and updates the positions of the trail elements. Recall that the `initialPositions()` function started all the tracers at the same position as the star itself, but we haven't drawn any of them yet. Now we move the tracers one at a time (for each star). First, we check to see that unassigned tracers are ready to be moved to a new location by ensuring that `star.trailCount < trailMAX`. Otherwise, we reset `trailCount` to `0` ❶. Each time a star moves to a new position, we assign the star's old position to the next available tracer—the one at position `[index][star.trailCount]` in the `trails` array ❷. This is why we always save the old positions of the stars before updating them in the `updateStarPositions()` function.

This process continues as long as tracers remain to be moved from the initial position. Once all the tracers have been assigned new positions on the canvas (and the stars continue to move), resetting `trailCount` lets us recycle them by bringing the last one to the first position (right next to the star).

Now that the tracer positions have been updated, the second block in the function draws the trails for both stars. For each trail, we use the same color as the color of the star it follows ❸ to draw the tracers. Finally, all the trails are drawn on the canvas, based on their most recent coordinates ❹. Once all the tracers are placed on the canvas, each trail will follow its parent star.

The Result

When you run the app, you should see a dynamic simulation of the binary star system in motion, as opposed to a static image. You'll see the stars continuously dancing around each other on the screen until you close the window or wait for the `frameCount` to hit its limit. Figure 5-13 shows screenshots of two different runs of the app.

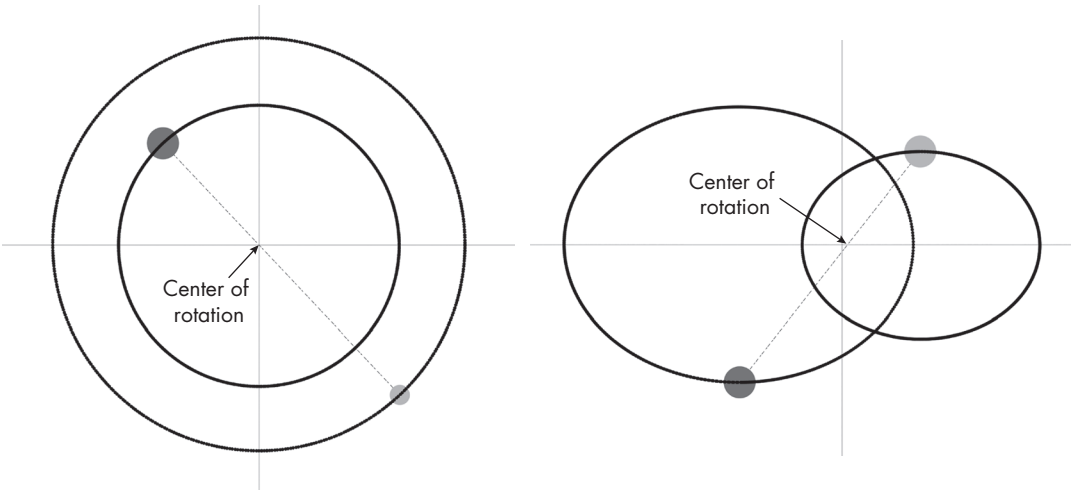


Figure 5-13: A binary star system with concentric, circular orbits (left) and a binary star system with elliptical orbits (right)

The example on the left shows a binary star system with concentric, circular orbits, meaning one star's orbit is entirely contained within the other. For this, we used the initial conditions shown in the code listings. The example on the right simulates a system with elliptical orbits by using the same initial conditions, except `x = -50.0` for star 1, `x = 90.0` for star 2, and `scaleFactor = 1`.

I encourage you to play around with the simulation by tweaking the initial conditions. Depending on the parameters, you may end up with an unstable system, in which case the stars fly off the screen or crash into each other. In fact, it's much harder to create a stable system like the ones shown in the figure. This instability also happens with real binary star systems as they get perturbed by a variety of external forces that offset the delicate balance needed to maintain a stable orbit.

EXERCISE

The 2D binary star system we developed in this section is helpful for exploring the interactions between the stars driven solely by the gravitational forces created by their own masses. However, many known binary star systems have

other companions, such as one or more planets orbiting the stars. Try modifying the project code to add a third object, a planet, attached to one of the stars. In other words, you'll convert the two-body problem into a three-body problem and solve the resulting system by using the same principles and tools.

Hint: Adding a planet is no different than adding another star with a small mass. To create a stable system, you'll have to do some online research to find real astronomical examples and use that data to set the initial conditions. Most of the functions in the app will require no change, as they're written in a generic fashion that can handle any number of stars or trails (subject to the limits set by the CPU/GPU and memory of your device). However, you'll need to change the `updateStarPositions()` function to consider the impact of the stars on the planet's orbit, and vice versa. To be realistic, you could also add a feature to check if the stars and the planet get too close or collide, and what happens as a result.

Summary

In this chapter, we used Kotlin and laws of physics to solve a variety of science problems—from the projectile motion of cannonballs and water jets, to swinging pendulums and cooling cups of coffee, to the out-of-this-world dance of a binary star system. Along the way, we developed increasingly complex mathematical models of physical systems to study their behavior. To do so, we went beyond the basics of Kotlin, harnessing JavaFX's visualization and animation tools to effectively display the results of our code.

Resources

Bate, Roger R., Donald D. Mueller, Jerry E. White, and William W. Saylor. *Fundamentals of Astrodynamics.* 2nd ed. Mineola, NY: Dover, 2020.

Bennett, Andrew G. "Runge-Kutta Methods." Accessed June 15, 2024. *https://onlinehw.math.ksu.edu/math340book/chap1/xc1.php.*

Cromer, Alan. "Stable Solutions Using the Euler Approximation." *American Journal of Physics* 49, no. 5 (May 1981): 455–459. *https://doi.org/10.1119/1.12478.*

Demanet, Laurent. "Introduction to Numerical Analysis." MIT OpenCourseWare, 2012. Accessed June 15, 2024. *https://ocw.mit.edu/courses/18-330-introduction-to-numerical-analysis-spring-2012/.*

Halliday, David, Robert Resnick, and Jearl Walker. *Fundamentals of Physics.* 12th ed. New York: Wiley & Sons, 2021.

Seyr, Alexander Josef. "Numerical Simulation of the Planetary Motions in the Solar System with Runge Kutta Methods." November 6, 2020. *https://static.uni-graz.at/fileadmin/_Persoenliche_Webseite/puschnig_peter/unigrazform/Theses/BachelorThesis_Seyr_2020.pdf.*

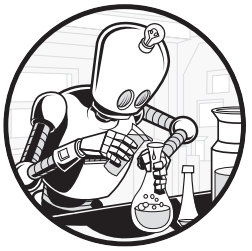
Young, Hugh, and Roger Freedman. *University Physics.* 15th ed. New York: Pearson, 2020.

PART III

RECURSION, SORTING, AND SEARCHING

6

RECURSIVE FUNCTIONS AND FRACTALS



Fractals are enchanting geometric shapes where the real meets the imaginary (imaginary numbers, that is). Repeating patterns keep emerging forever as we continue to zoom in on them. Mathematicians are still trying to define fractals precisely, but they agree on these key features: self-similarity (the way smaller parts of a fractal echo the whole) and the ability to scale ad infinitum.

In this chapter, we'll use Kotlin to draw a variety of fractals and explore their enigmatic beauty. Our main goal will be to codify the mathematical logic embedded in the design of each fractal, re-create those fractals, and possibly discover new ones by playing with the design parameters.

The Concept of Fractals

The notion of fractals is full of dualisms and contradictions. Benoit Mandelbrot, considered the father of fractal geometry, coined the term *fractal* from the Latin word *fractus*, meaning fractured or broken, yet detailed images of fractals are hardly fractured, instead showing a continuous flow of intricate patterns. Well-known fractals such as the Julia set and the Mandelbrot set display extremely complex patterns, yet the equations that generate those patterns are very simple. The fact that a fractal map contains an infinite amount of information that can be condensed into a single equation is a major discovery of the 20th century, a wonder that reveals the hidden beauty of mathematics.

Fractals aren't just exotic mathematical objects, however. They help us see the "geometry of nature," a term popularized by Mandelbrot. Indeed, fractal patterns are abundant throughout nature, seen in snowflakes, the branching of trees, the human nervous and circulatory systems, coastlines, clouds, hurricanes, and the spiral shapes of galaxies. The theory of fractals also has found applications in the physical and biological sciences, engineering, and information technology and given birth to new areas of research, such as chaos theory and its application in studying complex dynamic systems.

Before we start coding, let's review a simple fractal to explore a couple of its key features. Figure 6-1 shows the Sierpiński triangle, a geometric fractal named after the famous Polish mathematician Wacław Sierpiński, who created it in 1915. This fractal is constructed by repeatedly connecting the midpoints of the three sides of an equilateral triangle and all the resulting subtriangles.

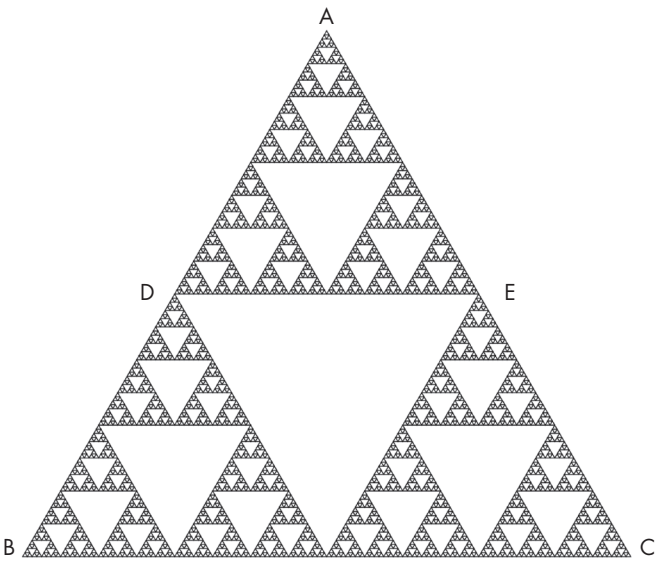


Figure 6-1: The Sierpiński triangle

The inherent beauty of this fractal derives from its self-similarity: look at any of the smaller triangles inside the outermost triangle and you'll see a mini version of the original figure. Moreover, if we amplify or scale up one of the smaller triangles, we'll find many more of the same inside it. For example, Figure 6-2 shows a zoomed-in view of just the triangle formed by points A, D, and E from Figure 6-1. The result is the same as the original triangle.

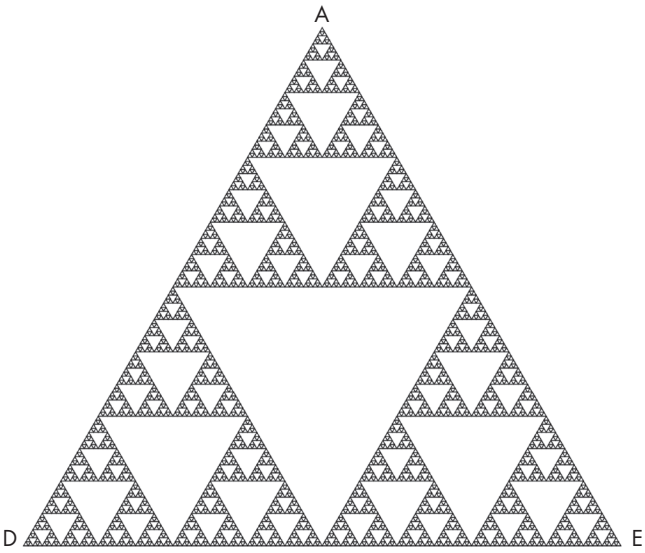


Figure 6-2: A zoomed-in section of the Sierpiński triangle, illustrating the fractal's ability to scale

No theoretical limit is placed on how deep we can go—even the smallest triangle from the original figure, when amplified, produces the same result. A fractal can scale up forever.

Fractals also have an interesting property called *fractal dimension*. Unlike familiar geometric objects such as lines, areas, and volumes that have one, two, and three dimensions, respectively, a fractal can have a fractional number of dimensions. The Sierpiński triangle, for example, has 1.585 dimensions: it's more than a one-dimensional line but less than a two-dimensional area. This is because it fits in a 2D plane but doesn't completely fill the area that defines the fractal boundary. You can find a more formal and in-depth discussion of this topic in many of the excellent resources listed at the end of the chapter.

Recursive Functions

The most efficient way to draw a self-similar pattern that can theoretically keep repeating itself forever is to use a *recursive function*. This is a function that continues to call itself from within its own function body until a stopping condition is met. Once we identify the basic building block of a fractal,

we can write a function to draw that building block and then let the function repeatedly call itself, until the fractal pattern is sufficiently developed. In the sections that follow, we'll use this approach to generate several well-known geometric fractals, including the Sierpiński triangle, the Sierpiński carpet, and a fractal tree. But first, let's get a feel for how recursion works by writing a function to calculate the factorial of an arbitrary positive integer n.

The factorial of a number n is defined as:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

In other words, $n!$ equals the product of all integers from 1 to n. For example, $3! = 3 \times 2 \times 1 = 6$, $4! = 4 \times 3 \times 2 \times 1 = 24$, and so on (by convention, $1! = 1$). Thinking recursively, we can also define the factorial of n as the product of n and the factorial of the next smaller integer $(n-1)$:

$$n! = n \times (n-1)!$$

Using this modified form of the equation, we can write a recursive function to calculate the factorial of n.

```
fun main() {
    // Find the factorial of a positive integer.
    val n = 5
    val factorial = getFactorial(n)
    println("The factorial of $n is: $factorial")
}

fun getFactorial(n: Int) : Int {
    if (n <= 1) return 1
      ❶ else return n * getFactorial(n - 1)
}
```

Something interesting is happening here. The function `getFactorial(n)` calls itself from inside the function body ❶ and thus kicks off a loop that runs until the most recent value of `n` equals `1`. At that point, the last call to the function returns `1` and the loop terminates, returning the value of the factorial.

Table 6-1 shows how the key function parameters change at each iteration, leading to the factorial value in the end.

Table 6-1: Anatomy of the Recursive Function `getFactorial(n)` for $n = 5$

Iteration	Argument (n)	Test (n == 1)	Returned value
1	5	False	5 × `getFactorial(4)`
2	4	False	5 × 4 × `getFactorial(3)`
3	3	False	5 × 4 × 3 × `getFactorial(2)`
4	2	False	5 × 4 × 3 × 2 × `getFactorial(1)`
5	1	True	5 × 4 × 3 × 2 × 1 = 120

Notice that a recursive function is essentially a loop that keeps on calling itself. We must therefore provide a stopping condition (in this case, `n <= 1`). Without one, the function will be trapped in an infinite loop.

It may not be obvious just yet, but recursive functions can make coding significantly simpler (the alternative would be to use complex nested loops). This becomes especially true when the recursive function needs to make multiple calls to itself, using different state variables. We'll see this idea in the upcoming projects.

Tail recursion is a technique in which a recursive function implements tail call optimization (TCO). This allows the compiler to avoid consuming additional *stack space* (a region in memory that stores information in a last-in, first-out order) for each recursive call. Instead, the compiler reuses the same stack space for each call, which can help prevent stack overflow errors.

To use tail recursion in Kotlin, the recursive call must be the very last call of the method. This means that the function must return the result of the recursive call directly, without performing any additional operations on it.

Here's an example of an alternative version of the code that calculates the factorial of a number by using tail recursion:

```
fun main() {
    val n = 5
    val factorial = getFactorial(n)
    println("The factorial of $n is: $factorial")
}

❶ tailrec fun getFactorial(n: Int, result: Int = 1): Int {
    return if (n <= 1) {
        result
    } else {
        getFactorial(n - 1, n * result)
    }
}
```

In this implementation, the `main()` function remains unchanged, but the `getFactorial()` function has been modified.

Notice how the `getFactorial()` function is defined with the `tailrec` keyword ❶, which indicates that it should be optimized for tail call recursion. The function takes two arguments: `n`, which is the number whose factorial will be calculated, and `result`, which is the current result of the calculation. If `n` is `0` or `1`, the function returns `result`. Otherwise, it calls itself recursively with `n - 1` as the new value of `n` and `n * result` as the new value of `result`. This continues until `n` is `0` or `1`, at which point the final result is returned.

For large numbers (`n`), this implementation not only saves memory but may also require less time to complete the computations.

Project 22: The "Hello, World!" of Fractals

Our first foray into the world of fractals will be a simple one: we'll write a recursive function to draw a series of concentric squares, each one smaller

than and located symmetrically inside the previous one. We'll visualize the fractal by using the canvas feature of JavaFX.

The Strategy

The JavaFX canvas allows us to draw a polygon based on the coordinates of its vertices. We need a way to calculate the four vertices of a square, given the x- and y-coordinates of one of the vertices and the length of any side (for a square, they're all equal). We'll use the scheme outlined in Figure 6-3. Keep in mind that the default origin (0, 0) of the canvas is located at the top-left corner.

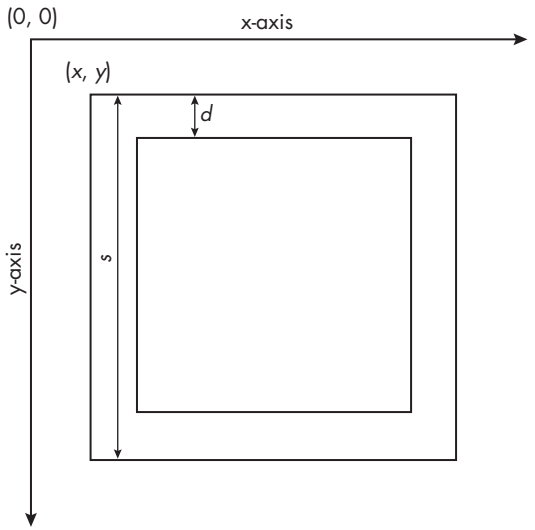


Figure 6-3: The strategy for drawing concentric squares

We'll start by choosing the coordinates for the top-left vertex of the outermost square, x and y. Given those, calculating the square's other coordinates is simply a matter of adding the side length s as appropriate. This gives us everything we need to draw the outermost square.

Next, we'll call a recursive function to draw the inner square or squares by using the following function parameters: the x- and y-coordinates for the top-left vertex of the most recently drawn square; the side length of that square, s; and a shrinkage factor, k (a percentage setting how much smaller the sides of the next square should be). We'll use this information to calculate d in Figure 6-3, the distance from the top of one square to the top of the next, as:

$$d = ks/2$$

That in turn will let us update the top-left coordinates and the side length of the next square as follows:

$$x = x + d$$

$$y = y + d$$

$$s = s - 2d$$

To prevent our recursive function from devolving into an infinite loop, we'll provide a stopping condition in the form of a global constant limiting the number of iterations.

The Code

Following the steps we just discussed, we first create the required variables and draw the outer square.

```
val x = 50.0
val y = 50.0
val s = 400.0
❶ val k = 0.15   // Reduce the side by 15%.
gc.fill = Color.BLACK
❷ gc.strokePolygon(doubleArrayOf(x, x, x + s, x + s),
    doubleArrayOf(y, y + s, y + s, y), 4)

// Call the resursive function.
drawSquares(x, y, s, k, gc)
```

This code places the top-left corner of the outer square at (50, 50) relative to the origin and sizes the square to 400×400. (The canvas itself is 500×500, as we'll see later.) At each iteration, the sides of the inner square will be reduced by 15 percent ❶; feel free to play with this value. The squares will be drawn in black, as specified by `Color.BLACK`, on a white background (the default). Calling `gc.strokePolygon()` draws a square ❷ (`gc` is the `GraphicsContext` object needed to draw to the canvas). This method requires that the coordinates of the vertices be passed in two separate `DoubleArrays`, one for the x-values and one for the y-values; the last parameter indicates the number of vertices—in this case, for a square, four.

Now let's implement our recursive `drawSquares()` function.

```
fun drawSquares(_x: Double, _y: Double, _s: Double,
                k: Double, gc: GraphicsContext) {

    if (iter <= ITER_MAX) {
        val d = 0.5 * k * _s
        val x = _x + d
        val y = _y + d
        val s = _s - 2 * d

        gc.strokePolygon(doubleArrayOf(x, x, x + s, x + s),
            doubleArrayOf(y, y + s, y + s, y), 4)

        iter += 1
      ❶ drawSquares(x, y, s, k, gc)
    }
}
```

The algorithm starts by checking the stopping condition: if `iter`, which has an initial value of `1` and is incremented by `1` per recursion, exceeds the maximum number of iterations set by `ITER_MAX`, then the loop will stop drawing squares on the canvas, and the program will terminate normally. Otherwise, we calculate a new set of parameters and proceed to draw another square. Notice that we used an underscore as a prefix for the x- and y-coordinates of the top-left vertex, as well as for the length of the side, while receiving parameter values. This naming convention allows us to use the same variable names inside the function as we did outside in the application class.

We then recursively call the `drawSquares()` function with the updated parameter values to draw the next inner square ❶. Figure 6-4 shows the program output with an `ITER_MAX` value of `22`, which means that 22 squares are inside the outermost square.

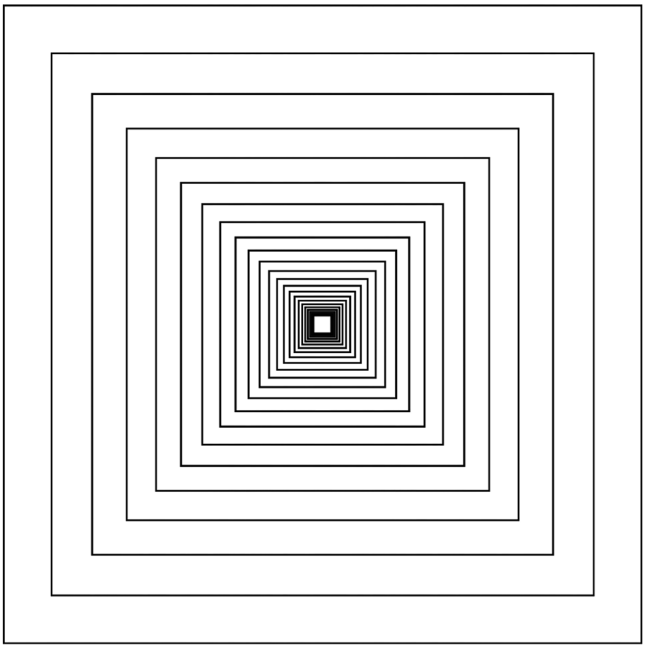


Figure 6-4: A simple fractal made from concentric, nonrotating squares

Our concentric squares are a simple example of a geometric fractal. It exhibits self-similarity in that if we removed a few of the outer squares, the resulting image, when scaled up, would look the same, even if we iterated many more times.

Here's the complete code for the app, including the boilerplate JavaFX components adapted for Kotlin.

```
import javafx.application.Application
import javafx.geometry.Pos
import javafx.scene.Scene
```

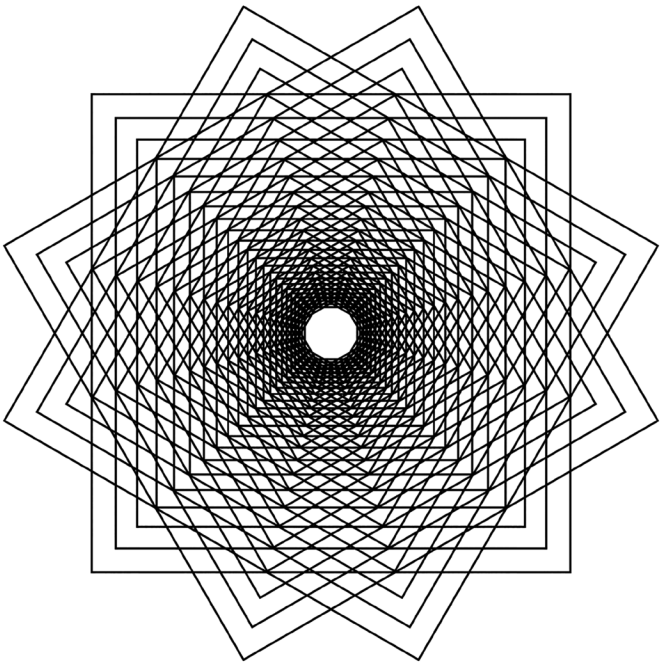
```
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.FlowPane
import javafx.scene.paint.Color
import javafx.stage.Stage

// global variables
val ITER_MAX = 22
var iter = 1

// beginning of the Application class
class GeometricFractal : Application() {
    override fun start(stage: Stage) {

        val canvas = Canvas(600.0, 600.0)
        val gc = canvas.graphicsContext2D

        val rootNode = FlowPane()
        rootNode.alignment = Pos.CENTER
        rootNode.children.add(canvas)

        val scene = Scene(rootNode, 600.0, 600.0)
        stage.title = "Geometric Fractal"
        stage.scene = scene
        stage.show()

        // problem-specific code segment
        val x = 50.0
        val y = 50.0
        val s = 400.0
        val k = 0.15
        gc.fill = Color.BLACK

        gc.strokePolygon(doubleArrayOf(x, x, x + s, x + s),
            doubleArrayOf(y, y + s, y + s, y), 4)

        drawSquares(x, y, s, k, gc)
    }
}

fun main() {
    Application.launch(GeometricFractal::class.java)
}

fun drawSquares(_x: Double, _y: Double, _s: Double,
                k: Double, gc: GraphicsContext) {

    if (iter <= ITER_MAX) {
        val d = 0.5 * _s * k
        val s = _s - 2 * d
        val x = _x + d
        val y = _y + d
        gc.strokePolygon(
            doubleArrayOf(x, x, x + s, x + s),
            doubleArrayOf(y, y + s, y + s, y), 4)
```

```
        // Update counter.
        iter += 1
        // recursive call
        drawSquares(x, y, s, k, gc)
    }
}
```

By introducing minor variations to this code, you can create other, similar geometric fractals, such as concentric rectangles or circles. Before you attempt the practice exercise, I encourage you to experiment with these variations.

EXERCISE

Modify your Project 22 code to make the fractal more interesting by drawing three squares during each iteration, rotating the second and the third by preset amounts (for example, 30 degrees and 60 degrees with respect to the y-axis). Here's what the program output might look like:



Project 23: Draw the Sierpiński Triangle

For this project, we'll draw the Sierpiński triangle (see Figure 6-1) using an approach similar to Project 22. First, we'll identify the defining features of

the fractal, including its geometric properties, and then write a recursive function to do most of the hard work. We'll use the same JavaFX template we used for Project 22 to create the graphical output.

Recall that the Sierpiński triangle is made by taking a triangle and joining the midpoints of its sides, producing three child triangles, then joining the midpoints of the child triangles' sides, and so on, recursively, ad infinitum. Typically, the triangles are equilateral, meaning they have three equal sides and internal angles of 60 degrees. We'll make use of these basic properties to come up with an efficient strategy for creating the fractal. However, working with equilateral triangles isn't strictly required; you can use the steps outlined in this section to create an isosceles Sierpiński triangle, with minor adjustments to the code.

The Strategy

To draw the Sierpiński triangle, we need a few key parameters: the x- and y-coordinates of the parent (outermost) triangle's top vertex, where the two inclined sides meet, and the parent triangle's base (*b*) and height (*h*). Figure 6-5 shows these parameters, including some additional ones for the child triangles that will be used in our code.

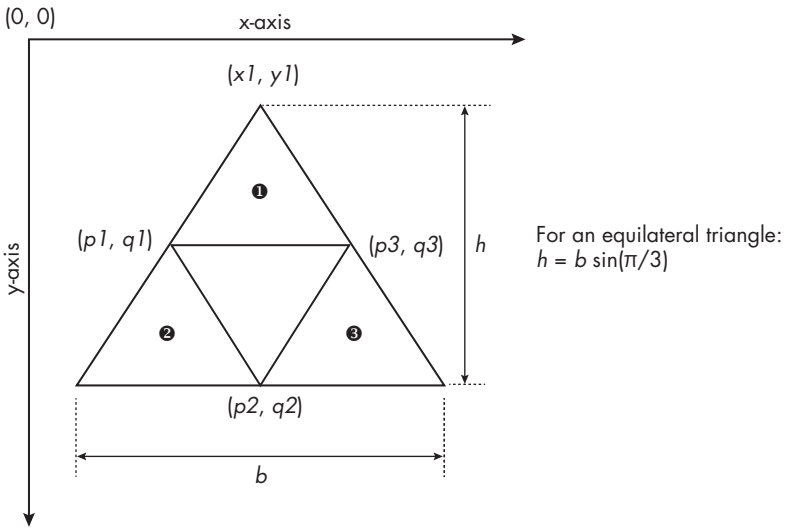


Figure 6-5: The strategy for drawing the Sierpiński triangle

We'll first draw the parent triangle, as we did for the concentric squares example. We'll then call a recursive function to connect the midpoints of the triangle's three sides, whose coordinates are calculated in terms of the parent triangle's base, height, and top vertex. Since this creates three smaller child triangles (as opposed to one smaller square), the recursive function will have to call itself multiple times, once for each of the child triangles ❶ ❷ ❸. We'll keep using recursion until a stopping condition is met.

The Code

We'll discuss the code in three parts. First, we'll import the required math functions and declare some global parameters and a data class (we'll reuse the rest of the JavaFX-specific codes from Project 22).

```
import kotlin.math.sin
import kotlin.math.PI
import kotlin.math.pow

// global parameters
val BASE = 500.0
val DEPTH = 7
val baseMin = BASE * ((0.5).pow(DEPTH))

data class Vertices(var p1: Double, var q1: Double,
                    var p2: Double, var q2: Double,
                    var p3: Double, var q3: Double)
```

The first global parameter, `BASE`, represents the base of the parent (outer) triangle, which is set to 500. The second parameter, `DEPTH`, is the number of iterations (how many times inner triangles are drawn), but we use it only to calculate our real stopping condition, `baseMin`. This is the smallest base length of the inner triangles when we stop recursion. It's more practical to think in terms of a minimum side length because how small an object we can draw is limited by the pixel size of the screen, as well as by our ability to see small objects. It wouldn't make sense to let the recursion run an arbitrary number of times (say, 50 or 100 times) while creating a static image.

The `baseMin` parameter is linked to the number of iterations `DEPTH` as:

$$\texttt{baseMin} = \texttt{BASE} \times \left(\frac{1}{2}\right)^{\texttt{DEPTH}}$$

This means `baseMin` decreases exponentially as `DEPTH` increases, which is why just 5 to 10 iterations can be adequate for bringing out the key features of simple geometric fractals. In this case, since we're using a `DEPTH` value of `7` and a `BASE` of `500`, `baseMin` will be `3.906`.

We'll use the data class `Vertices` to store and return the vertices of internal triangles, with the help of the `getVertices()` function (which we'll discuss shortly).

Next, let's have a look at the problem-specific code segment that defines and draws the parent triangle. The top vertex is set to (300.0, 50.0), and the base b and height h are set to be 500.0 and 500 $\sin(\theta)$, respectively, where θ is $\pi/3$ radians (or 60 degrees).

```
// problem-specific component inside the application class
val b = BASE
val h = b * sin(PI / 3.0)
val x1 = 300.0
val y1 = 50.0
```

```
val x = doubleArrayOf(x1, x1 - b/2, x1 + b/2)
val y = doubleArrayOf(y1, y1 + h, y1 + h)
// Draw the outermost triangle.
gc.strokePolygon(x, y, 3)
// Call the recursive function.
drawTriangle(x1, y1, b, h, gc)
```

We draw the parent triangle with a call to the `gc.strokePolygon()` function, passing the x- and y-coordinates of the three vertices and the number of vertices (in this case, three). We then make a single call to the recursive function `drawTriangle()` to generate the child triangles. Notice we're following the same basic steps we used in Project 22 to draw the concentric squares.

Finally, we'll define our recursive function to draw the inner triangles. We'll also define a helper function for calculating the midpoints of a triangle.

```
fun drawTriangle(x1: Double, y1: Double, base: Double,
                 height: Double, gc: GraphicsContext) {

  ❶ if (base > baseMin) {
        val (p1, q1, p2, q2, p3, q3) =
            getVertices(x1, y1, base, height)

        val p = doubleArrayOf(p1, p2, p3)
        val q = doubleArrayOf(q1, q2, q3)
        gc.strokePolygon(p, q, 3)

        // recurse for nonempty child triangles
      ❷ drawTriangle(x1, y1, base/2, height/2, gc)
      ❸ drawTriangle(p1, q1, base/2, height/2, gc)
      ❹ drawTriangle(p3, q3, base/2, height/2, gc)
    }
}

fun getVertices(x1: Double, y1: Double, base: Double, height: Double) =
                Vertices(x1 - base/4, y1 + height/2, x1, y1 + height,
                         x1 + base/4, y1 + height/2)
```

Our `drawTriangle()` function follows a similar pattern to our `drawSquares()` function from Project 22, but a couple of important differences exist. First, the stopping condition is now set with respect to `baseMin` ❶ instead of the maximum number of iterations (we've already discussed why that is the case). Second, we make three separate recursive calls inside the `drawTriangle()` function (instead of a single recursive call as in the previous project) to make sure that all three child triangles generated at each step contribute to creating the final fractal image. To see how this works, have a look at Figure 6-6.

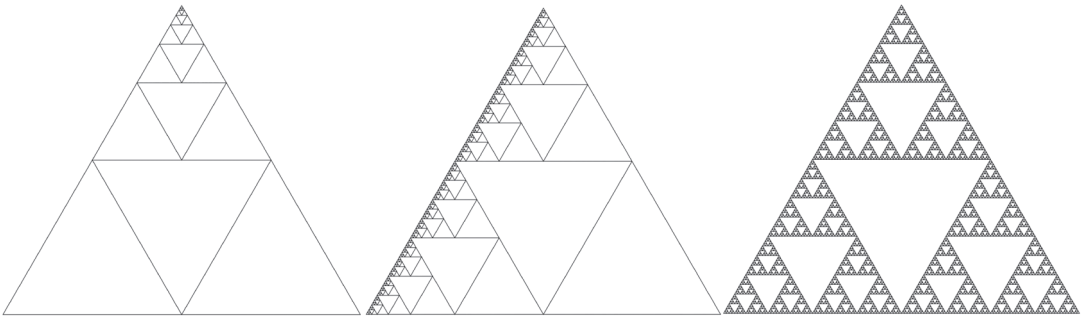
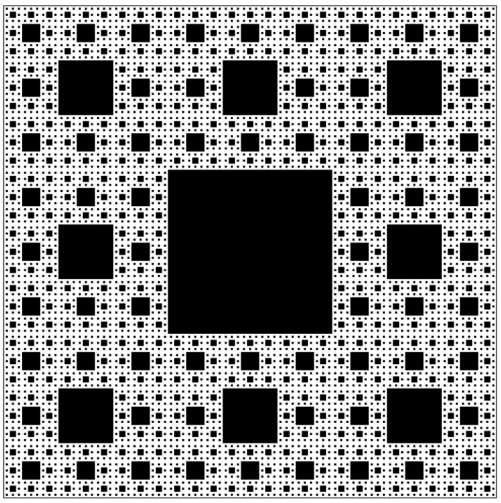


Figure 6-6: Evolution of the Sierpiński triangle with recursive calls (n = 7)

If we made only the first recursive call ❷ from inside the `drawTriangle()` function, the final figure will subdivide only the top child triangles, as shown on the left side of Figure 6-6. The center image shows what would happen if we made two recursive calls ❷ ❸; now the top and left child triangles are subdivided. Finally, the image on the right is what we want—the complete Sierpiński triangle, created by making three back-to-back recursive calls to the `drawTriangle()` function, ❷ ❸ ❹, with updated parameter values for the child triangles.

EXERCISE

Taking a cue from Project 23, develop a strategy and program for drawing the Sierpiński carpet, shown here:



Hint: Start by noting the similarity with the Sierpiński triangle. Instead of connecting midpoints of the sides, divide the parent square into nine equal child squares, and color the square at the center black. Continuing with the analogy, make eight recursive calls with updated parameters, one call for each of the unfilled child squares. (Instead of making eight separate calls, use a couple of nested `for` loops to make those calls more efficient.) As with the Sierpiński triangle, the sides of the child squares will shrink exponentially as the number of iterations goes up. The image shown above was generated with $n = 5$.

Project 24: Create a Fractal Tree

As the final project on simple geometric fractals, we'll draw a beautiful fractal tree. The tree will start as a single line (the trunk) that splits into two branches. Then each branch will itself split into two more branches, and so on.

The Strategy

The core strategy for this project is straightforward: draw a line, then recursively draw two more lines from the endpoint of that line, each at an angle relative to the parent line. Figure 6-7 outlines the strategy and the features we'll have to incorporate into the code.

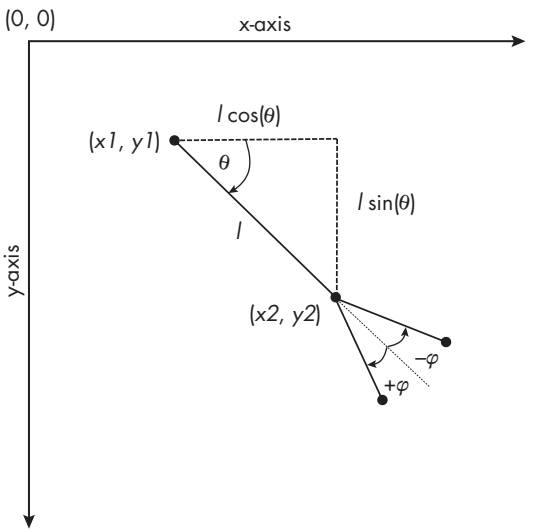


Figure 6-7: The strategy for drawing the fractal tree

First, we'll choose a starting point $(x1, y1)$ for the parent line, a length l, and θ, the parent line's angle with respect to the x-axis. Using

trigonometry, we calculate the coordinates for the endpoint of the parent line (which is also the starting point of the two child lines) as $x1 + l\cos(\theta)$ and $y1 + l\sin(\theta)$. The child lines are shorter than the parent line (we'll choose a shrinkage factor) and branch out from the parent line by some arbitrary angle φ, such that the new lines will be drawn at angles $(\theta + \varphi)$ and $(\theta - \varphi)$ relative to the x-axis, respectively. Keep calculating endpoints, drawing new lines, and branching out until a stopping condition is met (for example, when the new branches become too small).

The Code

In this example, we do not have any global variables or parameters, and the import block for math functions looks like this:

```
import kotlin.math.PI
import kotlin.math.sin
import kotlin.math.cos
```

Hence, the first code snippet we'll discuss covers the problem-specific parameter values within the application class and the call to the recursive function `drawTree()`.

```
val x = canvas.width / 2.0
val y = canvas.height - 100
val len = 55.0
val angle = -PI / 2
val phi = PI / 10

gc.stroke = Color.GRAY
drawTree(x, y, angle, len, phi, gc)
```

We center the starting point of the parent line horizontally on the canvas and place it 100 pixels above the bottom (we're assuming a canvas size of 600×600). The starting angle (`angle`) of $-\pi/2$ draws the tree in an upright position. We set the branch-out angle of the child lines relative to the parent line, φ (`phi`), to $\pi/10$, which produces optimal branching for the given parameters. Unlike in our previous fractal projects, we haven't drawn anything yet. This time, the parent line is drawn inside the recursive function, since it uses the same code as drawing the child lines. We'll define that function next.

```
private fun drawTree(x1: Double, y1: Double, theta: Double,
                     len: Double, phi: Double,
                     gc: GraphicsContext) {

  ❶ if (len > 10) {
        val x2 = x1 + len * cos(theta)
        val y2 = y1 + len * sin(theta)
        gc.strokeLine(x1, y1, x2, y2)

      ❷ drawTree(x2, y2, theta + phi, len - 4, phi, gc)
      ❸ drawTree(x2, y2, theta - phi, len - 4, phi, gc)
```

```
    } else {
        gc.fill = Color.BLACK
        gc.fillOval(x1 - 2, y1 - 2, 4.0, 4.0)
    }
}
```

The termination condition stops recursion when the `len` parameter passed into the function becomes less than or equal to `10` ❶. At each iteration, we calculate the endpoint of the current line and draw it with the `gc.strokeLine()` function. We then make two recursive calls ❷ ❸, reducing the length of the child lines by four pixels (approximately 7 percent of the original parent line length). When the stopping condition is met, instead of terminating the `drawTree()` function immediately, the app draws small circles at the end of all final child lines. With a proper selection of colors, the result can resemble a blossoming cherry tree. The grayscale version of this tree is shown in Figure 6-8.

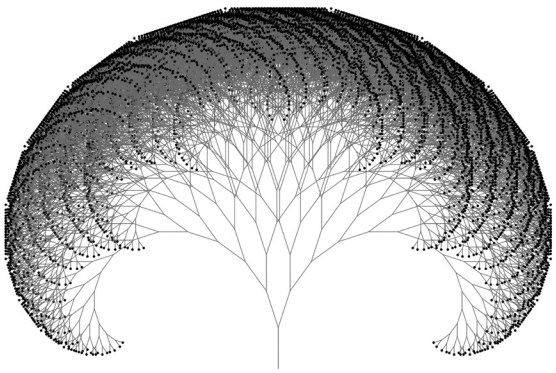


Figure 6-8: A blossoming fractal tree

For all the fractal projects we've worked on so far, and for this project in particular, the parameter values you choose will have a significant effect on the final appearance of the image. If you copy the parameter values verbatim from the book's code snippets, you'll be able to reproduce the exact same figures presented in this chapter. However, there are numerous possible combinations of parameter values you could use. I encourage you to play with the parameters; who knows, you might discover a whole new fractal family that no one has seen yet!

The L-System and Turtle Graphics

Closely related to fractals is the *L-system*, a technique for generating complex strings of characters over a series of iterative steps. The characters are then interpreted as different geometric operations, such as drawing lines or turning left or right. The L-system was introduced by Aristid Lindenmayer, a Hungarian biologist, in 1968 to study the development of

simple organisms and model plant morphology and growth. Lindenmayer proposed that plant development over time can be described by "natural" algorithms that are responsible for the geometric features seen in the arrangements of branches, leaves, petals, and flowers. (We'll discuss several other nature-inspired algorithms in Chapters 8 and 9.) Another feature of plant growth (incorporated in the L-system) that researchers were quick to notice was the prevalence of self-similarity—the same pattern being replicated at different scales. No wonder the L-system was soon picked up by mathematicians and computer scientists to study and visualize the beautiful geometry of nature that Mandelbrot succinctly called fractals.

In this section, we'll study the components of an L-system and develop our own L-system simulator in Kotlin. We'll also learn about Turtle Graphics, a common computer programming model used for visualizing the shapes and patterns created by L-systems. Since Kotlin's standard library doesn't include a Turtle Graphics package, we'll combine JavaFX canvas with Kotlin to create our own `Turtle` class.

Formalizing the L-System

An L-system requires a few ingredients to generate a string of characters interpretable as geometric instructions for drawing fractal objects: an alphabet, an axiom, a set of rules, and a depth. The *alphabet* is the set of characters that are available for use in the string, each with an associated meaning. For example, `F` might signify drawing a line, `+` might signify turning right, and so on (we'll define our full alphabet later). The *axiom* is an initial sequence of characters that the L-system starts from, and the *rules* establish techniques for transforming the axiom (or subsequent strings) over a series of iterations. When there are multiple rules, they're applied in sequence, and the substrings created by each rule are concatenated in the same order to form the new string. The depth sets the number of iterations over which to apply the rules before you arrive at the final string. Table 6-2 summarizes the components of an L-system.

Table 6-2: L-System Components

Feature	Function	Example	Interpretation
Alphabet	A set of allowable characters	`F`, `G`, `J`, `+`, `-`, `[`, `]`, `X`	See Table 6-3.
Axiom	An initial set of characters to start off production (to apply the rules recursively)	`F++F`	Move forward from the current position, turn right by a specified angle twice, and move forward again.
Rules	Directions on how to create a new string from a given axiom or string	`F -> F-F++F-F`	Replace every instance of `F` in a string with the set of characters that follow the arrow (->); add other characters not specified in the rule as is.
Depth	The number of times the rules are applied (axiom is assumed to have a depth of zero)	`5`	Apply the rules five times before creating the corresponding figure.

In addition to the components mentioned in Table 6-2, we need to set two other parameters: the length of the lines to be drawn (or the distance to jump over without drawing a line) and the angle of rotation. Typically, the length decreases with each iteration because the larger (outer) elements of a fractal are often drawn first, followed by finer (inner) elements. Reducing the length may also be required to limit the size of the final figure. The angle of rotation θ for a particular fractal remains constant throughout the process.

Let's look at a simple example to illustrate how to use L-system notations and procedures: drawing a triangle. For this exercise, the L-system parameters and associated steps are outlined in Figure 6-9.

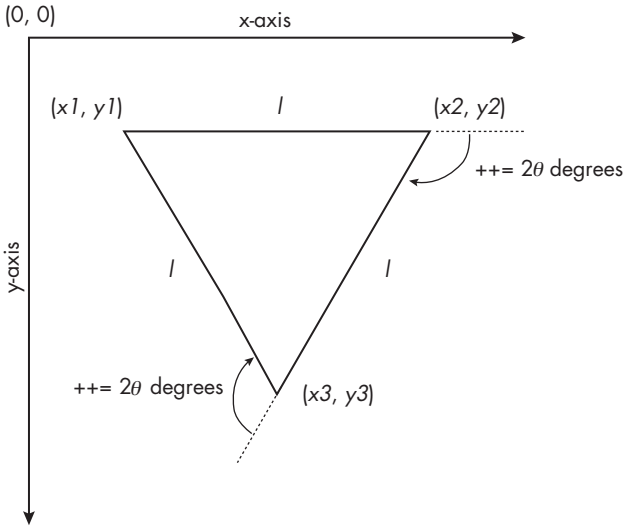


Figure 6-9: The L-system for drawing an equilateral triangle

We start at (*x1*, *y1*) on the canvas, facing in the positive x-axis direction, at angle 0 relative to the x-axis. Our axiom is just a single character, `F`, and our rule, `F -> F++F++F`, tells us that with each iteration we should replace any instance of the character `F` with the string `F++F++F`. After applying the rule for a depth of 1 (meaning apply the rule only once to the axiom before stopping), we have our final string: `F++F++F`.

We now follow this string of instructions: from our initial position, move forward (`F`) a specified length, *l*, to arrive at (*x2*, *y2*), then perform two right turns (`++`), each by a specified angle, θ, and move forward (`F`) again to arrive at (*x3*, *y3*). Finally, take two more right turns (`++`) and move forward (`F`), which brings us back to the starting point and completes the triangle. Each time we move forward from one point to another, we trace the path (connect the two points) with a line.

Drawing L-System Patterns with Turtle Graphics

Turtle Graphics is a drawing system that imagines a turtle moving around a graphics window. The turtle has at least two properties: its location in terms

of x- and y-coordinates and its orientation measured by an angle relative to the positive direction of the x-axis. The turtle drags around a pen. When the pen is down, it draws lines on the window as the turtle moves; when the pen is up, the turtle moves without tracing its path.

Turtle Graphics was originally part of an educational programming language called Logo, but the idea has also been implemented in other languages, including Python. It's used to teach concepts related to programming and the movement and animation of simple objects in a graphics window. It can also be used creatively to teach Euclidian geometry and to generate interesting patterns through recursion. Perhaps the most well-known application of Turtle Graphics in research is for visualizing the strings of instructions created by L-systems of geometric fractals.

When a language has built-in support for Turtle Graphics, we only need to import the relevant package or class and make use of its methods or functions to move the turtle around and draw lines, shapes, and patterns. Since the Kotlin standard library doesn't include a Turtle package, we'll develop a simple `Turtle` class that will work seamlessly with the `Canvas` object of JavaFX and deliver the same functionality. The broader goal of this exercise is to build a minimalist L-system simulator capable of generating L-system strings of arbitrary depth and realizing them with the help of the `Turtle` class to produce well-known geometric fractals. Table 6-3 shows the L-system notations (characters) we'll work with and the corresponding `Turtle` class methods that we'll need to implement.

Table 6-3: L-System Characters and Turtle Graphics Methods

Character(s)	Desired action	Turtle command
`F`, `G`	Move the turtle forward, and draw a line to connect old and new positions. Though `F` and `G` have identical actions, rules may not apply the same way to `F` and `G`.	`turtle.lineTo()`
`J`	Jump forward to a new position without drawing a line.	`turtle.moveTo()`
`+`	Turn the turtle to the right by a specified angle.	`turtle.turnRight()`
`-`	Turn the turtle to the left by a specified angle.	`turtle.turnLeft()`
`[`	Push (save) the turtle's current state (location, angle) to the stack.	`turtle.push()`
`]`	Pull the last saved state from the stack (while also removing it from the stack) and set that as the turtle's current state.	`turtle.pop()`
`X`	Do nothing. Skip to the next instruction.	No turtle call

While drawing complex shapes, the L-system strings may require the turtle to branch out in different directions from a base location. For this, the turtle needs to save its current state so it can later return to the base and choose another direction. The `push()` method helps with saving the current state, while the `pull()` method retrieves a saved state so that the turtle can start a new branch from there. These methods will rely on a *stack*, a

data structure where the most recent item added is the first to be retrieved, commonly called last in, first out (LIFO). This way, the turtle will return to more recent states first, to complete subbranches, before returning to earlier saved states to start drawing new main branches.

Project 25: Design an L-System Simulator

An L-system simulator is made up of several functions or classes that help interpret L-system rules, generate the final string of instructions, and draw the resulting image on a graphics window. Ideally, these components would be coded in a problem-independent manner, with some mechanism for the user to input the initial parameters for the L-system, perhaps through a file or at the command line. For simplicity's sake, we'll instead embed these parameters into the code itself, using values that will generate a snowflake pattern, but the rest of the code will be generic. All you'll have to do is update these problem-specific parameters to simulate different L-system objects.

The Code

We'll begin our L-system simulator code with some global declarations to keep our data organized, then define the problem-specific parameters for generating a particular L-system object. Next, we'll declare some helper functions to generate the L-system string based on rules and draw the final L-system string on the canvas. We'll also declare our `Turtle` class with methods for rendering the image and then tie everything together with an `LSystemApp` class.

Global Declarations

We'll start by declaring two data classes to organize L-system data, plus a special array type to create the stack functionality required for the `push()` and `pull()` methods.

```
// global declarations
data class Rule(val key: String, val apply: String)
data class State(val x: Double, val y: Double, val angle: Double)
val stack = ArrayDeque<State>()
```

The `Rule` data class will define each L-system rule by using the `key` and `apply` members, both of type `String`. The `key` property will hold the character that will be replaced if present when a rule is applied, and the `apply` property will hold the string that will replace the `key`. For example, for a rule `F -> F++F++F`, the values of `key` and `apply` would be `F` and `F++F++F`, respectively.

The second data class, `State`, saves the state of the turtle—specifically, its x- and y-coordinates and the direction it's facing relative to the x-axis. Finally, `stack` is declared by using Kotlin's `ArrayDeque` class, which implements a resizable (mutable) array of the deque (short for *double-ended queue*) data structure. This class has methods to facilitate the LIFO feature of the stack. The array's members will be of our custom `State` data type.

Problem Definition

Next, we'll declare the problem-specific parameters of an L-system, including its axiom and rules. The parameters we're using here will generate a simple fractal pattern that looks like a snowflake. Feel free to replace them with parameters for other well-known L-systems or to experiment with your own.

```
// problem definition
❶ val axiom = "F++F++F"
❷ val rules: List<Rule> = listOf(
    Rule("F", "F-F++F-F"),
    // Rule(),
    // Rule()
    )

var line = 100.0     // in pixels
val scaling = .33    // shrinkage factor per iteration
val ANGLE = 60.0     // turning angle in degrees (fixed)
val depth = 5        // number of iterations

❸ val turtle = Turtle(150.0, 200.0, 0.0)

var finalString = ""
```

First, we define the axiom, or starting string, for a snowflake fractal ❶. Earlier, we used this same string, `F++F++F`, to draw an equilateral triangle. We then create a list of type `Rule` called `rules`, where we keep all the rules pertaining to the L-system ❷. The snowflake requires only a single rule, but the comments inside `listOf()` show how to add more rules if needed. We set the initial value of `line` to 100 pixels based on a canvas size of 600×600, but the line length will shrink by one-third with each iteration. We also set the turning angle, `ANGLE`, to 60 degrees and the number of iterations, `depth`, to 5.

We only need to create a single instance of the `Turtle` class ❸. We set its initial position to (150.0, 200.0), with an initial angle of 0.0 (facing toward the positive x-direction). Notice that as the turtle begins to move, both its location and its orientation (angle) may change. Finally, we set `finalString` to an empty string at the start; its content will change after each iteration, and only the final value will be used to draw the fractal.

The generate() Function

The `generate()` function is a core component of the simulator. It executes the L-system rules for a given axiom and depth.

```
// function to generate final L-system string
fun generate() {
    var nextString = ""
  ❶ for (letter in finalString) {
        var match = false
      ❷ for (rule in rules) {
          ❸ if (letter.toString() == rule.key) {
                match = true
```

```
                nextString += rule.apply
                break
            }
        }
      ❹ if (!match) nextString += letter
    }
  ❺ finalString = nextString
}
```

We first declare an empty string called `nextString`, where we'll save interim values of the final string. We then use two `for` loops to iterate one character at a time through the most recent version of `finalString` ❶ and apply any relevant rules to that character ❷. The Boolean variable `match` tracks if a particular rule should be applied. When a selected character matches the `key` of the current rule ❸, we set `match` to `true` and add the replacement string specified by the `apply` element of the rule to `nextString`. This is followed by the `break` statement to break out of the inner `for` loop (only one rule will be applicable to a character for each iteration of the outer `for` loop). If none of the rules apply, we add the current character to `nextString` unchanged ❹. Once we've iterated over all the characters in `finalString`, we reset its value to `nextString` ❺.

The draw() Function

The `draw()` function draws the final image on the canvas by reading the characters in `finalString` and making corresponding calls to the methods of our `Turtle` class.

```
// function to draw per finalString instructions
fun draw(gc: GraphicsContext) {
  ❶ for (letter in finalString) {
      ❷ when (letter.toString()) {
            "F", "G" -> turtle.lineTo(line, gc)
            "J" ->      turtle.moveTo(line)
            "+" ->      turtle.turnRight(ANGLE)
            "-" ->      turtle.turnLeft(ANGLE)
            "[" ->      turtle.push()
            "]" ->      turtle.pop()
            "X" ->      { /* do nothing */ }
        }
    }
}
```

We elegantly implement the drawing process inside a single `for` loop ❶ and a compact `when` block ❷, which is much less verbose than a traditional multilevel `if...else` block. The `for` loop scans the characters of the variable `finalString` one at a time and passes them to the `when` block as `letter`. Next, we conduct a series of tests until a match is found with one of the possible L-system codes. Based on the match, we call the related `Turtle` method to draw or move on the canvas. For example, we match characters `F` and `G` to the `lineTo()` method and `J` to the `moveTo()` method. Notice that different method calls require different parameters, and the last three calls inside

the `when` block don't require any! We'll see what happens when those methods are called next.

The Turtle Class and Its Methods

Now we'll implement our `Turtle` class, where all the methods for carrying out the final drawing live. To keep the main body of our L-system application short and tidy, we'll declare this class in a separate file called *Turtle.kt*. We begin the file here:

```
import javafx.scene.canvas.GraphicsContext
import kotlin.math.*

❶ class Turtle (var x: Double, var y: Double, angle: Double) {
    var angleRad = angle * PI /180
    --snip--
}
```

The import block gives us access to the required JavaFX and Kotlin library components. The primary class constructor, which is part of the class header ❶, creates an instance of the `Turtle` class using three values: the turtle's starting x- and y-coordinates and its initial orientation (angle of the turtle). We set the first two parameters as `var` because their values will be updated each time the turtle moves or changes its direction. We keep the third parameter `angle` as `val` (not explicitly declared but implied), which is used to define a mutable property, `angleRad`.

The rest of the class body defines the seven different methods that can be called to perform different `Turtle` object tasks. See Table 6-3 for a summary of these methods and their corresponding L-system characters.

```
fun lineTo(line: Double, gc: GraphicsContext) {
    val xBegin = x
    val yBegin = y
    x += line * cos(angleRad)
    y += line * sin(angleRad)
    gc.strokeLine(xBegin, yBegin, x, y)
}

fun moveTo(line: Double) {
    x += line * cos(angleRad)
    y += line * sin(angleRad)
}

fun turnRight(delta: Double) {
    // origin @ bottom left
    angleRad += delta * PI /180
}

fun turnLeft(delta: Double) {
    // origin @ bottom left
    angleRad -= delta * PI /180
}
```

```
fun push() {
    stack.addLast(State(x, y, angleRad))
}

fun pop() {
    val (xPop, yPop, anglePop) = stack.removeLast()
    x = xPop
    y = yPop
    angleRad = anglePop
}

fun printTurtle() {
    print("x: ${round(x * 100) / 100.0}  y: ${round(y * 100) / 100.0}  ")
    println("angle: ${round((angleRad * 180 / PI) * 100) / 100.0} degrees")
}
```

The `lineTo()` method is the only one that actually draws anything on the canvas. It draws a straight line of length `line`, based on the same sine and cosine calculations we used to draw segments of the fractal tree in Project 24. All the other methods either move the turtle without drawing a line, change its direction, or save or restore its state. The final method simply prints the current state of the turtle, which is useful for debugging purposes.

The `push()` and `pop()` methods rely on methods of the `ArrayDeque` class, which we used to implement our stack. The `push()` method calls the `addLast()` function, while `pop()` calls the `removeLast()` function to add or remove an item to the end of the `stack` array, enforcing the LIFO rule. In both cases, we encapsulate the state information by using our custom `State` data class.

The LSystemApp Class

Finally, we'll bring the different components of the L-system simulator together to form a complete application, organized around an `LSystemApp` class. The following listing shows the full body of this class, as well as the `main()` function. The listing also shows where all the other components we discussed earlier fit (except for the `Turtle` class, which is in a separate file).

```
// import statements
--snip--

// global declarations
--snip--

// problem definition
--snip--

// function to generate final L-system string
fun generate() {
--snip--
}

// function to draw per finalString instructions
fun draw(gc: GraphicsContext) {
```

```
--snip--
}

// JavaFX-Kotlin Application class
class LSystemApp : Application() {
    override fun start(stage: Stage) {
        val canvas = Canvas(600.0, 600.0)

        val gc = canvas.graphicsContext2D
        // Move the origin to bottom left.
    ❶ gc.translate(0.0, canvas.height)
        // Let positive y-axis point up.
    ❷ gc.scale(1.0, -1.0)

        val pane = Pane()
        pane.children.add(canvas)
        val scene = Scene(pane, 600.0, 600.0)
        stage.title = "L-system Simulator"
        stage.scene = scene
        stage.show()

        // ---L-system-related code---
        finalString = axiom

    ❸ if (depth > 0) {
            for (i in 1..depth) {
                generate()
            }
          ❹ line *= (scaling).pow(depth - 1.0)
        }
        draw(gc)
    }
}

fun main() {
    Application.launch(LSystemApp::class.java)
}
```

Other than the `generate()` and `draw()` functions, much of the `LSystemApp` consists of the boilerplate JavaFX code. What's different this time is the addition of a couple of extra lines that relocate the origin of the canvas to the bottom-left corner ❶ and let the positive direction of the y-axis point upward ❷. These simple changes make testing previously published L-systems that provide axioms, rules, and initial conditions for many well-known fractals very convenient. Published L-system parameters are almost always based on the assumption that we draw the resulting fractals in the first quadrant of the coordinate system.

The code managing the L-system revolves around an `if` block that checks if `depth` is greater than `0` ❸. If not, we simply draw the axiom itself, without applying any rules. Otherwise, we call the `generate()` function `depth` times to apply the rules over the appropriate number of iterations. We then use the final content of `finalString` to draw the fractal. Notice that we set the length of the lines to be drawn based on `depth` ❹. This dynamic

adjustment allows us to keep the size of the fractal limited to the size of the canvas. If you prefer to use a larger or smaller canvas size (we're using 600×600), you may have to adjust the initial line length, as well as the initial position of the turtle.

The Result

We're now ready to put our brand-new L-system simulator to the test. Since we've already included problem-specific parameters for a snowflake fractal, we can run the simulator with different depths to see how the image gets more complex and manifests self-similar patterns as the depth increases. Figure 6-10 shows the evolution of the snowflake for depths 1, 3, and 5.

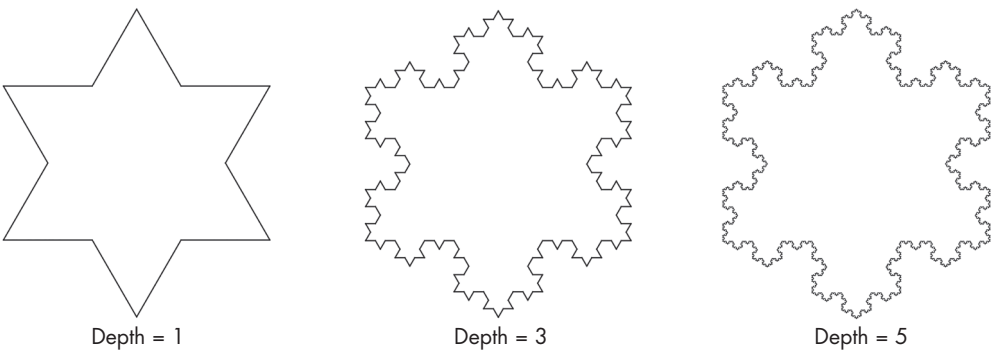


Figure 6-10: The evolution of a snowflake, using the L-system simulator

Researchers have developed so many interesting L-systems over the years that you could spend days playing with those systems in your simulator and looking at the intriguing patterns they generate. For example, Table 6-4 lists the L-system parameters necessary for creating three more fractals with different features. Some of these incorporate a wider range of symbols, and they all involve multiple rules.

Table 6-4: Additional L-System Examples

	Fractal name		
Property	**Sierpiński triangle**	**Pattern with rectangles**	**Fractal plant**
Axiom	`F+G+G`	`F-F-F-F`	`X`
Rules	`F -> F+G-F-G+F` `G -> GG`	`F -> F-J+FF-F-FF-FJ-FF+J-FF+F+FF+FJ+FFF` `J -> JJJJJJ`	`X -> F+[[X]-X]-F[-FX]+X` `F -> FF`
Line	+150.0	+15.0	+20.0
Scaling	0.5	0.5	0.7
Angle	120	90	25
Depth	6	2	6
Turtle	150.0, 200.0, 0.0	150.0, 450.0, 0.0	100.0, 50.0, 65.0
Canvas	600.0×600.0	600.0×600.0	600.0×600.0

The first set of parameters in Table 6-4 will create the familiar Sierpiński triangle. The second set, with a depth of only two, is for a geometric pattern that involves drawing multiple unconnected rectangles (see Figure 6-11, left). The third set is for a fractal plant, a popular object among fractal enthusiasts (see Figure 6-11, right). It involves a number of push and pull operations ([and]) to keep track of the plant's various branches.

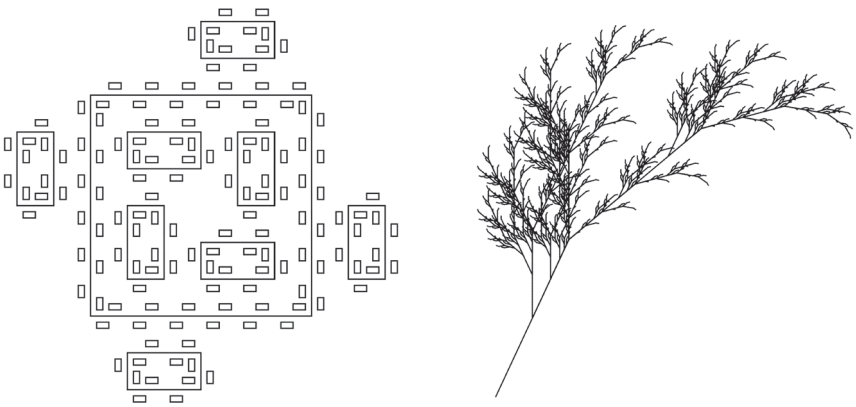


Figure 6-11: A rectangular pattern (left) and a fractal plant (right) generated by the L-system simulator

You can extend the basic L-system simulator we've developed in this section to generate 3D fractals, such as a space-filling Hilbert curve. Check out the resources listed at the end of this chapter to learn more about those advanced implementations. The primary benefit of the L-system simulator, however, is that it allows you to experiment with your own axioms and rules. Perhaps you'll come up with a brand-new fractal that no one has seen before and name it after yourself!

The Mighty Mandelbrot Set

No discussion of fractals would be complete without examining the Mandelbrot set, or M-set, and appreciating its complexity and beauty. The M-set differs from the other fractals we've discussed so far in two important ways. First, it's a nonlinear fractal, meaning its building blocks are made up of pixels organized in complex patterns instead of only straight lines. Second, it's mapped on a complex plane, so it requires us to use complex numbers.

The M-set hinges on the iterative properties of the deceptively simple quadratic function, shown in Equation 6.1.

$$f(z) = z^2 + c \tag{6.1}$$

Here c is a constant. By "iterative properties," I mean the way the value of $f(z)$ changes if we start with some initial seed value for z and then

recursively feed the value of the equation back into itself as the new z value. Equation 6.2 shows the recursive form of this function.

$$z_n = z^2_{n-1} + c \tag{6.2}$$

The list of numbers generated by repeated iterations of Equation 6.2, given a particular seed value z_0 and constant c, is known as the *orbit* of the function. Table 6-5 shows the orbits for a few different (z_0, c) combinations.

Table 6-5: Selected Orbits of Quadratic Function $f(z) = z^2 + c$

n **(iteration)**	**Case 1: $z_0 = 0$, $c = 1$**	**Case 2: $z_0 = 0$, $c = -1$**	**Case 3: $z_0 = 0$, $c = -0.65$**
1	1	–1	–0.65
2	2	0	–0.22749999
3	5	–1	–0.59824375
4	26	0	–0.29210441
5	677	–1	–0.56467501
6	458,330	0	–0.33114213
7	Very large number!	–1	–0.54034488

Each case in Table 6-5 uses a seed value of 0 but a different constant. Notice how minor variations in the value of the constant can set the orbit off on a completely different path. In general, the orbit will either diverge, meaning the function values will become increasingly large due to exponential growth (as in case 1), or remain bounded, such that the orbit values stay within a certain range. If the latter, several variations are possible:

- The orbit cycles through a fixed set of values (as in case 2, where the cycle has a period of 2).
- The orbit converges toward a fixed value; in case 3, the function value converges to –0.4486 after approximately 100 iterations.
- The orbit remains fixed on a single value (for example, when $z_0 = 0$ and $c = 0$).
- The orbit values remain bounded but behave chaotically, showing no apparent pattern.

All this hidden complexity is governed by the constant c as it takes on different values. So far, however, we've used only real numbers as values of c. It's time to introduce complex numbers into the mix—values of c that satisfy this equation:

$$c = x + iy \tag{6.3}$$

Here x and y are real numbers, and i is $\sqrt{-1}$. Table 6-6 shows a few examples of our quadratic function using complex values of c.

Table 6-6: Selected Orbits of Quadratic Function $f(z) = z^2 + c$, Where $c = x + iy$

***n* (iteration)**	**Case 1: $z_0 = 0$, $c = 0 + i$**	**Case 2: $z_0 = 0$, $c = 0 + 2i$**
1	i	$2i$
2	$-1 + i$	$-4 + 2i$
3	$-i$	$12 - 14i$
4	$-1 + i$	$-52 - 334i$
5	$-i$	$-108{,}852 + 34{,}738i$
6	$-1 + i$	Far away from (0, 0)
7	$-i$	Very far away from (0, 0)
M-set membership	Member	Nonmember

As with real values of c, complex values also produce two types of orbits: bounded (like case 1) or unbounded (like case 2). Given this, we're now ready to define the Mandelbrot set as the set of all complex numbers c for which the corresponding orbit generated by recursive function $f(z) = z^2 + c$ remains bounded, given a seed of $z_0 = 0$. Thus, looking at Table 6-6, $c = 0 + i$ is a member of the Mandelbrot set, while $c = 0 + 2i$ is not.

We visualize the Mandelbrot set by plotting the set's members on a *complex plane*, a coordinate system where the x-axis represents the real component of a complex number (x in Equation 6.3) and the y-axis represents the imaginary component (y in Equation 6.3). Given these x- and y-values, the magnitude of the orbit of a complex number is calculated as:

$$|a| = \sqrt{x^2 + y^2} \tag{6.4}$$

When a complex number is plotted as a point on the complex plane, the magnitude of the orbit would be the distance of that point from the origin.

Project 26: Code and Visualize the Mandelbrot Set

Let's turn our attention to creating an app that will help us visualize the M-set. Our definition of the M-set provides some clues about what's required: we need to find and plot the complex values of c that cause our quadratic function (Equation 6.1) to remain bounded as it iterates. However, we still need a few clarifications. First, what region of the complex plane should we investigate? It would help to know if the M-set members are clustered in a certain region or dispersed over a large area. Also, since the quadratic function can continue iterating ad infinitum, it isn't clear when exactly we should conclude that an orbit is bounded or unbounded. We could limit ourselves to a certain number of iterations, consult the magnitude of the orbit as defined by Equation 6.4, or both.

Fortunately, researchers have already uncovered helpful facts about the M-set that we can draw on to make our search efficient and practical. First, from numerous plots of the M-set already created, we know that the search space can be limited to an area bounded by [–2.0, 1.0] along the real

axis (x-axis) and [–1.5, 1.5] along the imaginary axis (y-axis). Second, it's also known that the M-set is a closed set entirely contained inside a circle of radius 2 around the origin. This means that a complex number c can't be a member of the set if $|z_n| > 2$ for any $n > 0$. We'll use this as a termination condition for the bounded/unbounded test. Otherwise, if we complete some threshold number of iterations (`iterMax`) without $|z_n|$ exceeding 2, we'll consider c to be a member of the M-set.

We'll set `iterMax` to `400` for a relatively clean image of the M-set, but I encourage you to experiment by setting `iterMax` lower or higher. In general, lower `iterMax` values will likely show more noise as well as various zones of convergence. As the number of iterations increases (the test condition becomes more stringent), the fractal boundary tends to become crisper.

The Code

Despite the sophisticated nature of the concept, the code for creating the M-set is relatively minimal. The app we'll create will have the following features and functionalities:

- The problem definition (done through a small number of global variables)
- The ability to iterate over the search space point by point
- The ability to check M-set membership conditions for each of those points
- The ability to draw the point on the canvas with a chosen color scheme

In addition, we'll need to adjust the scale of the image to ensure that the search space is properly mapped onto a canvas of a given size.

Declaring Global Variables

First, we'll declare some global variables. This code segment defines the bounds for the real and imaginary axes, sets the canvas size, determines the search increment along the x- and y-axes, and limits the number of iterations for the quadratic function.

```
// problem definition and global parameters
val xMin = -2.0
val xMax = 1.0
val yMin = -1.5
val yMax = 1.5
val xRange = xMax - xMin
val yRange = yMax - yMin

val canvasW = 600.0
val canvasH = (canvasW / xRange) * yRange
val increment = 0.003
val iterMax = 400
```

We set `xMin` and `xMax` as the lower and upper bounds for the real values of *c*; similarly, `yMin` and `yMax` are the lower and upper bounds for the imaginary part of *c*. The corresponding ranges (`xRange` and `yRange`) define a rectangular search space we'll explore to find potential M-set members.

Next, we set the width of the canvas, `canvasW`, to 600 pixels and the height of the canvas, `canvasH`, to a value that will maintain the proportionality of the image (meaning that the M-set, when plotted, won't look distorted). The benefit of this approach is that you only need to adjust `canvasW` if you want to create a larger or smaller image.

We set the last two parameters, `increment` and `iterMax`, to `0.003` and `400`, respectively. These values will determine the overall image quality. It's possible to link the `increment` parameter to `canvasW` and have it automatically calculated—you're welcome to try that out as a simple experiment.

Finding and Drawing M-Set Members

We'll now declare a function to iterate over the search space, check whether a particular complex number *c* is within the M-set, and mark the corresponding points on the canvas by using a color scheme.

```
// function to iterate over the search space and draw
// nonmembers using a grayscale and members as black points

private fun drawMSet(gc: GraphicsContext) {
    var y = yMin
  ❶ while (y <= yMax) {
        var x = xMin
      ❷ while (x <= xMax) {
          ❸ val cval = getConvergence(x, y)
          ❹ val speed = cval.toDouble() / iterMax
            val factor = 1.0 - speed
            gc.fill = Color.color(factor, factor, factor)
          ❺ gc.fillRect(canvasW * (x - xMin)/ xRange,
                canvasH * (y - yMin)/ yRange, 1.0, 1.0)
            x += increment
        }
        y += increment
    }
}
```

The `drawMSet()` function uses a nested pair of `while` loops to iterate over the search space. The outer loop ❶ iterates along the y-axis, starting with the minimum `y` value, `yMin`, and ending when `y` reaches the upper bound, `yMax`, incrementing `y` by `increment` (which we set to `0.003`) each time. The inner `while` loop ❷ does the same along the x-axis. With each iteration, we get an (*x, y*) pair representing a complex number (as defined by Equation 6.3), which we pass to the `getConvergence()` function ❸ to check if that number belongs to the M-set (we'll look at that function next).

Calling `getConvergence()` returns the number of iterations of the quadratic function it carried out. We divide this by `iterMax` to measure how

quickly the convergence decision was made ❹. If this resulting value (`speed`) is `1`, the number of iterations must have been `iterMax`, indicating that the orbit of *c* remained bounded. Therefore, we'll count this particular complex number as a member of the M-set. If `speed` is less than `1`, however, that would mean the orbit jumped outside the circle of radius 2 before reaching the maximum number of iterations, so we won't consider that number to be an M-set member.

We subtract `speed` from `1` and use the result (`factor`) to set the color of the corresponding point on the canvas, according to a grayscale scheme where (1, 1, 1) means white and (0, 0, 0) means black. If the orbit diverged very quickly, `factor` will be close to `1`, so the point on the canvas representing this number will be marked with a white or nearly white pixel. Conversely, if the orbit diverged only after many more iterations (but before reaching the maximum allowed), `factor` will be closer to `0`, so the corresponding point would be marked with a darker pixel. Of course, if the orbit remains bounded, `factor` will be exactly `0`, and the corresponding point will be marked with a pure black pixel, signifying that the point belongs to the M-set.

Finally, we plot the number by mapping its (*x*, *y*) coordinate pair onto a pixel location on the canvas ❺, using the scaling factor illustrated in Figure 6-12.

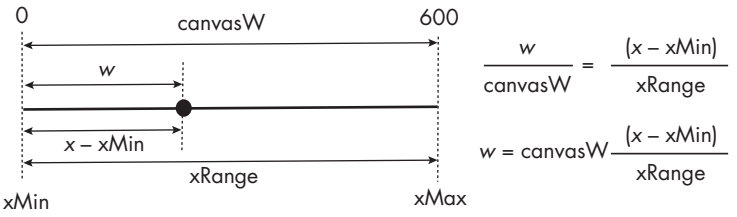


Figure 6-12: Mapping the x-coordinate of `c` *to the x-value of the pixel on the canvas*

We're using the proportionality rule to map the x-coordinate to a point on the canvas designated as *w*. We apply the same principle to the y-coordinate as well.

Checking for Convergence

The `getConvergence()` function takes in an (*x*, *y*) pair as arguments and checks if the corresponding complex number belongs to the M-set.

```
// function to check for membership in the M-set
private fun getConvergence(x: Double, y: Double): Int {
    var zx = 0.0
    var zy = 0.0
  ❶ for (i in 1..iterMax) {
        val X = zx * zx - zy * zy + x
        val Y = 2 * zx * zy + y
      ❷ if (X * X + Y * Y > 4.0) return i
        zx = X
```

```
        zy = Y
    }
    return iterMax
}
```

To follow the logic of the code, we need a bit of algebra. Recall from Equation 6.2 that the recursive form of our quadratic function is $z_n = z^2_{n-1} + c$. For any value of $n > 1$, z_{n-1} will be a complex number that can be expressed as:

$$z_{n-1} = zx_{n-1} + (i\, zy_{n-1}) \tag{6.5}$$

In Equation 6.5, zx_{n-1} is the real part of z_{n-1}, and zy_{n-1} is the complex part. We can now rewrite Equation 6.2 by using Equations 6.3 and 6.5. After simplification, we get:

$$z_n = (zx^2_{n-1} - zy^2_{n-1} + x) + i\,(2zx_{n-1}\, zy_{n-1} + y) \tag{6.6}$$

Equation 6.6 can be further simplified as:

$$z_n = X + iY \tag{6.7}$$

where $X = (zx^2_{n-1} - zy^2_{n-1} + x)$ and $Y = (2zx_{n-1}\, zy_{n-1} + y)$.

Thinking back to Equation 6.4, we check if the magnitude of z_n lies outside a circle of radius 2 by confirming whether:

$$|z_n| = \sqrt{X^2 + Y^2} > 2, \text{ or } X^2 + Y^2 > 4 \tag{6.8}$$

Returning to our code, we start by setting real and imaginary components `zx` and `zy` to `0`, the equivalent of setting seed value z_0 to 0. Next, the `for` loop ❶ iterates over Equation 6.7, calculating the values of `X` and `Y`. With each iteration, we check if the magnitude of z_n exceeds `2` ❷, per Equation 6.8. If so, the complex number c represented by the (x, y) pair isn't a member of the M-set, and we return the number of iterations it took to reach that conclusion. Otherwise, the iteration continues until we reach `iterMax`. If z_n remains bounded the whole time, we return `iterMax` to indicate that the number is a member of the M-set.

Bringing Everything Together

Let's take a look at how all these code segments fit together.

```
// import statements
--snip--

// problem definition and global declarations
--snip--

// function to iterate over the search space and draw
// nonmembers using a grayscale and members as black points
fun drawMSet(gc: GraphicsContext) {
--snip--
}
```

```
// function to check for membership in the M-set
fun getConvergence(x: Double, y: Double): Int {
--snip--
}

// Application class for drawing the M-set
class Mandelbrot : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val canvas = Canvas(canvasW, canvasH)
        val gc = canvas.graphicsContext2D
        root.children.add(canvas)

        val scene = Scene(root, canvasW, canvasH)
        scene.fill = Color.WHITE
        stage.title = "Mandelbrot Set"
        stage.scene = scene
        stage.show()

        // Search for M-set members and draw them on the canvas.
        drawMSet(gc)
    }
}

// the main function to launch the application
fun main() {
    Application.launch(Mandelbrot::class.java)
}
```

Other than our global declarations and our two function definitions, the only other problem-specific code is a single call to `drawMSet()` to create and draw a fractal. It's remarkable that despite the M-set's complexity, only a few lines of code are required to generate the fractal.

The Result

Let's use our app to explore the Mandelbrot set, which remains one of the most enigmatic mathematical objects ever discovered. Given the parameter values we used in "Declaring Global Variables" on page 255, Figure 6-13 shows the core M-set (the dark region) and some of its features.

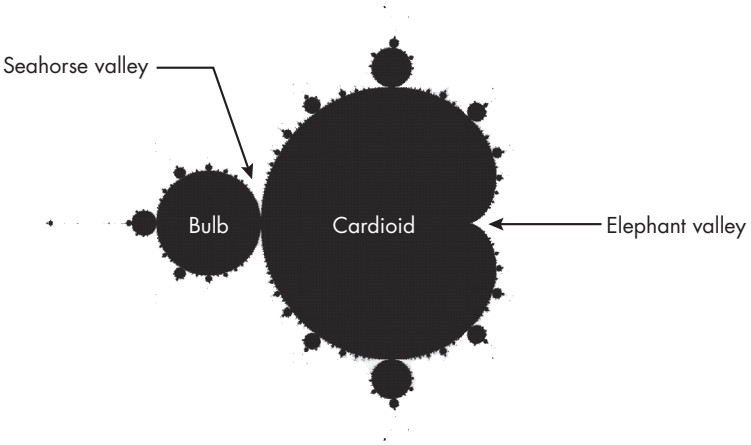


Figure 6-13: The Mandelbrot set

The central, heart-shaped part of the M-set is called the *cardioid*. The circular part to the left of the cardioid is called the main *bulb*. It's a circle centered at (–1.0, 0.0), with a radius of 0.25. Numerous other bulbs are attached to the main bulb and the cardioid all around their boundaries, and those bulbs seem to have antennas or tentacles. When we zoom in on them, we notice intricate patterns and even smaller cardioid-like features with their own bulbs and tentacles. In other words, some features of the M-set are replicated again and again, no matter how small the scale is (even though the replication is not exact)—a defining feature of any fractal.

The finer patterns, visible after sufficient magnification, aren't simple geometric shapes; instead, they're beautifully intricate and detailed in nature. Figure 6-14 shows magnified versions of two specific regions of interest, the *seahorse valley* and the *elephant valley*, as identified in Figure 6-13.

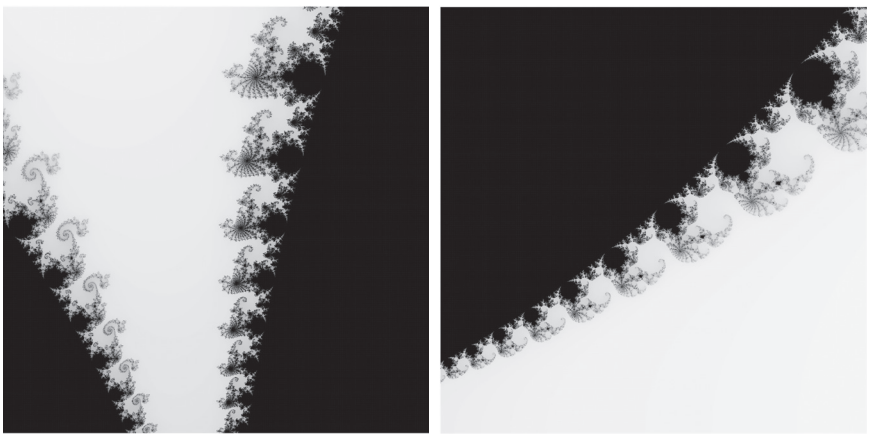


Figure 6-14: The seahorse valley (left) and the elephant valley (right)

You can generate both of these figures by changing the search region and using appropriate parameter values in your app, as summarized in Table 6-7.

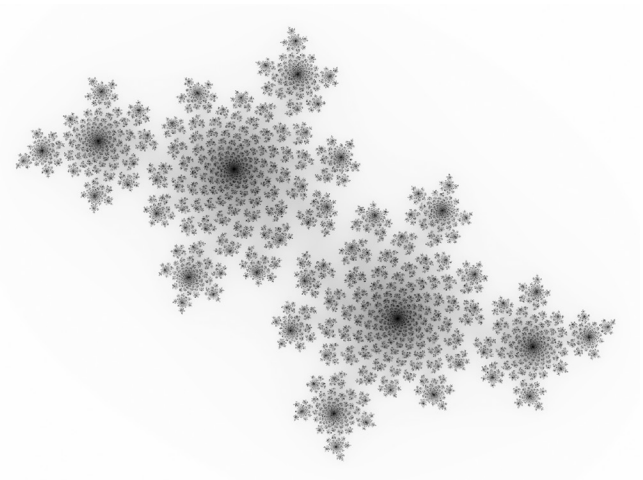
Table 6-7: Search Region and Parameter Values for Select Regions of the M-set

Parameter	Seahorse valley	Elephant valley
xMin	-0.8	0.275
xMax	-0.7	0.325
yMin	-0.2	-0.05
yMax	-0.1	0.0
canvasW	600	600
increment	0.0001	0.00005
iterMax	200	200

Enjoy exploring the M-set by using these parameters, or come up with your own parameter values and see what other features you can find.

EXERCISE

Modify the code used for drawing the M-set to draw a Julia set, shown here:



Julia sets are drawn using the same equation as the M-set (Equation 6.2), but instead of using (*x*, *y*) pairs to set the value of *c*, you use them to set the initial value of *z*. That is, $z_0 = x + iy$ for a given value of *c*. For this exercise, use $c = -0.4 + (i \times 0.6)$. Set canvas to `800.0`, `increment` to `0.0025`, and `iterMax`

(continued)

to 200. The search space boundaries along the x- and y-axes should be (–2, 2) and (–1,5, 1.5), respectively.

Hint: You'll need to change the `getConvergence()` function. Additionally, to ensure that the plot is created in the first quadrant of the coordinate system, you'll need to add two lines of code to handle the graphics (as we did for the L-system simulator):

```
gc.translate(0.0, canvas.height)
gc.scale(1.0, -1.0)
```

Summary

In this chapter, you used Kotlin to explore the enigmatic beauty of fractals. You learned how to design recursive functions to help draw simple geometric fractals, and you developed an L-system simulator to generate intricate self-similar patterns based on strings of instructions and a few transformation rules. Finally, you created an application for visualizing the famous Mandelbrot set.

No matter what the mathematical properties of a fractal are, you now have the tools to put together a few lines of code in Kotlin and JavaFX to render it. We've barely scratched the surface of fractal geometry, however. In particular, you have a lot to learn about the M-set that's beyond the scope of this book. If this chapter has aroused your interest in fractals, I encourage you to check out the listed resources for further reading.

Resources

Devaney, Robert L. *An Introduction to Chaotic Dynamical Systems.* 3rd ed. Boca Raton, FL: CRC Press, 2022.

Feldman, David P. *Chaos and Fractals: An Elementary Introduction.* Oxford, UK: Oxford University Press, 2012.

Flake, Gary William. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation.* Cambridge, MA: MIT Press, 2000.

Mandelbrot, Benoit B. *The Fractal Geometry of Nature.* San Francisco: W. H. Freeman & Co., 1982.

Ponce Campuzano, Juan Carlos. "The Mandelbrot Set." *Complex Analysis.* 2019. Accessed June 15, 2024. *https://complex-analysis.com/content/mandelbrot_set.html.*

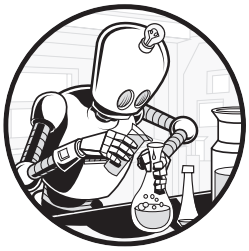
Prusinkiewicz, Przemyslaw, and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Electronic version, 2004. Accessed June 15, 2024. *http://algorithmicbotany.org/papers/abop/abop.pdf*.

Prusinkiewicz, Przemyslaw, Aristid Lindenmayer, and F. David Fracchia. "Synthesis of Space-Filling Curves on the Square Grid." In *Fractals in the Fundamental and Applied Sciences*, edited by Heinz-Otto Peitgen, José Marques Henriques, and Luís Filipe Penedo, 334–366. North-Holland: Elsevier, 1991.

Weisstein, Eric W. "Mandelbrot Set." Wolfram MathWorld. Accessed June 15, 2024. *https://mathworld.wolfram.com/MandelbrotSet.html*.

7

SORTING AND SEARCHING



One of the fundamental skills that any serious developer needs to learn is how to efficiently sort and search through a given dataset. This skill is invaluable for transforming raw data into actionable insights, whether you're working with a simple array or with complex data structures spanning terabytes of multifield information extracted from the vast expanse of the internet.

Sorting and searching are a dynamic duo that work hand in hand. *Sorting* organizes data into a specific order, which enables meaningful analysis of the dataset as a whole. Once the data is sorted, it becomes easier to identify patterns, trends, and outliers. Sorting also improves the speed and ease of *searching* for desired items within the dataset, especially when working with large volumes of data. Indeed, many search algorithms, such as binary search, interpolation search, and tree-based searches, rely on the organization achieved through sorting. Searching further complements

sorting by enabling targeted analysis, allowing for the quick location of specific data points or subsets within the dataset. Together, sorting and searching streamline data exploration, optimize retrieval efficiency, and empower decision-making processes.

A wide array of sorting and search algorithms are available. In this chapter's projects, we'll focus on a selected group of algorithms that have broad applications in fields that require working with large datasets. By mastering these algorithms, you'll be better equipped to tackle complex data challenges and make the most of sorting and searching capabilities.

Sorting Algorithms

Sorting algorithms allow us to rearrange a collection of data elements into a specific order, such as numerical or alphabetical or based on any other desired criteria. We can sort various types of data, including numbers, strings, records (lines of data in a database), and complex objects. Sorting is a fundamental building block for a variety of operations, such as merging, joining, and aggregating datasets. It paves the way for efficient data manipulation, which is crucial in domains like database management, algorithms, and programming. By organizing data structures, sorting provides a structured framework that promotes clarity, consistency, and ease of use. This streamlined approach enhances data management and maintenance, particularly in scenarios where data must be updated or modified frequently.

Each sorting algorithm has its own advantages and disadvantages in terms of time complexity, space complexity, and stability. Before we get into specific algorithms, it's important to review these concepts, as they'll assist us in selecting the appropriate algorithm for a given problem.

Time complexity refers to the estimation of the algorithm's running time based on the input size, which is denoted by n. It provides insight into how the algorithm's performance scales with larger datasets. Common notations like $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, and $O(2^n)$ indicate different growth rates of time complexity in increasing order. The smaller the growth rate, the quicker the algorithm is in sorting a collection of data.

BIG O NOTATION

Big O notation is a mathematical notation that describes how a function behaves when its argument approaches a specific value or tends toward infinity. In computer science, it's used to describe the performance or complexity of an algorithm.

In the context of sorting algorithms, big O notation indicates how the time or space complexity of an algorithm grows as the size of the input array grows. For example, the merge sort algorithm has a time complexity of $O(n \log n)$.

For an array size of 1,000, its time complexity would be $O(1{,}000 \log 1{,}000)$, or approximately 9,966 (using a base of 2 for the logarithm). If the array size doubles to 2,000, the time complexity increases to $O(2{,}000 \log 2{,}000)$, which is approximately 21,932. This means that doubling the array size would, on average, increase the time needed to sort the array with the merge sort algorithm by a factor of $21{,}932/9{,}966 \approx 2.2$.

By contrast, the insertion sort algorithm has an average time complexity of $O(n^2)$. Increasing the input array size from 1,000 to 2,000 therefore means going from a time complexity of $1{,}000^2 = 1$ million to one of $2{,}000^2 = 4$ million. In other words, doubling the size of the dataset quadruples the time complexity.

In practice, the actual runtime for merge sort, insertion sort, or any other algorithm will depend on many other factors, such as the implementation details, the hardware and software environment, and the specific input data.

Space complexity, on the other hand, describes the amount of additional memory an algorithm requires to perform the sorting operation, beyond the memory it needs to store the data being sorted. Some algorithms may operate with minimal extra space, where the swapping of data elements happens *in place*. Others may require significant auxiliary memory to perform sorting operations efficiently. This is also called *out-of-place* sorting, where full or partial copies of the original dataset are needed to carry out the sorting operation. The smaller the space complexity, the more efficient (and scalable) that algorithm is in terms of memory requirements.

Stability is another important consideration. A sorting algorithm is stable if it maintains the relative order of elements with equal values. In certain situations, preserving the initial order of equal elements is required, and a stable algorithm becomes essential.

Table 7-1 shows these properties for a selected group of sorting algorithms that we'll discuss in this chapter.

Table 7-1: Key Features of Selected Sorting Algorithms

	Time complexity			Space	
Algorithm	**Best**	**Average**	**Worst**	**complexity**	**Stability**
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Stable
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Stable
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$*	Unstable
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Unstable

*The worst case can be $O(n)$.

Of the sorting algorithms listed in Table 7-1, insertion sort is the simplest and most intuitive, but it isn't the most efficient in terms of average time complexity. It tends to be slower than the other algorithms for larger

datasets. Due to this limitation, insertion sort generally isn't used as a standalone algorithm, but rather as part of a hybrid sorting scheme that combines multiple methods, depending on the characteristics of the data.

Both merge sort and heap sort have similar time complexities, typically $O(n \log n)$. However, heap sort has an advantage in terms of space complexity because it's an in-place algorithm, meaning it requires minimal additional memory beyond the input array. On the other hand, merge sort requires additional space proportional to the input size. If stability is a desired property, then merge sort becomes the preferred choice over heap sort. It's a stable sorting algorithm, ensuring that elements with equal values maintain their original order.

In practice, quick sort often performs better than other sorting algorithms, except when the data is already sorted or nearly sorted. Quick sort benefits from lower space complexity, and it has smaller overhead, or fewer hidden operations that don't depend on the size of the input data. Many programming language libraries provide built-in functions for quick sort, making it easily accessible and widely used.

Project 27: Space-Efficient Sorting with Insertion Sort

Insertion sort is a simple and intuitive sorting algorithm that works by building a sorted array one element at a time. The algorithm maintains a sorted subarray within the given array and extends it by inserting elements from the unsorted part of the array into the correct position in the sorted part. At the beginning, the first element of the array is considered to be a sorted subarray of size 1. The algorithm then iterates through the remaining elements, one at a time, and inserts each element into its appropriate position within the sorted subarray.

To insert an element, the algorithm compares it with the elements in the sorted subarray from right to left. It shifts any larger elements one position to the right until it finds the correct position for the current element. Once the correct position is found, the element is inserted into that position. This process continues until all the elements in the array are processed, resulting in a fully sorted array.

Say we have the unsorted array [8, 3, 4, 5, 1, 2]. Here's how the insertion sort algorithm would process it:

1. Imagine that the given array is made up of two subarrays—a sorted array, which initially holds only the first element (8), and an unsorted array made up of the remaining elements.
2. Compare the second element of the array (index 1) with its preceding element (index 0) as follows:
   a. Compare 3 with 8. Since 3 is smaller, swap the elements.
   b. The array after the first pass is [3, 8, 4, 5, 1, 2].

3. Move to the next element (index 2) and compare it with the previous elements.
    a. Compare 4 with 8. Since 4 is smaller, swap the elements.
    b. Compare 4 with 3. Since 4 is greater, stop comparing.
    c. The array after the second pass is [3, 4, 8, 5, 1, 2].
4. Repeat this process for the remaining elements of the array. In the end, the array will be sorted in ascending order: [1, 2, 3, 4, 5, 8].

As indicated in Table 7-1, insertion sort has an average and worst-case time complexity of $O(n^2)$, where n is the number of elements in the array. However, it performs well for small lists or nearly sorted lists. It's an in-place sorting algorithm with space complexity of $O(1)$ for all cases, meaning it doesn't require additional memory to perform the sorting.

The Code

Implementing the insertion sort algorithm in Kotlin takes only a few lines of code. We'll create a dedicated function called `insertionSort()` to handle all the necessary steps for sorting an array and call this function from `main()` to get the job done.

```
fun main() {
    // Define an array to be sorted.
    val arr = intArrayOf(8, 3, 4, 5, 1, 2)

    println("\n*** Sorting an Array Using Insertion Sort ***\n")
    println("original array:\n${arr.contentToString()}")
    // Call the insertion sort function.
    insertionSort(arr)
    println("sorted array:\n${arr.contentToString()}")
}

fun insertionSort(A: IntArray) {
    // Sorting happens in place.
  ❶ for (i in 1 until A.size) {
        val key = A[i]
        var j = i
      ❷ while( j > 0 && A[j-1] > key ) {
            A[j] = A[j-1]
            j -= 1
        }
      ❸ A[j] = key
    }
}
```

This code snippet implements the insertion sort algorithm to sort an array of numbers (in this case, integers) in ascending order. We create an array called `arr` that holds the initial unsorted array elements. The content of this array is printed to the console, allowing us to see the original order of the numbers. We then call the `insertionSort()` function to perform the

sorting operation. It takes the array as input and modifies it in place, so you don't have to return the sorted array to the calling function.

Within the `insertionSort()` function, we iterate through the unsorted portion of the array by using a `for` loop ❶, starting from the second element (index `1`) and continuing to the last element. For each element, we temporarily store the value in a variable called `key`. Next, we use a `while` loop ❷ to move from right to left through the sorted portion of the array, comparing `key` with each element. The `while` loop continues as long as two conditions are met: more elements remain to the left (checked using `j > 0`), and the current element is greater than `key` (checked using `A[j-1] > key`). Inside the `while` loop, if an element is greater than `key`, it's shifted one position to the right. This makes space for `key` to be inserted at the correct sorted position.

When the `while` loop ends, we assign the value of `key` to the current position in the array ❸, effectively inserting the element into the sorted portion of the array at the correct position. The `for` loop then moves to the next element, and the process repeats for all the elements in the array. Once the sorting is complete, the code prints the sorted array to the console, displaying the numbers in ascending order.

The Result

If you run the code without changing the given unsorted array, the output should look like this:

```
*** Sorting an Array Using Insertion Sort ***

original array:
[8, 3, 4, 5, 1, 2]
sorted array:
[1, 2, 3, 4, 5, 8]
```

The code can easily be tweaked to sort floating-point numbers by assembling an array of either `Float` or `Double` data types. I encourage you to modify the code to accept user input regarding the preferred sorting order—either ascending or descending. After that, you can implement a suitable function based on the user's choice. Alternatively, you can use the same function with two subfunctions, which can be implemented using `when(choice)`, one for sorting an array in ascending order and one for doing the opposite.

Project 28: Faster Sorting with Merge Sort

Merge sort is a popular sorting algorithm that follows a divide-and-conquer approach. It works by recursively dividing an array into smaller subarrays until each subarray contains only one element. The subarrays are then merged back into longer arrays, placing the elements in the correct order in the process, eventually resulting in a fully sorted array. Figure 7-1 illustrates this process for the same [8, 3, 4, 5, 1, 2] array we used in Project 27.

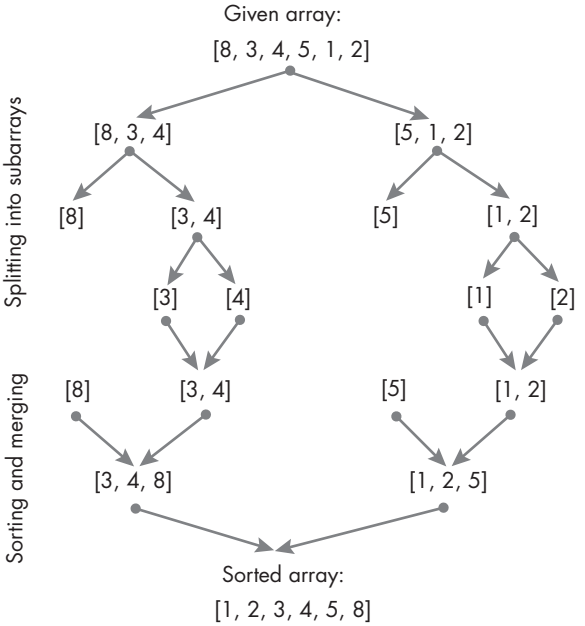


Figure 7-1: Visualizing the merge sort algorithm

Notice how the given array is initially divided into two subarrays, and then notice how each of those subarrays is further divided into two subarrays, and so on. The subarrays are then sorted and merged. When we implement the algorithm by using a recursive function, we'll first process entirely the left subarray of the first pair of subarrays—in this example, [8, 3, 4]—before moving on to the right subarray, [5, 1, 2]. Within each branch, we'll divide the subarrays into individual elements and then reassemble the sorted subarrays. Eventually, the final pair of sorted left and right subarrays will be merged to generate the final sorted array.

Merge sort guarantees a consistent time complexity of $O(n \log n)$ in all cases, making it efficient for large datasets. It's also a stable sorting algorithm, preserving the relative order of equal elements. However, it needs additional space for the merging step, making its space complexity $O(n)$. Nonetheless, merge sort's efficiency and stability make it a popular choice for sorting in various applications.

The Code

We'll follow a similar structure for the merge sort code as we did for the insertion sort: a `main()` function that kicks off the sorting process by passing an array to a `mergeSort()` function. This time, however, `mergeSort()` will recursively call itself until it reaches a stopping condition (when each subarray has only one element). To put everything back together, we'll use a helper function called `merge()`, which handles the task of sorting and merging the subarrays.

```
fun main() {
    val arr = intArrayOf(8, 3, 4, 5, 1, 2)

    println("\n*** Sorting an Array Using Merge Sort ***\n")
    println("original array:\n${arr.contentToString()}")
    // Call the recursive function.
  ❶ mergeSort(arr)
    println("\nsorted array:\n${arr.contentToString()}")
}

fun mergeSort(arr: IntArray) {
    val length = arr.size
    if (length < 2) return // done splitting subarrays

  ❷ val middle = length / 2
    val leftArray = arr.copyOfRange(0, middle)
    val rightArray = arr.copyOfRange(middle, length)

  ❸ mergeSort(leftArray)
    mergeSort(rightArray)
    merge(leftArray, rightArray, arr)
}

fun merge(leftArray: IntArray, rightArray: IntArray,
          arr: IntArray) {

    val leftSize = leftArray.size
    val rightSize = rightArray.size
    var i = 0   // for original array
    var l = 0   // for left array
    var r = 0   // for right array

    // Compare, sort, and merge.
  ❹ while(l < leftSize && r < rightSize) {
        if (leftArray[l] < rightArray[r]) {
            arr[i] = leftArray[l]
            l++
        } else {
            arr[i] = rightArray[r]
            r++
        }
      ❺ i++
    }

    // If all elements of a subarray are assigned, assign the
    // remaining elements of the nonempty array to "arr".
    while (l < leftSize) {
        arr[i] = leftArray[l]
        l++
        i++
    }

    while (r < rightSize) {
        arr[i] = rightArray[r]
        r++
```

```
        i++
    }
}
```

In the `main()` function, we begin by initializing an array called `arr` with a set of integer values. We also print the given array before proceeding so that we'll be able to compare this with the sorted array once it's generated. We then call the `mergeSort()` function ❶, which is responsible for carrying out the sorting process. This function takes an array `arr` as an argument.

Within `mergeSort()`, we first check the length of the incoming array. If it's less than `2`, the subarray has a length of `1`, so the function simply returns, and the splitting process stops. This is the all-important stopping condition that any recursive function needs. Next, we calculate the `middle` index of the array ❷ and create two subarrays: `leftArray` and `rightArray`. The former contains elements from index `0` up to but not including `middle`, while the latter contains elements from `middle` to the end of the array. To continue the process, the `mergeSort()` function recursively calls itself on both `leftArray` and `rightArray` ❸. As mentioned, this recursive step continues until the base case is reached—that is, when the length of the subarrays becomes `1`. Finally, we call `merge()` to reassemble `leftArray` and `rightArray` into a single, sorted array.

The `merge()` function accepts three parameters, `leftArray`, `rightArray`, and `arr`, representing the two subarrays to be merged and the original array that will be modified during the merging process. We start the function by initializing variables to keep track of the indices within the arrays; `i` is for traversing the original `arr`, `l` for the `leftArray`, and `r` for the `rightArray`. The actual merging and sorting occur within a `while` loop ❹ that continues as long as elements remain in both `leftArray` and `rightArray` to compare. During each iteration, the function compares the values at indices `l` and `r` in `leftArray` and `rightArray`, respectively. If the value in `leftArray` is smaller, it's assigned to `arr` at index `i`, and the `l` index is incremented. Conversely, if the value in `rightArray` is smaller, it's assigned to `arr` at index `i`, and the `r` index is incremented. Following each assignment, the `i` index is also incremented ❺.

The `while` loop concludes when either `leftArray` or `rightArray` has been fully processed. The remaining elements from the nonempty array are then assigned to `arr` to complete the merging process. We use two separate `while` loops for this task—one for `leftArray` and one for `rightArray`. Only one of these loops will actually execute.

The Result

When you run the merge sort code for the given input array, it should produce the following output:

```
*** Sorting an Array Using Merge Sort ***

original array:
[8, 3, 4, 5, 1, 2]

sorted array:
[1, 2, 3, 4, 5, 8]
```

I encourage you to repeat the same exercise you did with insertion sort, allowing the user to choose the order of sorting (ascending or descending) and then modifying the code to sort accordingly. I also recommend that you think about arrays containing both positive and negative numbers. You might soon realize that by selectively multiplying the entire array by –1 before and after sorting, you can use the same code to sort an array in ascending or descending order instead of writing two separate functions!

Project 29: High-Efficiency Sorting with Quick Sort

Quick sort is a well-known and highly efficient in-place sorting algorithm that's widely used in various real-world applications. It involves selecting a pivot element from the array and dividing the remaining elements into two subarrays, one for values less than the pivot and the other for values greater than the pivot. This mechanism places the pivot element itself in the correct position in the final sorted array, while the remaining elements end up on the appropriate side of the pivot. The process repeats recursively for the subarrays, selecting new pivot elements and further portioning the array, until everything is sorted.

Here's a step-by-step example of how quick sort works, using the array [8, 3, 4, 5, 1, 2]:

1. Choose a pivot element, which can be any element from the array. In this example, we'll choose the last element, 2.
2. Partition the array into two subarrays, the left subarray with elements less than the pivot and the right subarray with elements greater than the pivot. In this case, the left subarray becomes [1], and the right becomes [4, 5, 8, 3]. I'll explain where this order comes from later in the project.
3. Recursively apply quick sort to the subarrays. For the left subarray, no further action is needed: it has only one element, so it's already in its final position. For the right subarray, we now pick 3 as the pivot. This creates an empty left subarray as 3 is the smallest number. The right subarray now has [5, 8, 4].
4. Repeat step 3 until all subarrays are sorted, meaning each subarray has only one element or is empty.
5. Combine the sorted subarrays to get the final sorted array: [1, 2, 3, 4, 5, 8].

In this example, we always chose the last element of the array or subarray as the pivot element. Another option could have been to choose the first element as the pivot. For a wide range of inputs, choosing the first or last element as the pivot will work well, especially if the input data is randomly or uniformly distributed. However, if the array is already sorted or nearly sorted, pivoting around the first or last element will yield the

worst-case time complexity of $O(n^2)$. To avoid this, you can use one of the following alternative techniques for choosing a pivot:

Choose a random element

Randomly selecting a pivot element helps mitigate the inefficiency of choosing the first or the last element when the array is already mostly sorted. This approach can provide a good average-case performance since the pivot's position is less predictable. It reduces the likelihood of encountering worst-case scenarios, resulting in better overall efficiency.

Choose the median of three

This strategy involves using the median value among the first, middle, and last elements of the array as the pivot. This approach aims to balance the pivot selection by choosing a value closer to the true median of the dataset. It helps improve the algorithm's performance on a wide range of inputs, reducing the chance of worst-case behavior.

Compared to other sorting algorithms, quick sort is highly efficient for large datasets, and its average and worst-case time complexity are $O(n \log n)$ and $O(n^2)$, respectively. Quick sort has an average space complexity of $O(\log n)$, which can degenerate to $O(n)$ when the input array is already sorted or nearly sorted and the first or the last element is chosen as the pivot (worst case).

The Code

The code for quick sort is quite similar to that of merge sort in structure, as both algorithms rely on a divide-and-conquer approach. In the code, the `main()` function accepts an input array and passes it to the `quickSort()` function. Within `quickSort()`, we invoke a `partition()` helper function to determine the correct position for the pivot element. This allows us to divide the original array into a left array containing elements less than the pivot and a right array containing elements greater than or equal to the pivot. Finally, `quickSort()` is recursively called on these subarrays as long as `start` is less than `end`, which means at least two elements remain in the subarray.

```
fun main() {
    val arr = intArrayOf(8, 3, 4, 5, 1, 2)

    println("\n*** Sorting an Array Using Quick Sort ***\n")
    println("original array:\n${arr.contentToString()}")
    // Call the recursive function.
  ❶ quickSort(arr, start = 0, end = arr.size -1)
    println("\nsorted array:\n${arr.contentToString()}")
}

fun quickSort(arr: IntArray, start: Int, end: Int) {
    // Check that the termination condition for recursion
    // base case is when start = end.
  ❷ if (start < end) {
```

```
        val pivotIndex = partition(arr, start, end)
        quickSort(arr = arr, start = start, end = pivotIndex - 1)
        quickSort(arr = arr, start = pivotIndex + 1, end = end)
    }
}

fun partition(arr: IntArray, start: Int, end: Int): Int {
    val pivot = arr[end]
    var i = start

    for (j in start until end) {
      ❸ if (arr[j] < pivot) {
            swap(arr, i, j)
            i++
        }
    }
  ❹ swap(arr, i, end)
    return i
}

fun swap(arr: IntArray, i: Int, j: Int) {
    val temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

In the `main()` function, we call the `quickSort()` function by passing three parameter values: the array to be sorted (`arr`) and the indices for its first and last elements (`start` and `end`) ❶. As before, we print the array before and after sorting.

In the `quickSort()` function, we start by checking whether the starting index of the incoming subarray is less than the ending index ❷. When this is no longer true, the subarray will have only one element, so the recursion of that branch will stop. Otherwise, we call the `partition()` helper function, which returns the final (sorted) position of the pivot element. We store this position as `pivotIndex` and use it to divide the original array into left and right subarrays. We then recursively call `quicksort()` on the left and right subarrays until the stopping condition is met.

The real sorting work happens inside the `partition()` function. After setting `pivot` to the value of the last element in the subarray, we use two index variables, `i` and `j`, to swap the positions of the elements inside a `for` loop. Both start at the beginning of the subarray, and then `j` steps through the subarray looking for elements with values less than `pivot` ❸. Each time one is found, the values at `i` and `j` are swapped, and then `i` is incremented. In effect, this moves elements less than the pivot to earlier in the array, and elements greater than the pivot to later in the array. Once the `for` loop is done, the pivot itself is swapped with the element at `i` ❹, which puts the pivot element into its final sorted position. Then the final value of `i` is returned so that two new subarrays can be formed on both sides of the final position of the last pivot element. The swaps themselves are relegated to a

swap() helper function, which uses the temp variable to avoid overwriting the value at i. Apart from this one extra variable, the sorting happens in place.

The Result

If you run the code with the example array, the output should look like this:

```
*** Sorting an Array Using Quick Sort ***

original array:
[8, 3, 4, 5, 1, 2]

sorted array:
[1, 2, 3, 4, 5, 8]
```

I mentioned earlier that I would explain how the order of the subarray elements is determined. This has to do with the swapping algorithm in the partition() function. During the first round of processing the [8, 3, 4, 5, 1, 2] array, for example, 2 is the pivot, and the first element in the array less than the pivot is 1. This element gets swapped with the 8 at the start of the array (accessed using index variable i), yielding an array of [1, 3, 4, 5, 8, 2]. Then the pivot itself (2) is swapped with the next element of the array (3—again accessed via i), yielding [1, 2, 4, 5, 8, 3].

I encourage you to manually step through the entire process of sorting the array with quick sort. You can refer to Figure 7-2, which shows the original input array, the intermediate subarrays after each round of processing, and the final sorted array. By going through the comparisons and swaps yourself, you can visualize the partitioning and sorting process in a more tangible way.

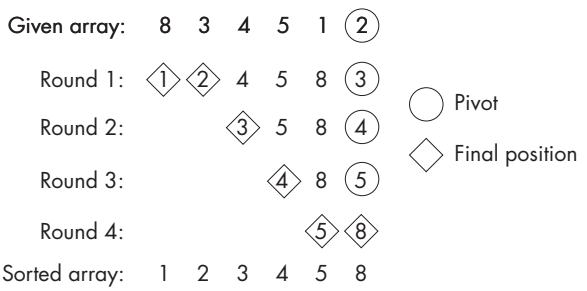


Figure 7-2: The quick sort steps for [8, 3, 4, 5, 1, 2]

You can also autogenerate the subarrays at each stage by printing the left and right arrays from inside the quicksort() function, just after the position of the pivot is determined.

EXERCISE

Heap sort is an algorithm that sorts an array in ascending or descending order by first converting it into a *binary heap*, a structure that organizes data in a tree-like fashion. A binary heap has two main properties:

- It's a complete binary tree: every node in the tree has at most two child nodes, and all levels of the tree are fully filled, except possibly the last level, where the nodes are as far to the left as possible.
- It satisfies either the *max heap* or the *min heap* property. For max heap, the value of each node must be greater than or equal to the values of its children. For min heap, the value of each node must be less than or equal to the values of its children.

Heap sort involves building a heap from the input array, repeatedly extracting the maximum (or minimum) element from the heap and placing it in the sorted portion of the array. Heap sort has a time complexity of $O(n \log n)$ on average and in the worst case. And as an in-place sorting algorithm, it has a space complexity of $O(1)$. Although it isn't stable, heap sort is used for its efficiency and minimal memory requirements.

Do some research to learn more about the heap sort algorithm. Then write a Kotlin program to implement the algorithm.

Search Algorithms

Searching through a data structure is a fundamental operation in computer science. It helps us track down specific elements or retrieve information stored within a collection of data. While this task may seem trivial for a small amount of data, as the volume of data increases—up to large databases, filesystems, or even the whole internet—knowing how to choose the right search algorithm becomes paramount to keeping our digital life humming.

Search algorithms are intimately connected to the data structures they're designed to search, since how the data is organized affects how efficiently a particular item can be found and accessed. For the purposes of the coming projects, we'll focus on several algorithms that are used to search a graph, which is a type of data structure. Before we get to the algorithms themselves, however, it's important to establish how graphs are structured.

What Is a Graph?

In the field of graph theory, a graph is a mathematical structure consisting of a set of vertices (also known as *nodes*) and a set of edges (also known as *arcs* or *links*) that connect pairs of vertices. Vertices can represent any kind

of objects, such as cities, people, or even more abstract concepts. Edges represent relationships or connections between the vertices. Mathematically, a graph is denoted by G and defined as $G = (\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is a set of vertices or nodes, and $\boldsymbol{E}$ is a set of edges or links.

Figure 7-3 depicts a simple graph consisting of five nodes and five edges. Each circle in the figure represents a vertex, and each line represents an edge. The nodes are named with sequential numbers for convenience. In real-world cases, most nodes would be names with strings, however. When node names are designated by whole numbers, we can treat them as either of type `Int` or of type `String` in the code.

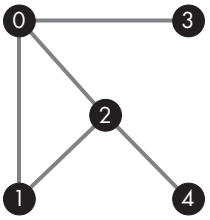


Figure 7-3: A simple graph with five nodes and five edges

Graphs can be categorized into two main groups: undirected and directed. In an *undirected graph*, the edges allow movement between vertices in both directions. This type of graph is often used to represent scenarios like a road network, where traffic can flow both ways. By contrast, each edge in a *directed graph* has a specific direction associated with it, restricting the way you can move between vertices. For example, a directed graph can represent a water or power distribution network, where the flow always moves from areas of high pressure to areas of low pressure or from high voltage to low voltage, respectively.

When the edges of a graph have weights associated with them, the graph is called a *weighted graph*. The weight in this case could be a proxy for cost, distance, or any other edge-related property. Weighted graphs can be either directed or undirected.

How to Search a Graph

In the coming projects, we'll consider three different algorithms for searching a graph. The first, *depth-first search (DFS)*, is a technique that starts at a particular node and explores as far (or "as deep") as possible along one branch before backtracking and exploring the next. In this way, it traverses the depth of a data structure before exploring its breadth. DFS is often implemented by using a *stack* data structure (we explored stacks in Chapter 6 while developing the L-system simulator). This way, DFS can use the youngest node in the stack to extend the branch and explore each adjacent node at the end of a branch before backtracking and moving to the next branch. DFS is useful in many applications, including scheduling

problems, detecting cycles in graphs, and solving puzzles with only one solution, such as a maze or a sudoku puzzle.

The next algorithm, *breadth-first search (BFS)*, takes the opposite approach of DFS, exploring the data structure level by level. It starts at a given node and visits all its immediate neighbors. Then it moves on to the next level, visiting all the neighbors' neighbors, and so on. In this way, BFS prioritizes exploring the breadth of the entire data structure over the depth of any individual branch. As we'll discuss in Project 31, BFS typically uses a queue data structure, allowing it to visit each level in order. It's useful for finding the shortest path, web crawling, analyzing social networks, and exploring all reachable nodes in a graph while using the smallest number of iterations.

The choice between DFS and BFS depends on the specific problem and the characteristics of the data structure being searched. DFS is typically used when we want to conduct a deep exploration and potentially find a target item more quickly, while BFS is suitable for situations where we want to visit all nodes at a certain distance from the starting point or find the shortest path between nodes.

The final algorithm we'll explore is called *A* search* (pronounced "A-star search"). It excels in finding the shortest path in a graph or a maze by combining heuristic decision-making with real-time exploration to guide the search. The term *heuristic* refers to general decision-making strategies that rely on intuition, educated guesses, or common sense to arrive at a plausible solution or direction to explore. While heuristics can't guarantee an optimal or perfect outcome, they often provide an advantage in situations where constraints such as limited information, time, or resources exist.

The A* algorithm's heuristic is to consider both the cost of reaching a specific node and an estimate of the remaining effort required to reach the destination. In this way, A* is able to intelligently prioritize the most promising paths. This strategic approach, similar to having a GPS in a labyrinth, helps save time and effort in the search process. Due to its versatility, A* is frequently applied in fields such as pathfinding in video games, robotics, navigation systems, and various optimization problems.

Project 30: Stack-Based Searching with Depth-First Search

In this project, we'll explore the core steps of depth-first search and implement them in Kotlin. We'll employ the stack data structure in the code, although it's worth noting the existence of other viable methods for implementing the core DFS algorithm. Later on, I'll share some hints on an alternative approach.

For a given graph (a network of nodes and edges), here are the steps to perform a DFS by using a stack:

1. Start by selecting a node as the starting node (it can be any node).
2. Push the starting node onto the stack.
3. While the stack is not empty, pop a node from the stack.

4. If the popped node is not yet visited, mark it as visited and push its neighbors to the stack; or else pop the next node from the stack.
5. Repeat steps 3 and 4 until the stack is empty.

Recall from Chapter 6 that a stack follows the LIFO principle, whereby items are removed from the stack in the reverse order in which they were added. The LIFO principle allows the DFS algorithm to backtrack from the end of one branch before starting on a new, unvisited branch. This ensures an exhaustive search of the entire graph, although it would also be beneficial to include a stopping condition. When each node is visited, this condition would check if the desired goal of the search has been achieved, such as finding a specific object or completing a particular task. Once the goal is met, the search can be terminated early. For this project, we'll use the graph shown in Figure 7-3.

The time complexity of the DFS algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. The space complexity of DFS depends on the implementation (a stack versus a recursive function); the worst-case space complexity is $O(V)$.

The Code

Let's now examine the code that implements the core steps of DFS. We'll use the code to traverse the entire example graph shown earlier in Figure 7-3.

```
import java.util.ArrayDeque

fun main() {
  ❶ val graph = mapOf(
        "0" to setOf("1", "2", "3"),
        "1" to setOf("0", "2"),
        "2" to setOf("0", "1", "4"),
        "3" to setOf("0"),
        "4" to setOf("2")
    )
    println("\n*** Depth-First Search of a Graph ***\n")
    println("Graph to search:")
    for ((key,value) in graph)
        println("Node: $key,  Neighbors: $value")

  ❷ val visited = dfsStack(graph, "0")
    println("\nVisited nodes:\n$visited")
}

fun dfsStack(graph: Map<String, Set<String>>, start: String):
             Set<String> {

    val visited = mutableSetOf<String>()
    val stack = ArrayDeque<String>()
    stack.push(start)

  ❸ while (stack.isNotEmpty()) {
        val node = stack.pop()
```

```
            if (node !in visited) {
                // Do something as needed.
                visited.add(node)
              ❹ for (next in graph[node]!!) {
                      stack.push(next)
                }
            }
        }
        return visited
}
```

First, we import the `ArrayDeque` class from `java.util`, which we'll use to implement the stack. Next, we declare the `main()` function, which serves as the entry point of the program. It defines the graph as a `map` pairing each node (`"0"` through `"4"`) with a set of all its neighboring nodes ❶. For example, node `"2"` is paired with the set `["0", "1", "4"]`, since it's connected to those nodes. We print the graph to the console, then call the `dfsStack()` function to perform the search, passing the graph and a starting node as arguments ❷. Upon completion of the search, the list of visited nodes is returned, which is printed as the program's final output.

Inside the `dfsStack()` function, we create a mutable set called `visited` to keep track of the visited nodes and an `ArrayDeque` named `stack` to store the nodes during traversal. We push the starting node to the stack, then enter a `while` loop that iterates for as long as the stack is not empty ❸. In each iteration, the last node from the stack is removed by using `pop()` and assigned to the variable `node`. If the node hasn't been visited before, we could perform additional operations or processing specific to the application at this point—for example, checking if the node matches our search criteria and breaking from the loop if it does. The node is then added to the `visited` set by using the `add()` function.

Next, we add all neighboring nodes, retrieved from `graph` by using `node` as the key, to the stack via the `push()` function ❹. We use the nonnull assertion operator (`!!`) while adding `graph[node]` to the stack to avoid additional null safety checks that aren't required for undirected graphs (every node will have at least one link or edge). The `while` loop terminates once the stack is empty, at which point the set of visited nodes is returned to `main()`.

Note that we could have used the `ArrayDeque` class from `kotlin.collections` (as we did in Chapter 6) instead of `ArrayDeque` from `java.util` to implement the stack. In that case, we would replace `push()` with `addLast()` and `pop()` with the `removeLast()` function. I've chosen to use the Java version in part to illustrate an alternative stack implementation and in part because the `ArrayDeque` method names like `push()` and `pop()` fit naturally with the stack architecture. Both techniques follow the LIFO principle, meaning that the last element added to the stack is the first one removed.

The Result

If you run the code with the given graph, you should get the following output:

```
*** Depth-First Search of a Graph ***

Graph to search:
Node: 0,  Neighbors: [1, 2, 3]
Node: 1,  Neighbors: [0, 2]
Node: 2,  Neighbors: [0, 1, 4]
Node: 3,  Neighbors: [0]
Node: 4,  Neighbors: [2]

Visited nodes:
[0, 3, 2, 4, 1]
```

The list of visited nodes `[0, 3, 2, 4, 1]` indicates the algorithm has traversed the entire graph. To see where this order comes from, and to better understand how the stack facilitates the DFS process, consider Table 7-2, which shows the intermediate values at each step of the algorithm.

Table 7-2: Anatomy of the Depth-First Search Using Stack

Stage	Node	Node not visited?	Visited nodes	Neighbor nodes	Nodes on stack
Initialization, with `start` of 0	N/A	N/A	[] (empty)	N/A	[0] (`start` pushed to stack)
Inside the `while` loop	0	true	[0]	[1, 2, 3]	[1, 2, 3]
	3	true	[0, 3]	[0]	[1, 2, 0]
	0	false	no change	N/A	[1, 2]
	2	true	[0, 3, 2]	[0, 1, 4]	[1, 0, 1, 4]
	4	true	[0, 3, 2, 4]	[2]	[1, 0, 1, 2]
	2	false	no change	N/A	[1, 0, 1]
	1	true	[0, 3, 2, 4, 1]	[0, 2]	[1, 0, 0, 2]
	2	false	no change	N/A	[1, 0, 0]
	0	false	no change	N/A	[1, 0]
	0	false	no change	N/A	[1]
	1	false	no change	N/A	[] (empty; `while` loop terminates)

Let's take a look at a few rows from Table 7-2 to understand how DFS works. In the first row, we see what happens during the initialization phase, before entering the `while` loop. We set the starting node to `"0"` and push it onto the stack. At this stage, node `"0"` hasn't been marked as visited yet. Next, we move inside the `while` loop, where the rest of the processing happens. First, we pop the last node from the stack, which is `"0"` (this makes the stack momentarily empty). Since this node isn't yet marked as visited, we add it to the list of visited nodes, which goes from `[]` to `[0]`. We then add all this node's neighbors (accessed with `graph["0"]`) to the stack, which goes from `[]` to `[1, 2, 3]`.

The next time through the loop, `"3"` is popped from the stack, since it's the last element. It hasn't been visited yet, so it's added to `visited`, and its only neighbor `"0"` is pushed to the stack. The process continues until the stack is found to be empty at the start of a `while` loop iteration. I strongly encourage you to go over the remaining rows of the table to get a hands-on feel for how the DFS algorithm works in practice.

EXERCISE

In Project 30, we used a stack to search a graph. This process can be simplified by using recursion, which repeats a few key steps until a stopping condition is met. A recursive function can make the DFS process shorter and clearer. To implement DFS by using recursion, do the following:

1. Write a recursive function for DFS called `dfsRecursion()` that replaces the `dfsStack()` function. The function would have the three parameters: `graph`, `start`, and `visited` (as a mutable list). The recursive part of the function could look like this:

   ```
   for (next in (graph[start]!! - visited)) {
       dfsRecursion(graph, next, visited)
   }
   ```

   The recursion will stop when no more unvisited nodes that are connected to the current node remain. Notice that `graph[start]!! - visited` is a set operation, not a regular subtraction operation.
2. In Project 30, we defined the graph as a `Map` object. We could also define it by using instances of Kotlin's `Pair` data class, one pair for each node and its neighbors. Make this change.
3. Use integers instead of strings to denote each node in the graph.
4. Run the code after each change and compare your results with the original output from the stack version of the code.

Project 31: Queue-Based Searching with Breadth-First Search

In this project we'll continue our exploration of search algorithms by implementing breadth-first search. BFS guarantees that all nodes at the same level are visited before moving on to the next level. This process continues until all nodes in the graph have been visited. As in Project 30, we'll use the `ArrayDeque` class from `java.util` to implement the BFS algorithm. This time, however, we'll use the class as a *queue*, a data structure that adheres to the *first in, first out (FIFO)* principle. Whereas items are always added ("pushed") or removed ("popped") from the end of a stack, items are added

("enqueued") at the end of a queue and removed ("dequeued") from the beginning of the queue. This ensures that items are processed in the order in which they were added to the queue.

To perform a BFS, we'll follow these steps:

1. Select a node as the starting node (it can be any node).
2. Create a mutable list called `visited` and add the starting node to it.
3. Create an empty queue and enqueue (add) the starting node.
4. While the queue is not empty, perform the following steps:
    a. Dequeue the front node from the queue.
    b. Process the dequeued node as needed (perhaps printing its value or performing some operation).
    c. Enqueue all the unvisited neighbors of the dequeued node and mark them as visited.

We'll use the graph shown in Figure 7-3 for this project as well.

The time complexity of the BFS algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. The space complexity of the BFS algorithm is typically $O(V)$. Both DFS and BFS therefore have the same time complexity, but their space complexity can vary depending on the implementation and the structure of the graph.

The Code

The code for BFS closely resembles that of DFS, but I'll highlight a few important distinctions as we discuss the program.

```
import java.util.ArrayDeque

fun main() {
    // Define the graph to be searched.
    val graph = mapOf(
        "0" to setOf("1", "2", "3"),
        "1" to setOf("0", "2"),
        "2" to setOf("0", "1", "4"),
        "3" to setOf("0"),
        "4" to setOf("2")
    )

    println("\n*** Breadth-First Search of a Graph ***\n")
    println("Graph to search:")
    for ((key,value) in graph)
        println("Node: $key,  Neighbors: $value")

  ❶ val visited = bfsQueue(graph, "0")
    println("\nVisited nodes:\n$visited")
}

fun bfsQueue(graph: Map<String, Set<String>>, start: String): Set<String> {
    val visited = mutableSetOf<String>()
```

```
    visited.add(start)
    val queue = ArrayDeque<String>()
    queue.offer(start)

  ❷ while (queue.isNotEmpty()) {
        val node = queue.poll()
        for (next in graph[node]!!) {
          ❸ if (next !in visited) {
                queue.offer(next)
                visited.add(next)
            }
        }
    }
    return visited
}
```

The `main()` function is essentially the same as that of the previous project. We define the input graph by using a `map` data structure and print the graph, displaying each node and its neighbors. We then call the `bfsQueue()` search function, passing the graph and the starting node as arguments ❶. The function returns the visited nodes, which are printed as the final output of the program.

Inside the `bfsQueue()` function, we initialize a mutable list called `visited` to keep track of visited nodes as before, along with an `ArrayDeque` called `queue` to store the nodes to be visited. We then add the starting node to both the visited set and the queue, using the `offer()` method for the latter. Next, we initiate a `while` loop that continues until the queue becomes empty ❷. Within the loop, we dequeue a node from the front of the queue by using the `poll()` method, placing it in the `node` variable. We then iterate over each neighbor of the current node, obtained from the graph. If a neighbor hasn't been visited (meaning it isn't present in the `visited` set), it's enqueued by using the `offer()` method and added to the `visited` set ❸. After processing all the neighbors, the loop continues until the queue becomes empty. The `visited` set is then returned, containing all the nodes visited during the search.

The Result

For the given graph, if you run the code without any changes, the code will produce the following output:

```
*** Breadth-First Search of a Graph ***

Graph to search:
Node: 0,  Neighbors: [1, 2, 3]
Node: 1,  Neighbors: [0, 2]
Node: 2,  Neighbors: [0, 1, 4]
Node: 3,  Neighbors: [0]
Node: 4,  Neighbors: [2]

Visited nodes:
[0, 1, 2, 3, 4]
```

Again, the list of visited nodes `[0, 1, 2, 3, 4]` indicates the algorithm has successfully traversed the entire graph. This time, the nodes are marked as visited in numerical order, a function of the FIFO principle of the queue. Table 7-3 shows the intermediate values of the key variables as the process unfolds and how the BFS algorithm works.

Table 7-3: Anatomy of the Breadth-First Search Using a Queue

Stage	Node	Neighbor nodes	`next` node	Node not visited?	Nodes on queue	Visited nodes
Initialization, with `start` of 0	N/A	N/A	N/A	N/A	[0]	[0]
Inside the `while` loop	0	[1, 2, 3]	1	true	[1]	[0, 1]
			2	true	[1, 2]	[0, 1, 2]
			3	true	[1, 2, 3]	[0, 1, 2, 3]
	1	[0, 2]	0	false	[2, 3]	no change
			2	false	[2, 3]	no change
	2	[0, 1, 4]	0	false	[3]	no change
			1	false	[3]	no change
			4	true	[3, 4]	[0, 1, 2, 3, 4]
	3	[0]	0	false	[4]	no change
	4	[2]	2	false	[] (empty; `while` loop terminates)	no change

Let's go over a few of the rows in Table 7-3 to gain a better understanding of how the BFS algorithm is implemented. At the initialization stage, we identify node `"0"` as the start node and add it to both the `visited` list and the queue. Both of these lists now contain `"0"` (see the first row).

Next, we move inside the `while` loop, which runs as long as `queue` is not empty. We start with the front node `"0"` and fetch its neighboring nodes, `"1"`, `"2"`, and `"3"`. For each, we check that it hasn't been visited before; when this is true, we add that node to both `queue` and `visited`. Since none of these nodes were visited, they're all added to `queue` and `visited` when we're done with node `"0"`.

The process continues by pulling the next front node, `"1"`. This time both its neighbors, `"0"` and `"2"`, show up in the visited list, so nothing is added to `queue` or `visited`. Each time we remove a node from `queue`, the queue shrinks in size. In the final step, node `"4"` is pulled out, making `queue` empty, which breaks the `while` loop. The code returns the `visited` list as the final output.

Comparing Tables 7-2 and 7-3 will help you gain a deeper understanding of the unique features of the DFS and BFS algorithms.

Project 32: Heuristic Searching with A*

In this project, we'll explore the A* search algorithm, an informed search algorithm that uses a heuristic function to guide the search. Its primary objective is to find the optimal path between two nodes in a graph by considering the cost of each path. To that end, it's best suited for working with weighted graphs, where each edge has an associated score. Figure 7-4 shows the graph we'll use for the project.

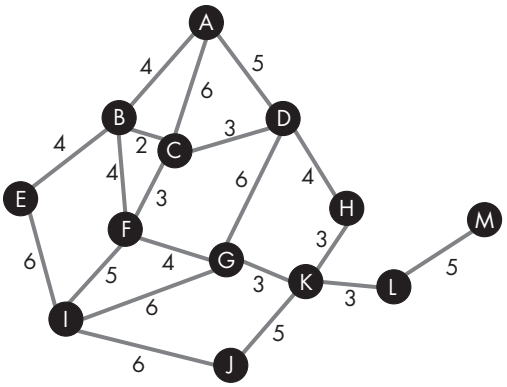


Figure 7-4: An example graph for Project 32 (start node = A, target node = J)

The graph in the figure has 13 nodes (A through M) and 20 edges, making it significantly more substantial than the example graph we used in the previous projects. The values along the edges represent the cost of traveling between the two nodes connected by that edge. We're assuming that the graph is undirected, so travel along an edge can go in either direction, and that the cost for each edge is *symmetric*, meaning it's the same no matter the direction of travel. For this project, we're interested in determining the lowest-cost route from node A (the start) to node J (the target).

As the A* algorithm traverses a graph, it uses two distinct functions to help make decisions. One calculates the *g-score*, the actual cost of traveling from the start node to the current node. The other calculates the *h-score*, the estimated or heuristic cost of traveling from the current node to the target node. Added together, these two scores give the *f-score*, the estimated total cost of the path:

$$f\text{-}score = g\text{-}score + h\text{-}score$$

One of the key strengths of the A* algorithm is its efficiency in finding the shortest path based on this informed approach. But for this to work, we need a good heuristic function.

The Heuristic Function

In the context of the A* search algorithm, a heuristic function, denoted as $h(n)$, is a function that estimates the cost from the current node to the target node in a graph. The purpose of the heuristic function is to guide the search algorithm by providing an informed estimate of how far a node is from the target, which helps A* make more efficient decisions about which nodes to explore next.

An *admissible* heuristic function for the A* algorithm is a function that never overestimates the cost of reaching the goal from any node. With an admissible set of h-scores, A* is guaranteed to find the shortest or least costly path. However, not all sets of admissible h-scores are equally good. The algorithm's performance depends on how close the h-scores are to the true costs. The more accurate the h-scores are, the faster the algorithm will find the optimal path.

Another desirable property of the heuristic function is consistency. A *consistent* function satisfies this condition: the cost of reaching the goal from a node is always less than or equal to the cost of reaching the goal from any neighbor of that node, plus the cost of moving to that neighbor. Consistency implies admissibility but not vice versa. A consistent set of h-scores can make the A* algorithm more efficient, as it will expand fewer nodes and converge to the optimal solution very quickly.

Consistent h-scores may be hard or impossible to obtain for large and complex real-world problems. However, we can still estimate admissible h-scores that are of high quality by using various techniques, depending on the problem type. Here are some common approaches for generating heuristic functions:

Ad hoc selection of h-scores

This method will work when the graph is small and it's possible to make conservative guesses about the h-scores depending on the depth of a node. For example, one can set all h-scores to some arbitrary small value that's guaranteed to be both admissible and consistent.

Domain knowledge

In some cases, domain-specific knowledge can be used to craft heuristic functions. This requires an understanding of the problem and what makes a good heuristic based on expert insights. For example, in the case of solving an eight-piece sliding puzzle with a 3×3 grid, a practical heuristic is the Manhattan distance, determined by adding the horizontal and vertical distances between each tile's current position and its target location.

Relaxation heuristics

This method involves simplifying a problem by temporarily ignoring certain constraints. Relaxation frequently results in an admissible heuristic because it tends to underestimate the actual cost. Take, for example, pathfinding problems, where one can use the Euclidean

distance between two points as a heuristic, ignoring any obstacles that may lengthen the path.

Abstraction

This method involves simplifying the problem representation by grouping or abstracting specific elements within it. Abstraction can lead to admissible and consistent heuristics. Consider, for example, a navigation problem, where you could choose to abstract the map by representing cities as nodes and major highways as edges, while ignoring smaller streets.

Pattern databases

In problems with large state spaces, where the graph includes numerous nodes and links (such as puzzle games), pattern databases can be employed to precompute heuristic values for subsets of the state space. These databases store the cost-to-goal for small subsets of the problem, and the heuristic for a given state is estimated as the sum of the costs associated with the relevant subsets.

In the context of the graph shown in Figure 7-4, we'll employ a combination of abstraction and ad hoc heuristic approaches to estimate a set of h-scores that are both admissible and consistent. Since we lack additional information about the nodes, such as their coordinates, we'll begin with a simplifying assumption (abstraction): all edges or links within the graph have the same weight or cost. Furthermore, we'll assume this weight equals the smallest weight found within the graph (ad hoc). Our approach can be summarized as a three-step process:

1. Edge weight assumption: Assume that all edges within the graph have an identical weight, and set this value to the smallest weight found within the graph.
2. Minimum links count: For each node, determine the minimum number of edges or links needed to traverse from that node to the target node.
3. H-score estimation: The h-score for each node is estimated by multiplying the smallest weight determined in step 1 with the minimum number of links needed to reach the target node, as found in step 2.

Given the relatively modest size of the graph, using this process to calculate h-scores is straightforward and quick. A brief examination of the weights reveals that the smallest one within the graph is 2 (for the link connecting nodes B and C). Now let's consider nodes I and K, immediate neighbors of the target node J. Their h-scores will be $2 \times 1 = 2$, since both I and K are only one link away from the target. Similarly, h-scores for nodes E, F, G, H, and L, which are two links away from the target, can be estimated as $2 \times 2 = 4$. Following this logical progression, the h-score for the starting node A, located farthest from the target, is estimated to be $2 \times 4 = 8$ because at least four links must be traversed to reach the target. Once these heuristic values are computed, you can easily incorporate them into

the application's `getHScore()` function, a lookup function that retrieves the h-score for a given node. (We'll discuss this function later, along with the rest of the code.)

Given our approach of utilizing the minimum number of links necessary to traverse from a given node to the target, along with our use of the smallest weight present in the graph for h-score calculation, the resulting h-scores meet the criterion of admissibility. They never overestimate the cost of reaching the target. I invite you to verify that these h-scores also meet the criterion of consistency as defined earlier in the section. You can do this either manually or by writing a few lines of code.

The Algorithm

Given our heuristic function, here are the steps we'll take to find the optimal path between two nodes by using the A* algorithm. This method assumes that at least one valid route exists between the starting node and the target:

1. Initialize two mutable maps to keep track of the visited and the unvisited nodes, respectively. The visited map starts empty; the unvisited map starts with all nodes in the graph.
2. Initialize each unvisited node's g-score and f-score to infinity (or the maximum possible value of the corresponding type) and its previous node property to "none."
3. Set the starting node's g-score to 0 (as the journey starts here, no previous node exists to come from), calculate or look up its h-score, and set its f-score equal to its h-score (since g-score = 0). Leave its previous node property set to "none."
4. While the unvisited map is not empty:
    a. Select the node with the lowest f-score from the unvisited nodes and designate that as the current node. (The starting node will be the first current node.)
    b. If the current node is the target node, add the current node to the visited map and terminate the loop (the target has been reached).
    c. Otherwise (when the current node is not the target node), retrieve the current node's neighbors from the graph.
    d. For each neighbor that has not already been visited, calculate a new g-score by adding the weight of the edge between the current node and the neighbor to the g-score of the current node. If this new g-score is lower than the neighbor's existing g-score, update the neighbor's attributes (g-score, f-score, previous node).
    e. Add the current node to the visited map and remove it from the unvisited map.

5. Once the loop ends, the visited map is returned, which contains information about the nodes explored during the search, their directional relationship (as captured in the "previous node" property), and the associated costs (g-scores and f-scores).
6. Use the information contained in the visited map to reconstruct the entire optimal path.

These steps outline the essence of the A* algorithm. They involve maintaining an open set of nodes to be explored and a closed set of nodes that have been visited, and calculating the cost of each node based on the actual cost from the starting node (g-score) and the estimated cost to the target node (h-score). By iteratively selecting the node with the lowest total cost (f-score), the algorithm efficiently finds the shortest path from the starting node to the target node.

The time complexity of the A* search algorithm depends on the nature of the problem and the quality of the heuristic function used. In the worst case, the time complexity of A* is $O(b^d)$, where b is the branching factor (the average number of edges per node) and d is the depth of the shallowest target node (the minimum number of edges or steps needed to reach the target from the starting node). The space complexity of the standard A* algorithm depends on the data structure used for the open and closed lists (for example, this could be implemented by using priority queues). In the worst case, the space complexity can be very high, also up to $O(b^d)$, due to the storage of nodes in the open and closed lists.

A well-chosen (admissible and consistent) heuristic can significantly improve the performance of A*, however, by efficiently guiding the algorithm to the target node, reducing the search space, and potentially making the actual time and space complexities much lower in practice. In the best-case scenario, when the heuristic function is perfect and the algorithm efficiently explores the most promising paths first, for example, A* can have a time complexity of $O(d)$.

The Code

The code for A* search is more involved than the code for DFS and BFS. For this reason, I'll break it down into a number of segments, starting with the global declarations and the `main()` function.

```
// no import block

data class Node(
    var gScore: Int,
    var fScore: Int,
    var previousNode: String
)

fun main() {
    // Define the graph to be searched.
  ❶ val graph = mapOf(
```

```
        "A" to mapOf("B" to 4, "C" to 6, "D" to 5),
        "B" to mapOf("A" to 4, "C" to 2, "E" to 4, "F" to 4),
        "C" to mapOf("A" to 6, "B" to 2, "D" to 3, "F" to 3),
        "D" to mapOf("A" to 5, "C" to 3, "G" to 6, "H" to 4),
        "E" to mapOf("B" to 4, "I" to 6),
        "F" to mapOf("B" to 4, "C" to 3, "G" to 4, "I" to 5),
        "G" to mapOf("D" to 6, "F" to 4, "I" to 6, "K" to 3),
        "H" to mapOf("D" to 4, "K" to 3),
        "I" to mapOf("E" to 6, "F" to 5, "G" to 6, "J" to 6),
        "J" to mapOf("I" to 6, "K" to 5),
        "K" to mapOf("G" to 3, "H" to 3, "J" to 5, "L" to 3),
        "L" to mapOf("K" to 3, "M" to 5),
        "M" to mapOf("L" to 5)
    )

    println("### A* algorithm ###")
    println("\nDisplay graph:")
    displayGraph(graph)

    val startNode  = "A"
    val targetNode = "J"    // Also, ensure its h-score is 0.
❷   val visitedList = aStar(graph, startNode, targetNode)

    println("\n--- Final Visited List ---")
    displayList(visitedList)
    displayShortestPath(visitedList, startNode, targetNode)
}
```

This application doesn't require an import block, as the search algorithm can be implemented without relying on any specialized library functions. The sole global component is a data class that holds three key attributes of a node: its g-score, its f-score, and the previous node along the optimal path.

In the `main()` function, we first define the graph shown in Figure 7-4 as a `Map` ❶. Each node is specified in terms of its name (`"A"`, `"B"`, `"C"`, and so on) along with an inner `Map` pairing each of the node's neighbors with the weight of the edge leading to that neighbor. You can think of `graph` as a map of maps (similar to a list of lists) encapsulating all nodes in the network and their interconnections. Once the graph is defined, it's printed by calling the `displayGraph()` function.

We next define the start and target nodes (`"A"` and `"J"` in this example) and call the `aStar()` function by passing the start and target nodes and the graph to be searched as arguments ❷. A call to this function returns a list of visited nodes (`visitedList`) as a `Map` of type `<String, List<Any>>`. This list represents a subset of nodes that the algorithm explored while trying to locate the optimal path. Crucially, A* search doesn't need to visit all nodes in the graph, as it relies on heuristic information to zoom in on the region that includes the optimal solution. We use the `displayList()` function to print this visited list and then call the `displayShortestPath()` function, which reconstructs and displays the optimal path.

The Display Functions

Let's take a closer look at the various display helper functions called from the `main()` function, starting with the `displayGraph()` function, which prints the whole graph.

```
fun displayGraph(graph: Map<String, Map<String, Int>>) {
    for ((node, neighbors) in graph) {
        println("Node: $node")
        print("Neighbors: ")

        for ((nNode, cost) in neighbors) {
            print("$nNode:$cost ")
        }
        println()
    }
    println()
}
```

This function takes in `graph` as its sole argument, which as we've seen is a `Map` of type `<String, Map<String, Int>>`. It uses two `for` loops to print the elements of `graph`. The outer loop cycles through the nodes, one at a time, printing each one. The inner loop extracts and prints each of the current node's neighbors, along with the associated edge weights (labeled as `Cost` in the output). You'll see how the output looks later when we examine the results.

Now we'll consider the `displayList()` function, which prints the characteristics of each visited node after the A* search is complete.

```
fun displayList(mapList: Map<String, Node>) {
    println("   (g-score, f-score, previous)")

    for ((node, attributes) in mapList) {
        println("$node: $attributes")
    }
    println()
}
```

This simple function uses a `for` loop to extract and print the collection of visited nodes and their attributes. Each element in this list, which is presented as a `Map` object, has two components: the name of the visited node and a `Node` object with three data points linked to the node—its g-score (`Int`), f-score (`Int`), and the previous node (`String`). The latter is the node from which we would have departed to reach the current node, ensuring the minimum f-score for the current node.

Finally, here's the `displayShortestPath()` function, which takes in the list of visited nodes, the start node, and the target node, and identifies the optimal path:

```
fun displayShortestPath(visited: Map<String, Node>,
                        startNode: String, targetNode: String) {
```

```
    var currentNode = targetNode
    var path = targetNode
    println("path initialized from target: $path")

    while (currentNode != startNode) {
      ❶ val previousNode = visited[currentNode]!!.previousNode
        // previousNode is placed before "path" so no need to reorder.
      ❷ path = previousNode + path
        println("previousNode: $previousNode")
        println("path updated: $path")
        currentNode = previousNode
    }

    val cost = visited[targetNode]!!.gScore
    println("\nThe shortest path from $startNode to $targetNode is:")
    println("Path: $path")
    println("Cost: $cost")
}
```

The function reconstructs the path in reverse, working backward from the target to the starting node. We start by initializing two variables to `targetNode`: `currentNode`, representing the current position in the path, and `path`, where the entire path is built up node by node. We then enter a `while` loop that iterates until `currentNode` becomes `startNode`. In the loop, we access `currentNode` from the list of visited nodes (supplied as a `Map` of type `<String, Node>`) and use its `previousNode` property to look up its previous node ❶. Next, we concatenate `previousNode` with the current value of `path` ❷ and update `currentNode` to `previousNode` for the next iteration.

After the loop ends, we retrieve `cost`, the g-score of the target node, from the list of visited nodes, using `targetNode` as the key. We then print the reconstructed optimal path and its cost.

The aStar() Function and Its Helpers

Let's now dive into the core of the A* algorithm implemented in `aStar()` and its helper functions. This code very closely follows the steps outlined earlier for implementing the A* algorithm.

```
fun aStar(graph: Map<String, Map<String, Int>>,
          startNode: String, targetNode: String):
          Map<String, Node> {

    // Define two mutable maps.
    val visited = mutableMapOf<String, Node>()
    val unvisited = mutableMapOf<String, Node>()

    // Initialize all unvisited nodes.
    for (node in graph.keys) {
        // The list is made of g-score, f-score, and previous node.
      ❶ unvisited[node] = Node(Int.MAX_VALUE, Int.MAX_VALUE, "none")
    }
```

```
    // Update the start node attributes in the unvisited list.
    val hScore = getHScore(startNode)

    // for startNode: g-score = 0, f-score = 10, previous node = none
  ❷ unvisited[startNode] = Node(0, hScore, "none")

    println("--- Initialized state of unvisited list ---")
    displayList(unvisited)

  ❸ while (unvisited.isNotEmpty()) {
        // Set the node with minimum f-score to current node.
      ❹ val currentNode = getCurrentNode(unvisited)

      ❺ if (currentNode == targetNode) {
            // Add the targetNode to visited.
            visited[currentNode] = unvisited[currentNode]!!
            println("--- Target node:$currentNode reached ---")
            break
        }

        val neighbors = graph[currentNode]!!

        for (node in neighbors.keys) {
          ❻ if (node !in visited) {
                val newGScore =
                    unvisited[currentNode]!!.gScore + neighbors[node]!!

              ❼ if (newGScore < unvisited[node]!!.gScore) {
                    unvisited[node] = Node(
                        newGScore,
                        newGScore + getHScore(node),
                        currentNode)
                }
            }
        }

        // Add currentNode to visited.
        visited[currentNode] = unvisited[currentNode]!!

        // Remove currentNode from unvisited.
        unvisited.remove(currentNode)
    }
    return visited
}
```

The algorithm begins by creating two mutable maps: `visited` and `unvisited`. At first, `unvisited` contains all the nodes in the graph, each initialized with the maximum possible g-score and f-score, and with a previous node of `"none"` ❶. The `visited` map, which is initially empty, keeps track of the nodes that have been visited. Next, the `startNode` in the `unvisited` map is updated to have a g-score of `0` and an f-score equivalent to its h-score ❷,

which is retrieved with the `getHScore()` helper function. As shown here, this helper is implemented as a simple lookup function:

```
fun getHScore(node: String) = when (node) {
    "A" -> 8   // start node
    "B" -> 6
    "C" -> 6
    "D" -> 6
    "E" -> 4
    "F" -> 4
    "G" -> 4
    "H" -> 4
    "I" -> 2
    "J" -> 0   // target node
    "K" -> 2
    "L" -> 4
    "M" -> 6
    else -> 0
}
```

These scores were estimated by using the hybrid three-step process explained earlier. Note that the h-score for the target node `"J"` is `0`.

Returning to the `aStar()` function, we next display the list of unvisited nodes and enter a `while` loop that continues until the `unvisited` map is empty or the target node is reached ❸. Within the loop, `currentNode` is set to the unvisited node with the minimum f-score by using the `getCurrentNode()` helper function ❹. Here's how that helper function is implemented by using Kotlin's built-in `.minByOrNull` method:

```
fun getCurrentNode(unvisited: Map<String, Node>) =
    unvisited.minByOrNull { it.value.fScore }!!.key
```

Back in `aStar()`, we check if `currentNode` is the same as `targetNode` ❺. If it is, we add the current node to the `visited` map and break the loop. Otherwise, for each neighbor of the current node not already in the `visited` map ❻, we calculate a new g-score by adding the edge weight to the current node's g-score. If the new g-score is lower than the neighbor's current g-score ❼, the neighbor's attributes in the `unvisited` map are updated: its g-score is set to `newGScore`, its f-score is set to its new g-score, plus its h-score (again retrieved with the `getHScore()` function), and its previous node is set to `currentNode`.

After processing all neighbors, the `currentNode` is added to the `visited` map and removed from the `unvisited` map. When the `while` loop terminates, the `visited` map is returned with all the information needed to reconstitute the optimal path.

USING NOT-NULL ASSERTION

The implementation of the A* algorithm in this chapter uses the not-null assertion operator (`!!`) multiple times. However, this is generally discouraged in favor of the Elvis operator (`?:`), which allows for more graceful handling of null exceptions. For Project 32's undirected graph problem, we can represent the graph by using a map of maps data structure. This approach is straightforward, intuitive, and particularly well suited for educational purposes. In real-world code, though, using the not-null assertion operator can be risky because it can lead to a `NullPointerException` if the value is `null`.

Therefore, in production-ready code that is expected to be used by other users, you should use safer options such as null-safe calls or the Elvis operator. For example, you could replace

```
val neighbors = graph[currentNode]!!
```

with:

```
val neighbors = graph[currentNode] ?:
                error("No neighbors found for $currentNode")
```

Using this approach, you can replace all code segments using `!!` with code to generate appropriate error messages.

The Result

We're now ready to run the code and have a look at its console output.

```
*** A-star algorithm ***

Display graph:
Node: A
Neighbors: B:4 C:6 D:5
Node: B
Neighbors: A:4 C:2 E:4 F:4
Node: C
--snip--
Node: L
Neighbors: K:3 M:5
Node: M
Neighbors: L:5

--- Initialized state of unvisited list ---
   (g-score, f-score, previous)
A: Node(gScore=0, fScore=8, previousNode=none)
B: Node(gScore=2147483647, fScore=2147483647, previousNode=none)
C: Node(gScore=2147483647, fScore=2147483647, previousNode=none)
--snip--
```

```
L: Node(gScore=2147483647, fScore=2147483647, previousNode=none)
M: Node(gScore=2147483647, fScore=2147483647, previousNode=none)

--- Target node:J reached ---

--- Final Visited List ---
   (g-score, f-score, previous)
A: Node(gScore=0, fScore=8, previousNode=none)
B: Node(gScore=4, fScore=10, previousNode=A)
D: Node(gScore=5, fScore=11, previousNode=A)
C: Node(gScore=6, fScore=12, previousNode=A)
E: Node(gScore=8, fScore=12, previousNode=B)
F: Node(gScore=8, fScore=12, previousNode=B)
H: Node(gScore=9, fScore=13, previousNode=D)
K: Node(gScore=12, fScore=14, previousNode=H)
G: Node(gScore=11, fScore=15, previousNode=D)
I: Node(gScore=13, fScore=15, previousNode=F)
J: Node(gScore=17, fScore=17, previousNode=K)

path initialized from target: J
previousNode: K
path updated: KJ
previousNode: H
path updated: HKJ
previousNode: D
path updated: DHKJ
previousNode: A
path updated: ADHKJ

The shortest path from A to J is:
Path: ADHKJ
Cost: 17
```

The output starts by printing the entire graph, node by node, along with each node's neighbors and edge weights. Next, the initial state of the `unvisited` map is shown after updating the starting node's attributes. Apart from node `"A"`, each node should have the maximum possible g- and f-scores (`2147483647`) and a previous node of `"none"`. Once the target node is reached, a message is printed before exiting the `while` loop. Then the final list of all visited nodes is printed. Looking over the list, we can see that not every node as been visited—nodes `"L"` and `"M"`, representing a dead end, were skipped. Notice also that the target node's g-score is the same as its f-score because its h-score is 0. Also, as expected, all g-scores are less than or equal to their corresponding f-scores. This is because the f-score is the sum of the g-score and the h-score, and the latter is assumed to be greater than or equal to 0.

Finally, the terminal output shows the step-by-step process of reconstructing the optimal path, followed by the full path and its total associated cost.

EXERCISES

Project 32 demonstrated a basic implementation of the A* algorithm, but you can add or experiment with many more features. The following four challenges outline different ways you could modify the program. For each challenge, you'll have to do additional online research to learn more about the key concept and possible implementation schemes.

1. Implement and experiment with different custom heuristic functions (h-scores) for a shortest-pathfinding problem, where each node represents a geographic location and the weights are actual distances between the connected nodes. Assume that the x- and y-coordinates for all nodes are also available. Compare the different heuristic functions' performance and accuracy in finding optimal paths.
2. Modify the A* algorithm to allow dynamic weighting of the heuristic function (h-score) relative to the cost so far (g-score). You can introduce a parameter (for example, a weight, w) that can be adjusted to change the relative importance of h- and g-scores. Typically, $w > 1$ increases the importance of h-scores over g-scores (speed over optimality), whereas $w < 1$ implies assigning more importance to g-scores and exploring a broader search space at the cost of speed of convergence. The default scheme used in Project 32 is equivalent to $w = 1$, where the g-score and h-score are considered equally important.
3. Implement A* by using different data structures for the unvisited and visited sets of nodes, such as priority queues, Fibonacci heaps, or even custom data structures. Measure the impact on the algorithm's efficiency.
4. Explore memory-efficient variations of A*, such as memory-bounded A* (MA*) or simplified versions like recursive best-first search (RBFS). Compare their memory usage and computational efficiency.

Summary

In this chapter, we explored some representative concepts and algorithms from two related domains: sorting and searching. These essential concepts and tools have extensive use in the realms of computer and data science, particularly in the context of information retrieval from databases, search engine performance optimization, data visualization, data mining, machine learning, and network routing.

Within the domain of sorting, we implemented the insertion sort, merge sort, and quick sort algorithms and gained insight into their respective strengths, weaknesses, time and space complexities, and stability characteristics. In the searching domain, our projects revolved around

navigating graph data structures. We implemented the depth-first search (DFS), breadth-first search (BFS), and A* algorithms.

Throughout these projects, we harnessed the power of various Kotlin features, including both stack and queue data structures, as well as lists, maps, and more intricate constructs like maps of maps. Last but not least, by tackling the exercises, you'll not only solidify your grasp of these core concepts but also raise your sorting and searching skills to a professional level.

Resources

Bhargava, Aditya Y. *Grokking Algorithms*. 2nd ed. Shelter Island, NY: Manning, 2024.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 4th ed. Cambridge, MA: MIT Press, 2022.

Even, Shimon. *Graph Algorithms*. 2nd ed., edited by Guy Even. New York: Cambridge University Press, 2012.

Heineman, George, Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell*. 2nd ed. Sebastopol, CA: O'Reilly, 2016.

Kopec, David. *Classic Computer Science Problems in Python*. Shelter Island, NY: Manning, 2019.

Skiena, Steven. *The Algorithm Design Manual*. 3rd ed. Cham, Switzerland: Springer Nature, 2020.

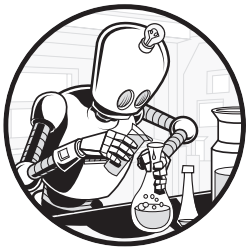
Wengrow, Jay. *A Common-Sense Guide to Data Structures and Algorithms*. 2nd ed. Raleigh, NC: The Pragmatic Bookshelf, 2020.

PART IV

OPTIMIZATION WITH NATURE-INSPIRED ALGORITHMS

8

THE GENETIC ALGORITHM



Many wonders of modern science were inspired by nature. Airplane and glider designs were based on the flight of birds. Camouflage—a tactic for survival—derives from mimicry, a form of antipredator adaptation. The hooked barbs of a thistle led to the invention of Velcro. Even rather dull-looking termite mounds teach us about natural ventilation and cooling, an idea used in modern architecture.

The world of computing is no different. The exciting field of machine learning, especially deep learning, is inspired by how the human brain processes information. By copying natural strategies that evolved over millions of years, we've developed algorithms to solve problems that were previously thought to be unsolvable with traditional mathematical tools.

In this chapter and the next, you'll learn how these nature-inspired algorithms work, about their advantages and limitations, and how to

implement them in Kotlin. This chapter focuses on the genetic algorithm, an evolutionary process–based method. The next chapter covers particle swarm optimization and ant colony systems, two methods that mimic the behavior of biological agents or species. For each method, I'll start with the key concepts and then show you how to code and apply them to real-world problems.

Nature-Inspired Algorithms

Nature-inspired computing refers to observing how nature solves complex biological or physical problems and then applying similar strategies to contemporary scientific, engineering, or management problems. The core of nature-inspired computing is *nature-inspired algorithms (NIAs)*, which rely on strategies learned from nature.

Biology-based NIAs can be observed in natural processes, such as the evolution of a species or the functioning of neurons in the human brain. These processes led to the development of genetic algorithms and deep neural networks. Individual and collective behaviors of members (or *agents*) of a population can also form the basis for new NIAs. For example, the foraging behavior of ants around their colony inspired the ant colony optimization algorithm. Whereas ants tend to work independently without any explicit collaboration with other members of the colony, the behavior of a large school of fish or birds indicates swarm intelligence, which has led to the development of the particle swarm optimization algorithm.

Even nonliving natural processes involve embedded strategies optimized for meeting certain goals. Examples of such algorithms include gravitational search (based on Newton's law of gravity) and simulated annealing (based on thermodynamics). In general, these algorithms serve as powerful tools for optimizing various processes or systems, resulting in significant gains in efficiency and cost savings.

Before going into detail on genetic algorithms, I'll introduce the concepts of optimization and global solutions. Additionally, I'll highlight instances where NIAs prove more effective than traditional mathematical tools for solving real-world problems.

The Optimization Problem

NIAs are well suited to solving optimization problems, in which we want to find the best solution of all possible solutions. To solve such problems, we minimize or maximize an *objective function*, a mathematical expression that represents the goal of what we want to achieve through optimization. It is expressed in terms of one or more *decision variables*, quantities we can adjust to optimize the objective function.

For real-world problems, the decision variables will be bounded. Additional constraints may limit and define the decision space within which the optimal solution must be found.

Let's consider a simple example with only one bounded decision variable:

$$\text{Minimize } f(x) = x^2 - 2 \tag{8.1}$$

$$-3 \leq x \leq 3$$

In Equation 8.1, $f(x)$ is an objective function of a single variable x. Our goal is to find the value of x for which $f(x)$ will be minimum, provided x stays within ±3.

Since x in this case has an exponent of 2, x^2 will always be positive (irrespective of whether x is positive or negative) and will continue to increase as x increases in absolute terms. Therefore, the right-hand side of Equation 8.1 will have the smallest value when $x = 0$. In other words, the optimal solution (marked by an asterisk) for this problem is $x^* = 0$, and the corresponding optimum value of the function is $f(x)^* = -2$.

Figure 8-1 shows a visual representation of this function, which takes the shape of a parabola with its vertex at (0, –2). We can also visually confirm that $f(x)$ has its minimum at point C (the vertex of the parabola).

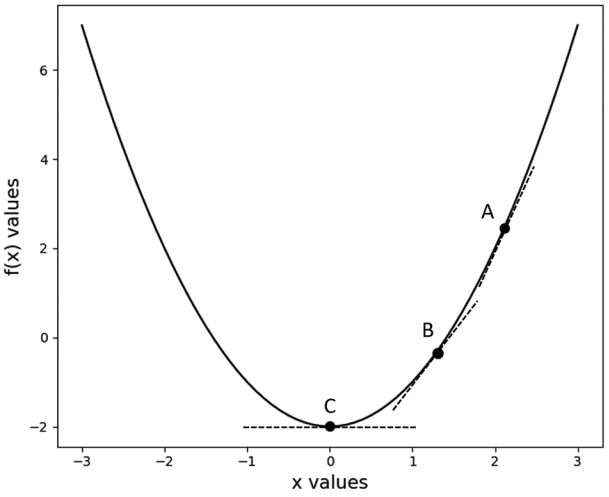


Figure 8-1: The optimal value for a parabolic function

A property of this function allows us to identify the optimal solution without knowing its exact location. The dashed lines touching the function at points A, B, and C in Figure 8-1 show the *slope*, also called the *gradient*, of the function at those locations. The slope of a function measures how much the value of a function changes when the value of the decision variable changes by a small amount. Notice that the gradient at point C, where the function value is minimum, is 0 (the dashed line is horizontal). Thus, if we randomly started our search for the optimal solution at point A, we could have moved in the direction of decreasing gradient (for example, from A to B or from B to C) until the gradient becomes 0. If we continue to move beyond the vertex to the opposite side, the slope will change its direction

and start increasing. This will cause the function value to increase, and we'll move away from the optimal solution.

For a function that is smooth (no kinks) and continuous (no jumps) and has only one maximum or minimum within the decision space, the gradient-based search strategy will always work in finding the *global optimum*—the best possible solution for a given problem. In fact, for a well-behaved function like this, we can find the optimal solution by simply setting the slope of the function with respect to the decision variable (called the derivative in differential calculus) to 0 and solving the resulting equation for the optimal solution. This approach will also work for functions with two or more decision variables as long as the function is *well behaved*, meaning it is both smooth and continuous.

Things get messier when we deal with a multimodal function with multiple locations where the gradient is 0, as shown in Figure 8-2.

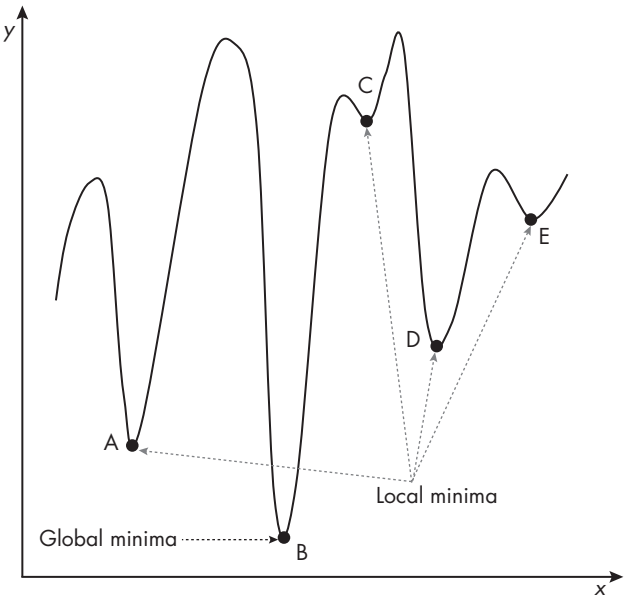


Figure 8-2: Local and global minima for a univariate function

Figure 8-2 shows four local minima at points A, C, D, and E, and one global minimum at point B within the decision space. In a situation like this, whether an algorithm based on gradient descent will converge to the global minimum depends on the point from which we start the search. Nothing guarantees that we'll find the global minimum unless we make multiple attempts from different starting points (initial conditions).

For a better appreciation of the challenge involved when we try to find global optima for a multivariate function, consider the graph in Figure 8-3. This shows the results of the two-variable Eggholder function, discussed further in the final project of this chapter. For a problem like this, a simple gradient-based algorithm can easily get stuck at one of the many local minima. To make things worse, the equations defining such functions are

typically not differentiable, and we cannot use calculus-based tools to estimate the global optima.

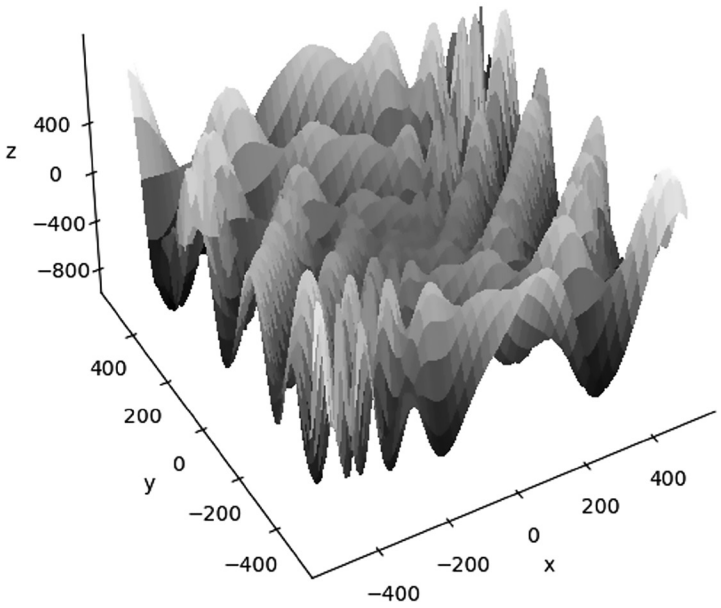


Figure 8-3: The Eggholder function with numerous local minima

Functions of two decision variables have a silver lining, however: we can create 3D plots of these functions for a bounded decision space. Based on a visual inspection of the surface or contour plots, it may then be possible to narrow down the search space to a number of smaller subzones where we can conduct an extensive local search to uncover the global minima (more than one global minimum could exist).

What about functions of higher dimensions? In fact, complex real-world optimization problems can have hundreds of decision variables. It is not humanly possible to conceive what a function of several hundred variables might look like in a *hyperspace* (a higher-dimensional space beyond human comprehension). Our best bet for identifying an optimal or near-optimal solution in a hyperspace is to conduct a broad-based search combining *heuristics* (special knowledge about the nature of the problem) and *randomization* (selecting initial conditions or intermediate values randomly). This strategy is likely to allow the algorithm to escape local optima and find solutions that are superior to what a pure random search might reveal. We typically repeat this process numerous times and accept the best-so-far solution as a proxy for the unknown global optima.

Even when looking for an optimal combination of decision variables that can have only discrete values (whole numbers), the brute-force approach of trying out all possible combinations normally doesn't work in higher dimensions. This is because the number of combinations may be so large that it is practically impossible to complete that search in a reasonable

amount of time. It's in this context that nature-inspired algorithms come to our rescue.

When to Use NIAs

Compared to traditional mathematical tools, NIAs are less sensitive to the nature or complexity of the optimization problem. An objective function may be nonlinear, nonsmooth, multidimensional, and multimodal, but these attributes are not a big concern for NIAs (though we still have to choose the right tool from the basket of options). NIAs are especially suitable for solving very large optimization problems and finding near-optimal solutions without expending too many resources (such as computational time or energy use).

Traditional optimization methods, whether they employ a gradient descent algorithm or not, are deterministic: if we start the search from a given point, we'll always reach the same solution or approximation after a given number of steps. This feature makes deterministic algorithms more prone to getting stuck at local optima because no built-in freedom exists to explore a different path unless the initial condition is changed.

NIAs, on the other hand, are *stochastic*, meaning that their results cannot be predicted beforehand. This is because NIAs typically have multiple built-in steps that rely on random selection. For the same initial condition, a stochastic algorithm can produce very different results. This innate ability to randomly choose a different path allows NIAs to avoid getting stuck at local optima and to eventually find the global or near-global optima.

In addition, some NIAs are based on the efforts of agents that operate independently (for example, ants in the ant colony optimization algorithm). This allows us to implement the algorithm so that it can benefit from parallel processing to improve computational efficiency.

In sum, we can use NIAs to solve large, complex, multidimensional optimization problems for which no known analytical solutions exist or for which such solutions cannot be found due to the nature of the problem. However, NIAs are not the ideal choice for solving the many optimization problems that can be efficiently solved using deterministic methods (for example, using linear or integer programming or various graph search algorithms).

An Overview of the Genetic Algorithm

The genetic algorithm is among the best-known NIAs. It is modeled after the biological evolution of species driven by both the sexual reproduction of parents, who contribute genetic materials, and natural selection (survival of the fittest). In addition to inheriting genes from its parents, the offspring's *chromosomes* (collections of genes) undergo random alterations called *mutation* that introduce new features to its gene pool. The offspring is then subjected to a selection process based on its *fitness* (a measure of how well an individual contributes to reaching a certain goal) before it is allowed to reproduce. The process eventually leads to a generation of individuals with a significantly enhanced gene pool.

Figure 8-4 shows the main components of the genetic algorithm.

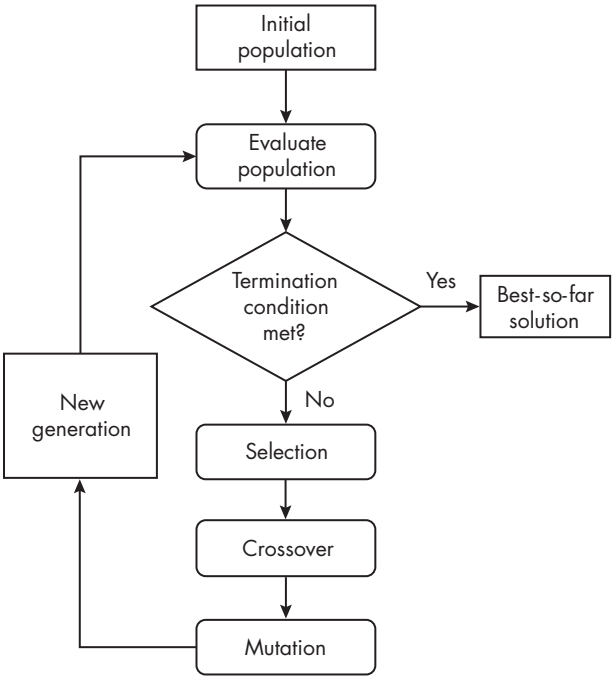


Figure 8-4: The key components of the genetic algorithm

All genetic algorithms start with a population of randomly created individuals. Each individual is essentially a potential solution represented by its gene pool. These individuals are evaluated and screened based on their fitness, which we attempt to maximize or minimize until a termination condition is met. Otherwise, we choose a batch of individuals with better fitness values who are then allowed to mate, produce offspring, and replace their parents as the next generation. I'll explain these steps further in the upcoming sections.

Genetic Operators

Genetic operators include the three core components of genetic algorithms—selection, crossover, and mutation—that work in tandem and allow the algorithm to converge toward a solution. *Selection* refers to the process of choosing an individual from a population based on their fitness (their potential contribution to finding the optimal solution). Selection may involve the entire population or a subset of the population, as individuals are drawn at random based on specific strategies. *Crossover* involves combining genetic materials from parents to create offspring. In the genetic algorithm, it always involves two parents and is therefore a binary operator. *Mutation* is a random alteration of an individual's genetic information. It is a unary operator because it is applied to one individual at a time.

Selection

The selection operation ensures that better genes are passed on from one generation to the next. The implementation of this process may vary depending on the problem, but the end goal is to select two parents (chromosomes) to participate in the reproduction through crossover. The commonly used strategies for selection include tournament, roulette wheel, and rank-based selection.

Tournament

The tournament selection process is based on running fitness-based competitions among randomly selected individuals, as shown in Figure 8-5.

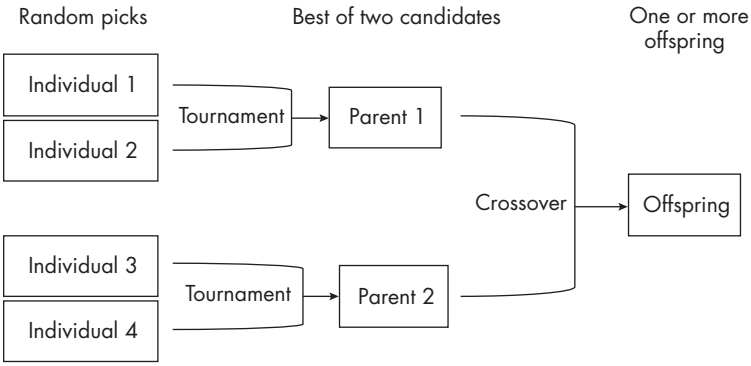


Figure 8-5: Using tournaments to select parents

To create a new child, the process starts by randomly selecting four individuals grouped into two pairs. From each pair, the individual with better fitness is selected as a parent. The process selects two parents per round who will reproduce via crossover (explained later) to give birth to an offspring.

Roulette Wheel

As the name implies, roulette wheel selection is comparable to spinning a dial on a board divided into segments. The area of these segments is proportional to the relative fitness of the members of the population from which parents are to be chosen. Let me explain the process with a numerical example, as shown in Table 8-1.

Table 8-1: Roulette Wheel Data

Individual	Fitness	Relative fitness (RF)	Cumulative RF
P1	12	0.286	0.286
P2	5	0.119	0.405
P3	8	0.190	0.595
P4	10	0.238	0.833
P5	4	0.095	0.929

Individual	Fitness	Relative fitness (RF)	Cumulative RF
P6	3	0.071	1.000
SUM =	42	1.000	

Figure 8-6 shows the graphical representation of the example in Table 8-1.

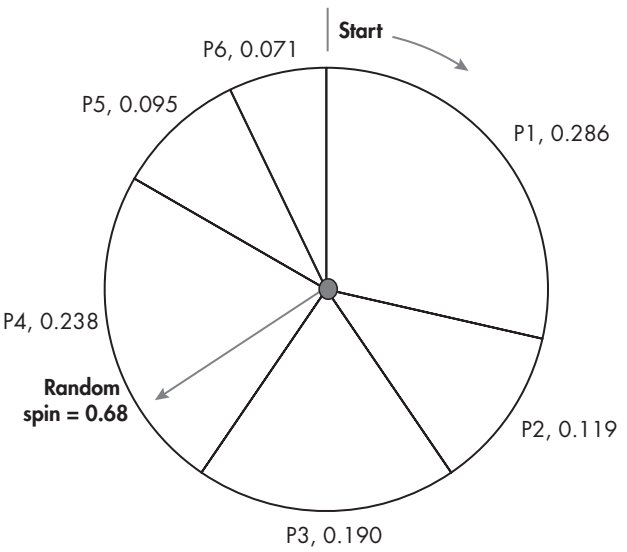


Figure 8-6: Selecting parents using the roulette wheel method

In this example, we consider a population of six individuals, P1 through P6. Their fitness values are given in the second column of Table 8-1. The relative fitness (RF) values are calculated by dividing individual fitness values by the sum of all individual fitness values (for example, RF for P1 = 12/42). The last column represents the cumulative RF (CRF), which is created by adding all RF values up to a certain row. For example, the CRF corresponding to P2 = 0.286 + 0.119 = 0.405. The last CRF, which is the sum of all individual RF values, will be 1.0. In the roulette wheel scheme, RF values are used as proxy probabilities for individuals to be selected at random when an unbiased virtual dial is spun.

In Figure 8-6, these six individuals are represented by six different segments whose areas are the same as their RF values (shown next to the individual names). To implement the roulette wheel method, we draw a random number between 0 and 1 from a uniform distribution, which has the same effect as spinning the dial. (This is done programmatically by using the `random()` method in the standard Kotlin math library.) Let's say that the value of this random number is 0.68, equivalent to having the dial stop inside the fourth segment (between CRFs of 0.595 and 0.833). Based on this draw, we would select P4 as parent 1 and repeat the process one more time to choose parent 2.

Rank-Based Selection

The third selection method, rank-based selection, is very similar to the roulette wheel method. Here, we order the individuals in ascending or descending order, depending on the problem, and assign each individual a rank based on their fitness. If two or more individuals have the same fitness, they are assigned an average value (based on their positions in the ordered list) as their rank. Finally, the ranks are used to calculate RF values and select the mating parents as we would using the roulette wheel scheme.

Crossover

The crossover operation is designed to intermix the genes of two parents to create one or two offspring who become members of the next generation. As with the selection operator, many ways of splitting the chromosomes and recombining the genes are available. Figure 8-7 shows the schema for a simple but effective approach to this operation, called a single-point crossover.

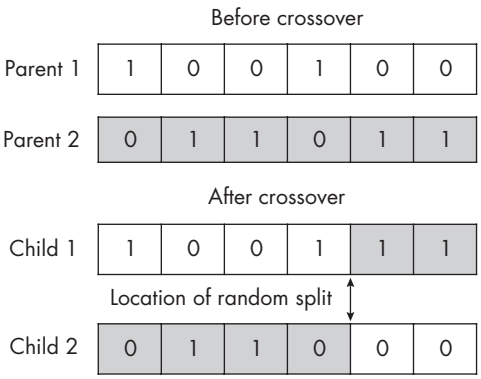


Figure 8-7: The single-point crossover operation

We start the process by identifying two parents through the selection operation. These parents would normally have chromosomes consisting of different genes. In the example in Figure 8-7, both parents have chromosomes made of binary genes denoted by 0 or 1. Parent 1's genes are shown as white cells, whereas parent 2's genes are gray cells.

The first step of the crossover operation is to draw a random integer from a uniform distribution between 1 and the number of genes minus 1, which would be between 1 and 5 (inclusive) in our example. Let's say the integer drawn is 4. We'd then split chromosomes of both parents at this location (between the fourth and fifth genes shown in Figure 8-7). Finally, we'd swap the split parts by adding the last two genes from parent 2 to parent 1 (the two gray cells of child 1) and adding the last two genes from parent 1 to parent 2 (the two white cells of child 2).

In this example, we used two parents to create two children. However, we could also decide to produce only one child per iteration to keep the algorithm simple and easy to code. For *real-coded genes*—genes represented by real numbers—a crossover operation will produce only one child

because of the way the method is implemented. We'll discuss real-coded genes further in the final project of this chapter.

Many other types of crossover operations, such as multipoint crossover and ordered crossover, exist. For real-coded genes used in mathematical function optimization, crossover operations could be based on an arithmetic, geometric, or weighted mean of fitness values.

Mutation

Mutation involves randomly changing the values of genes or, for real-coded genes, adding a small noise to those values before adding a child to the next generation. Mutation is applied to every gene in the chromosome one at a time. First, we randomly draw a real number between 0 and 1 and compare that with a mutation threshold (probability), typically set to a very small value. If the random value drawn is less than or equal to the mutation threshold, we alter the genetic content for that gene. For a binary chromosome where the genes are either 1 or 0 (indicating inclusion or exclusion of some entity in the solution), this alteration is conducted by flipping the gene value from 0 to 1 or vice versa.

Figure 8-8 visually explains this process.

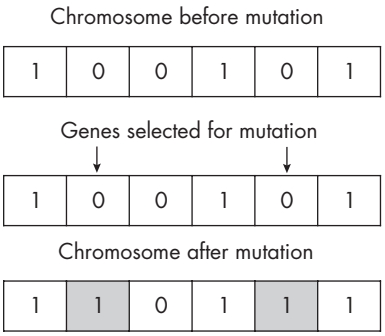


Figure 8-8: Mutation in a binary chromosome

In Figure 8-8, the second and fifth genes have been randomly selected for mutation. Given that these are binary genes, their gene values have been flipped from 0 to 1.

Elitism

Before we move on to tackling our first genetic algorithm project, I'll introduce one more important concept—*elitism*. This technique involves sorting the current population based on their fitness, then adding a fraction of that sorted population to the next generation before attempting crossover and mutation. This operation is called elitism because it favors the fittest individuals. Elitism generally helps reduce the number of computations needed to locate the optimal solution because it protects some of the best chromosomes from getting altered or diluted by crossover and mutation operations.

Project 33: Evolve Gibberish into Shakespeare

In our first coding project, we'll create a population with random collections of genes as their chromosomes. We'll then use a genetic algorithm to refine those chromosomes until one of the individuals becomes as eloquent as Shakespeare and repeats Hamlet's famous line "To be, or not to be: that is the question," expressed in its gene sequence!

The Strategy

To solve this problem, we'll create a population of size 100. No hard-and-fast rule applies on this, and a bit of experimentation is required to estimate a reasonable size for a given problem. Many factors are at play that will determine the convergence rate of the algorithm, including population size, the way the genetic operators are implemented, and the stopping condition. One possible strategy is to start with a smaller population size and then gradually increase it until further improvements in the solution become negligible.

Next, we need to determine the size of the chromosomes. For this specific problem, each individual's chromosome will have 42 genes—the length of the text we aim to reproduce using the algorithm. These genes will be randomly selected from a pool of 87 genes, which in this case is a collection of alphanumeric characters (including punctuation and parentheses). Since our goal is to exactly match the target text, this collection includes both uppercase and lowercase letters.

In our genetic algorithm implementation, we'll use elitism and tournament-based selection as our operators. Additionally, we'll employ a single-point crossover scheme. For mutation, we will use a threshold of 1/42 to ensure that on average one gene will mutate for each new child created via crossover.

The Code

The overall structure of the code closely resembles the general structure of the genetic algorithm described in Figure 8-4. We'll discuss each of its components in the following sections.

Global Declarations

In this code segment, we create a data class, and declare and/or set required global parameters and collections. We also create two mutable lists of data objects to store population states for the current and next generations.

```
data class Solution(
    val chromosome: String,
    val fitness: Int
)

❶ val TARGET = "To be, or not to be: that is the question."
❷ val VALID_GENES: String =
```

```
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" + // letters
            "1234567890" +                                    // numbers
            ", .-;:_!/?#%&()={[]}$*@\"\'"                     // symbols
val chromosomeLength = TARGET.length
val POP_SIZE = 100
val MAX_GEN = 1000
val ELITISM = 0.15
val eliteSize = (POP_SIZE * ELITISM).toInt()
❸ val MUTATION_THRESHOLD = 1.0/chromosomeLength

val population: MutableList<Solution> = mutableListOf()
val nextgen: MutableList<Solution> = mutableListOf()
```

Let's walk through this segment step by step. At the top of the block, we create a `Solution` object (data class) that will be used to create the individuals who will make up the population and undergo genetic alterations.

Next, we define the target string or the desired end state for the fittest individual in the population ❶. The target string has 42 characters (including spaces), which are stored in a string named `TARGET`. The target string is built from a pool of genes—characters that we typically use while composing phrases in English. This gene pool is saved as `VALID_GENES` ❷.

We set the population size (`POP_SIZE`) to 100 and the number of generations (`MAX_GEN`) to 1,000. We also employ elitism. Fifteen percent of the population (the top 15 fittest individuals) will be automatically included in the next generation. The remaining members of the next generation will be produced through selection, crossover, and mutation. The threshold for mutation has been set to `1.0/chromosomeLength` ❸ so that on average 1 gene out of 42 will undergo mutation per offspring. (You may need to adjust this rule of thumb for other optimization problems. For example, you may have to explicitly set the mutation threshold from 1 to 3 percent when too few genes exist in the chromosome.)

The last two lines create two mutable lists of type `Solution`, which store the individuals belonging to the current generation (`population`) and to the next generation (`nextgen`).

Initializing Population and Fitness Evaluation

The initial population is created by making a call to the `initPopulation()` function, which in turn relies on the `getFitness()` helper function.

```
fun initPopulation() {
    // Initialize a population of POP_SIZE individuals.
    for (i in 0 until POP_SIZE) {
        var chromosome = ""
        for (j in 0 until chromosomeLength) {
           ❶ chromosome += VALID_GENES.random()
        }
        // Calculate fitness of the new chromosome.
        val fitness = getFitness(chromosome)
        // Add the new individual to the population.
      ❷ population += Solution(chromosome, fitness)
    }
```

```
    // Sort population (in place) in descending order.
    population.sortByDescending {it.fitness}
    println("\nBest solution from initial population:")
    println(population[0].toString())
    println("\n... initPopulation done ...\n")
}

fun getFitness(chromosome: String): Int {
    var fitness = 0
    val pairs = TARGET.zip(chromosome)
    for (pair in pairs) {
      ❸ if (pair.first == pair.second)
                fitness += 1
    }
    return fitness
}
```

The `initPopulation()` function creates the number of individuals specified by `POP_SIZE` (100, in this case) whose chromosomes are created by randomly picking individual genes (all 42 of them) from the supplied gene pool, `VALID_GENES` ❶. Once the chromosome is complete, its fitness is evaluated by calling the `getFitness()` function. A new `Solution` is created using the chromosome and fitness value and then added to the `population` ❷.

Before exiting this function, we sort the population in descending order and print the best solution from the initial population. This presorting is needed to check for the termination condition and implement elitism for the first generation inside the `runGA()` function. For subsequent generations, sorting is done at the end of each iteration inside `runGA()`.

Within the `getFitness()` function, we create a list named `pairs` of type `Pair<Char, Char>` and calculate the fitness value for the given chromosome based on pair-wise comparisons. For each matching gene, fitness is incremented by 1 ❸. If a chromosome matches the target string exactly, it will have a maximum fitness value of 42.

The Driver Function

In the code block for the `runGA()` function, we implement the core components of the genetic algorithm. This includes iterating over multiple generations, checking for the termination condition, and creating the next generation by using elitism, selection, crossover, and mutation—the entire collection of genetic operators.

```
fun runGA() {

    // Iterate for a specified number of generations.
  ❶ for (generation in 1 .. MAX_GEN) {

        // Step 1: Check for termination condition.
      ❷ if (population[0].fitness >= chromosomeLength) {
            println("\n*** Target reached at generation = " +
                    "${generation - 1} ***\n")
```

```
            break
        }

        // Step 2: Implement elitism.
    ❸ selectElites()

        // Step 3: Implement crossover and mutation.
    ❹ for (i in eliteSize until POP_SIZE) {
            // Select parents for crossover.
            val parent1 = tournament()
            val parent2 = tournament()

            // Produce a child by using crossover and mutation.
            val child = crossover(parent1, parent2)

            // Add the child to nextgen.
            nextgen += child
        }

        // Step 4: Transfer nextgen to the current population.
    ❺ for (i in nextgen.indices)
            population[i] = nextgen[i].copy()

        // Step 5: Clear nextgen for the next iteration.
        nextgen.clear()

        // Step 6: Sort population in descending order (in place).
     ❻ population.sortByDescending {it.fitness}

        // Step 7 (optional): Print the best solution per generation.
        val formatString = "%5d %44s %4d"
    ❼ println(formatString.format(generation,
            population[0].chromosome, population[0].fitness))
    }
}
```

The outermost `for` loop ❶ runs the genetic processes for the specified number of generations. Inside this loop, we first check for the termination condition by comparing the best fitness value from the current `population` with the maximum possible fitness ❷. If the condition is met, the program terminates after printing a message that it has reached the target. If the condition is not met, we implement elitism by calling the `selectElites()` function ❸, discussed in detail in the next section.

We then move on to the first inner `for` loop ❹, which creates the remaining members of the next generation by selecting new parents by tournament, creating a child by calling the `crossover()` function (which also applies mutation to the newly created chromosome, as discussed in the next section), and then adding the child to the mutable `nextgen` list.

We use a second inner `for` loop ❺ to individually copy the next-generation solutions (`nextgen`) to `population` before `nextgen` is cleared for the next iteration. Notice that given the simple structure of the `Solution` data class, the `copy()` method applied to the elements of `nextgen` creates a deep copy and prevents cross-referencing between `population` and `nextgen`. In addition, transferring

nextgen values to population at the end of each iteration eliminates the need to store multiple generations of solutions, which saves a lot of memory.

In the final segment of the outermost for loop, we sort the newly updated population in descending order ❻ and print three key values per generation: the iteration number, the chromosome with the best fitness, and the corresponding fitness ❼.

The Operator Functions

The runGA() function relies on several operator functions that perform the key genetic operations.

```
fun selectElites() {
    // Assign top eliteSize individuals to nextgen.
  ❶ for (i in 0 until eliteSize)
        nextgen += population[i].copy()
}

fun tournament(): Solution {
    // random sampling with replacement
    // Use the entire population, including elites.
    val candidate1 = population.random().copy()
    val candidate2 = population.random().copy()
    // Return the winner of the tournament.
  ❷ return if (candidate1.fitness >= candidate2.fitness) candidate1
             else candidate2
}

fun crossover(parent1: Solution, parent2: Solution): Solution {
    // random single-point split crossover
    val split = (1 until chromosomeLength).random()

    // Use slice to extract segments from a string.
  ❸ val crossChromosome =
       parent1.chromosome.slice(0 until split) +
               parent2.chromosome.slice(split until chromosomeLength)

    // Apply mutation to crossChromosome.
  ❹ val newChromosome = mutation(crossChromosome)

  ❺ return Solution(newChromosome, getFitness(newChromosome))
}

fun mutation(crossChromosome: String): String {
    // A String object is immutable in Kotlin.
    // Create a char array whose elements can be modified.
    val chars = crossChromosome.toCharArray()
    for (i in 0 until chromosomeLength) {
      ❻ if ((0..1000).random()/1000.0 <= MUTATION_THRESHOLD)
            chars[i] = VALID_GENES.random()
    }
  ❼ return String(chars)
}
```

The `selectElites()` function is a one-liner. It promotes the top 15 individuals (`eliteSize` = 15) from the current generation to the next generation without subjecting them to further genetic processes ❶.

The `tournament()` function randomly picks two individuals from the current population and returns the winner of the competition based on their fitness values ❷.

The `crossover()` function takes in two parents as arguments, splits their chromosomes at a random location, and combines the split parts from both parents to create a new chromosome for the offspring ❸. Next, this newly created chromosome (`crossChromosome`) is passed to the `mutation()` function ❹, which returns the final chromosome saved as `newChromosome`. A single offspring is then returned once the fitness value for the newly created chromosome is calculated by making a call to `getFitness()` ❺.

Finally, the `mutation()` function applies mutation to randomly selected genes. It first converts the chromosome from a string object to a character array because strings are immutable in Kotlin. The mutation operation, triggered by the `MUTATION_THRESHOLD` parameter ❻, is applied to each gene in the chromosome. Once the mutation operation is done, the character array is converted back to a string and returned as the new (mutated) chromosome ❼.

The main() Function

The `main()` function simply prints a few key problem-specific parameters and makes two function calls to finish the job.

```
fun main() {
    println("\n*** Text-matching using the genetic algorithm ***\n")
    println("Target string: $TARGET")
    println("Population size: $POP_SIZE, Generations: $MAX_GEN, " +
            "Chromosome length: $chromosomeLength")
    println("Mutation threshold: $MUTATION_THRESHOLD")

    // Initialize the population.
    initPopulation()
    // Run the genetic algorithm.
    runGA()
}
```

The first call to `initPopulation()` initializes the current population with random chromosomes. The second call to `runGA()` conducts the necessary genetic operations.

The Result

Each time you run this program, it will take a different number of iterations to exactly match the target string. This is because we're using a stochastic method that depends on many internal levels of random selection. This is a very helpful feature for solving large real-world problems that may not have a deterministic or known solution.

Here is some sample output from the program:

```
*** Text-matching using the genetic algorithm ***

Target string: To be, or not to be: that is the question.
Population size: 100, Generations: 1000, Chromosome length: 42
Mutation threshold: 0.023809523809523808

Best solution from initial population:
Solution(chromosome=u[n_ebJvtj=J[h5j{bNx:BhPch'qyM/)3RVz"K_]P:, fitness=3)

... initPopulation done ...

    1   u[n_ebJvtj=J[h5j{bNx:BhPch'qyM/)3RVz"K_]P:    3
    2   c_g-i1KpZQn[[qXq%hwp:,shb ]7k?PEL_ol @izl.    4
    3   C@eSnKo7T_b6o@thqvgL Kh=FU[( &bCF{veDP"4/d    5
    4   C@eSnKo7T_b6o@thqvgL Kh=FU[( &bCF{veDP"4/d    5
    5   rnkFi6Z8U /NP An%d]m&vSZSS{&6F/e=qJ9*iio#k    6
    6   yT;_e}Jvtj=J[h5j{bNx:BhPch[( &bF qJS @iz/d    7
    7   yT;_e}Jvtj=J[h5j{bNx:BhPch[( &bF qJS @iz/d    7
    8   342y"BZo@_b6o@thqvgL Kh=FD[( &bCFqJSq@izl.    8
    9   342y"BZo@_b6o@thqvgL Kh=FD[( &bCFqJSq@izl.    8
   10   p42y"BZo;bTcXxD?{bNL BhPcU[( &bCF{veDPiol.    9
   11   342y"aZo@_b6o@thqvgL Kh=FD[( ?/e qJ9*iio#k   10
   12   =[eSnKo8U XNP thqvgL Kh=FD[( ?/e qJ9*iio#k   11

--snip--

  370   To be, or not to be: that i( the question.   41
  371   To be, or not to be: that i( the question.   41
  372   To be, or not to be: that i( the question.   41
  373   To be, or not to be: that i( the question.   41
  374   To be, or not to be: that i( the question.   41
  375   To be, or not to be: that i( the question.   41
  376   To be, or not to be: that i( the question.   41
  377   To be, or not to be: that i( the question.   41
  378   To be, or not to be: that i( the question.   41
  379   To be, or not to be: that is the question.   42

*** Target reached at generation = 379 ***
```

In this instance, starting with chromosomes that had no resemblance to the target string, it took 379 generations for the algorithm to re-create the target string exactly. We haven't made any attempt to fine-tune the global parameter values to increase the speed of convergence, yet the code converges to the optimal solution almost instantly (the processing time will depend on the configuration of your device). Pretty impressive!

EXERCISE

Modify the `crossover()` function to create and return two children rather than one child. You'll now need half as many calls to the `crossover()` function to create the next generation. This will, among other things, require you to modify the inner `for` loop of the `runGA()` function. While this will reduce the number of computations, its impact on the efficiency of the algorithm is not certain. You can run both versions of the program by using the same set of global parameters to test whether one version consistently outperforms the other. To be sure, solve several test problems instead of just one.

Project 34: Solve the Knapsack Problem

You're likely familiar with Noah's ark, the vessel Noah and his followers built to save themselves from a great deluge. The challenge that Noah faced was massive: he had to build a vessel of unprecedented size and choose who or what to take on board. To a mathematician, this latter decision is a classic example of an optimization problem where one tries to maximize the value of the objects that can be accommodated within a limited space.

Let's build a miniature version of this challenge and solve it using the genetic algorithm. We'll name this project Jonah's ark. Jonah lives in a flood plain that faces the risk of flash floods. Jonah knows he must be ready to leave the area at short notice. His quickest route to safety involves using a small-engine boat to get away from the rising river through a tributary beyond the reach of flood waters. Of course, the boat is small and can carry only so many items without sinking. Jonah must decide which of the valuable objects in his possession he should take with him without exceeding the capacity of the boat.

Jonah was able to come up with a short list of the 12 objects most valuable to him—which is still too many to take on board. Now he needs to figure out which combination of those objects he should choose so that their total worth (sum of assigned values) to him will be maximized without exceeding the capacity of his boat.

The Strategy

The Jonah's ark problem is a variation of what is known in mathematics as the *knapsack problem*:

> Let n be the number of objects one has to choose from. Let $\boldsymbol{V} = [v_1, v_2, \ldots, v_n]$ be the list of values (worth) of those objects and $\boldsymbol{W} = [w_1, w_2, \ldots, w_n]$ be the list of weights of those objects. Also, let $\boldsymbol{W}_{max}$ be the maximum weight that the knapsack can carry. The goal is to find a subset of m objects so

that the sum of values for that subset is maximized while ensuring that the sum of corresponding weights remains ≤ W_{max}.

We'll leverage the genetic algorithm to address this problem. It's evident that we'll have to make changes to the problem definition part of the code. First, given that Jonah now has a choice among 12 distinct objects, we'll set the number of chromosomes to 12. Each gene in the chromosome will assume a binary value, where 1 signifies the inclusion of an object in the solution and 0 denotes its exclusion. We'll also calculate the fitness of a solution differently based on which objects are included and their respective values and weights. I'll explain this further when we discuss the related code segment.

One important consideration is the composition of the initial population. We need to ensure that the initial population has some diversity. If all genes are randomly assigned, we might get a population with zero fitness. This would make crossover useless, and we would be relying solely on mutation, which is a very slow process. Therefore, while initiating the population, we'll force each member to have a nonzero fitness.

Before we start coding, we need to address a few technical considerations. First, we'll adopt a 0-1 approach to solve this problem, meaning we'll either include an object or completely exclude it in the solution. We're not allowed to take a fraction of an object (and a fraction of its value). Second, we assume that we have only one copy of each object, so we cannot repeat any object in our solution. Third, we assume that we have only one knapsack to fill.

The Code

We developed a fully functional genetic algorithm program in Project 33. For the most part, we'll reuse that code and make a few adjustments needed to describe and solve the knapsack problem (or the Jonah's ark problem).

Problem Definition and Global Parameters

This code segment is composed of an import statement, data class declarations, the creation of a list of items to choose from, global parameters, and the creation of mutable lists to track population states for both the current and the next generations, as well as the best solutions from each generation.

```
import kotlin.math.roundToInt

// Define required data classes.
data class Solution(val chromosome: IntArray, val fitness: Int)
data class Item(val value: Int, val weight: Int)

// Define the basket of items.
❶ private val items: List<Item> = listOf(
    Item(75, 15),
    Item(55, 32),
    Item(50, 30),
    Item(68, 43),
    Item(62, 54),
```

```
        Item(45, 38),
        Item(68, 62),
        Item(84, 85),
        Item(87, 87),
        Item(95, 83),
        Item(35, 21),
        Item(63, 53)
    )
    val chromosomeLength = items.size
❷   val maxWeight = 175

    // global parameters and declarations
    val POP_SIZE = 25
    val MAX_GEN = 30
    val ELITISM = 0.1
    val eliteSize = (POP_SIZE * ELITISM).toInt()

    // Limit the mutation threshold value to three decimal places.
❸   val MUTATION_THRESHOLD =
            ((1.0/chromosomeLength)*1000.0).roundToInt() / 1000.0

    val population: MutableList<Solution> = mutableListOf()
    val nextgen: MutableList<Solution> = mutableListOf()
    val bestSolutions: MutableList<Solution> = mutableListOf()
```

The code segment begins with a single import statement for the `roundToInt()` method that we'll use shortly. We then define two simple data classes, `Solution` and `Item`, which are used to create individual members of the population and objects with their key attributes (value and weight). Notice that we're creating the chromosome as an integer array and not as a string, as in Project 33.

NOTE *Depending on your IDE, you might encounter a "weak warning" while declaring the first data class of this project (`Solution`). This is because we're using a property with an `Array` type (`chromosome`) in a data class (`Solution`). While this warning indicates potential issues for certain use cases, it does not apply to the problems discussed in this chapter and the next. If you find the warning bothersome, an alternative approach would be to use regular classes instead of data classes. In that case, you can manually add necessary custom methods that a data class generates automatically, such as `copy()` and `toString()`. I encourage you to experiment with this approach as a further learning opportunity.*

Next, we create a `List` of type `Item` with the 12 objects ❶. The capacity limit for the boat (`maxWeight`) is set to 175 units ❷ (we'll assume this is in addition to Jonah's own weight).

Given the relatively small number of objects to choose from, we've set the population size (`POP_SIZE`) to 25 and the number of generations (`MAX_GEN`) to a modest 30. Elitism has been set to 0.1, or 10 percent. The `MUTATION_THRESHOLD` value is set a bit differently (its value is rounded off to three significant digits after the decimal point) ❸, but it still complies with the rule of thumb.

Note that the mutation threshold can be rounded to a few decimal places without affecting the results. This can speed up the calculations for more complex problems that need larger populations and longer runs to converge.

The last three lines of code create three mutable lists to store members of the current and next generations and the set of best solutions picked from successive generations.

Initializing Population and Fitness Evaluation

This section differs in just a few ways to what we developed for Project 33.

```
fun initPopulation() {
    // Initialize a population of valid solutions (of nonzero fitness).
    // Each solution is represented by a chromosome.

    for (person in 0 until POP_SIZE) {
      ❶ val chromosome = IntArray(chromosomeLength)

        var not_done = true
      ❷ while (not_done) {
            for (gene in 0 until chromosomeLength) {
                chromosome[gene] = (0..1).random()
            }
            val fitness = getFitness(chromosome)
          ❸ if (fitness > 0) {
                population += Solution(chromosome, fitness)
                not_done = false
            }
        }
    }

    // Sort population (in place) in descending order.
    population.sortByDescending {it.fitness}

    println("\nBest solution from initial population:")
    print(population[0].chromosome.contentToString())
    println(" " + (-population[0].fitness).toString())
    println("\n... initPopulation done ...\n")
}

fun getFitness(chromosome: IntArray): Int {
    // Get sum of values and weights.
  ❹ val sumValue  = (chromosome.zip(items) {c, item -> c * item.value}).sum()
  ❺ val sumWeight = (chromosome.zip(items) {c, item -> c * item.weight}).sum()

    return if (sumWeight <= maxWeight) sumValue else 0
}
```

Within the `initPopulation()` function, we first create each chromosome as an integer array ❶. This is because we're allowing only binary gene values (0 or 1) in individual chromosomes. Initially, all the genes will be set to 0 while the chromosome is initialized. We then randomly change these

values to 1 and 0 inside a `while` loop ❷. Further, we add only solutions that have nonzero or positive fitness values to the initial population ❸. This will help us get started with a better set of chromosomes and avoid a situation where all initial solutions have zero fitness values, which is difficult to improve on!

The remaining part of the function is the same as before—we're sorting the initial population to get it ready for elitism inside the `runGA()` function and printing out the current best solution from the initial population.

The helper function `getFitness()` receives a chromosome as its parameter and evaluates its fitness. It calculates the fitness as the weighted sum of values (`sumValue`), where weights are the genes from the chromosome ❹. It also calculates the weighted sum of weights as `sumWeight` ❺. If the sum of weights $\leq W_{max}$, the function returns the chromosome's fitness; otherwise, 0 is returned.

As mentioned earlier, we must ensure that both `sumWeight` $\leq W_{max}$ and `sumValue` > 0. We enforce the former condition in this function. The latter is enforced inside the `while` loop of the `initPopulation()` function. A chromosome is used only if its fitness, as returned by the `getFitness()` function, is greater than zero ❸.

The Driver Function

We likewise need to make only minor changes to this part of the code, which was developed for Project 33. First, we'll delete the termination condition at the beginning. For knapsack problems, the optimal solution is generally unknown beforehand. We have to run the code several times to get a sense of what the best solution might be. Second, we'll now save the best solutions from all generations in a list and pick the best overall solution from that list as the potential optimal solution.

Here is the revised code for the `runGA()` function:

```
fun runGA() {
    // Run the algorithm for a specified number of generations.
    for (generation in 1 .. MAX_GEN) {

        // Step 1: Implement elitism.
        selectElites()

        // Step 2: Implement crossover and mutation.
        for (i in eliteSize until POP_SIZE) {
            // Select parents for crossover.
            val parent1 = tournament()
            val parent2 = tournament()

            // Produce a child by using crossover and mutation.
            val child = crossover(parent1, parent2)
            // Add child to nextgen.
            nextgen += child
        }
```

```
            // Step 3: Transfer nextgen to the current population.
            for (i in nextgen.indices)
                population[i] = nextgen[i].copy()

            // Step 4: Clear nextgen for the next iteration.
            nextgen.clear()

            // Step 5: Sort the population in descending order (in place).
            population.sortByDescending {it.fitness}

            // Step 6: Add the fittest solution to bestSolutions.
        ❶ bestSolutions += population[0]

            // Step 7 (optional): Print the fittest solution.
        ❷ printSolution(generation, population[0])
        }
    }
```

Apart from deleting the termination condition based on the fitness value, both of the revisions to the code are at the end of the code segment. First, the fittest solution from each generation is now added to the mutable `bestSolutions` list ❶. Second, we've added a new print function called `printSolution()` ❷ to tidy up the printing without adding clutter to `runGA()`. This function simply formats and prints the generation number along with the chromosome and fitness of the fittest solution for each generation.

```
fun printSolution(generation: Int, solution: Solution) {
    val str1 = "%04d".format(generation).padEnd(10, ' ')
    val (c, f) = solution
    val str2 = c.contentToString()
    val str3 = f.toString().padStart(6, ' ')
    println(str1 + str2 + str3)
}
```

This function prints a line composed of three substrings. The first substring represents the generation or iteration number. We assign 10 character spaces for this, of which 4 are allocated for displaying the number; the remaining spaces will be added after the number as padding (white spaces). The second substring simply contains the sequence of 12 genes converted into a string. The third substring contains the fitness value. We assign six spaces for the number, of which up to three will be used to display the fitness value; the remaining spaces will be added as padding in front of the characters displaying the fitness value.

The Operator Functions

We'll skip discussing the `selectElites()` and `tournament()` functions as no changes are required to use them in this example (you can copy them from Project 33). However, we have a different chromosome structure for the

knapsack problem and additional constraints to satisfy. This means we'll have to make changes to the `crossover()` and `mutation()` functions.

```
fun crossover(parent1: Solution, parent2: Solution): Solution {
    // random single-point split and crossover
  ❶ val split = (1 until chromosomeLength).random()

    // Use copyOfRange() to extract elements from an array.
    // .copyOfRange(a,b): a = start index, b = not inclusive
    val arr1 = parent1.chromosome.copyOfRange(0, split)
    val arr2 = parent2.chromosome.copyOfRange(split, chromosomeLength)

  ❷ val newChromosome = arr1 + arr2

    // Apply in-place mutation to the new chromosome.
  ❸ mutation(newChromosome)

  ❹ return Solution(newChromosome, getFitness(newChromosome))
}

fun mutation(newChromosome: IntArray) {
    // Carry out in-place mutation.
    for (i in 0 until chromosomeLength) {
        if ((0..1000).random()/1000.0 <= MUTATION_THRESHOLD) {
            // Simplest way to flip values between 0 and 1 is i = 1 - i.
          ❺ newChromosome[i] = (1 - newChromosome[i])
        }
    }
    // nothing to return
}
```

As before, the `crossover()` function starts with randomly locating a point to split the chromosomes ❶. We use the `copyRangeOf()` method to copy different ranges of genes from parent 1 and parent 2 because the chromosomes are of type `IntArray` instead of `String`. The new chromosome is created by combining the first part of parent 1 with the second part of parent 2 (creating one child per crossover) ❷.

Next, we call the `mutation()` function to mutate this newly created chromosome in place ❸. Since arrays are passed by reference (memory location) rather than by value, all the genetic alterations will be applied directly to the selected elements of `newChromosome`, and we don't need to return a separate mutated chromosome to the calling function.

Once this step is complete, a new child (`Solution`) is created and returned by using the newly created chromosome and its fitness ❹.

Finally, the `mutation()` function scans every gene in the chromosome and applies mutation to a gene by comparing a random number between 0 and 1 with the `MUTATION_THRESHOLD`. When the condition is met, it flips the value of the gene from 0 to 1 or vice versa ❺.

The main() Function

The `main()` function for this project is similar to that of Project 33, with one additional call to `printBestSolution()` to print the best overall solution. Here is the code snippet including the print function:

```
fun main() {
    println("\n*** Solving the 0-1 knapsack problem " +
            "using the genetic algorithm ***\n")
    println("Population size: $POP_SIZE, Generations: $MAX_GEN")
    println("Number of items to pick from: $chromosomeLength")
    println("Mutation threshold: $MUTATION_THRESHOLD")

    // Initialize the population.
    initPopulation()
    // Run the genetic algorithm.
    runGA()
    // Print the best overall solution.
    printBestSolution()
}

fun printBestSolution() {
  ❶ bestSolutions.sortByDescending { it.fitness }
    println("\nBest solution found after $MAX_GEN generations:")

  ❷ val (chromosome, fitness) = bestSolutions[0]
  ❸ val sumWeight = (chromosome.zip(items)
                        {c, item -> c * item.weight}).sum()
    println(bestSolutions[0].toString())
    println("Sum of weights: $sumWeight   Sum of values: $fitness")
}
```

The `main()` function is very short. It starts with printing key global parameters, then calls `initPopulation()` to create the initial population of solutions and the driver function `runGA()` to run the genetic algorithm that we've customized for the knapsack problem. Finally, it prints the best overall solution by calling the `printBestSolution()` function.

Next, the `bestSolutions` list is sorted in descending order so that the first item represents the best overall solution ❶. The properties of this item are then deconstructed as chromosome and fitness ❷. Finally, the sum of weights of the objects in this optimal (or near-optimal) solution is calculated as a weighted sum, the weights being the individual gene values (0, 1) ❸. The last line prints the sum of weights and fitness for the best overall solution.

The Result

The following sample output from a run of the code provides an indication of what to expect when you run the code:

```
*** Solving the 0-1 knapsack problem using the genetic algorithm ***

Population size: 25, Generations: 30
Number of items to pick from: 12
Mutation threshold: 0.083

Best solution from initial population:
Solution(chromosome=[1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1], fitness=285)

... initPopulation done

0001      [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1]   285
0002      [1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1]   285
0003      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0004      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0005      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0006      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0007      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0008      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0009      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0010      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0011      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0012      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0013      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0014      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0015      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0016      [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0]   296
0017      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0018      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0019      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0020      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0021      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0022      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0023      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0024      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0025      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0026      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0027      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0028      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0029      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311
0030      [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]   311

Best solution found after 30 generations:
Solution(chromosome=[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1], fitness=311)
Sum of weights: 173   Sum of values: 311
```

The optimal solution for Jonah is to choose objects 1, 2, 3, 4, and 12, which will give Jonah a combined value of 311 units. The total weight of the optimal choice is 173 units, just shy of the maximum allowable weight of 175 units.

How do we know that no better solutions exist? In this case, you can verify the solution by using a brute-force approach—generating all possible combinations and checking corresponding sums of values and weights. I encourage you to search online for relevant tools or code examples you can use to confirm that 311 is indeed the best value Jonah can get under the circumstances.

Again, remember that the genetic algorithm is a stochastic algorithm, meaning no two runs will produce identical results. Moreover, nothing guarantees that for a given set of parameter values, the algorithm will consistently converge to the optimal solution every time you run the program. You may have to run the program multiple times or adjust the program parameters to eventually locate the optimal solution.

On the other hand, genetic algorithms can help solve real-world combinatorial problems with hundreds of decision variables, where checking all possible combinations for the global optimal solution is impractical or impossible. They take far less time and require significantly less computational effort to generate near-optimal solutions.

EXERCISE

In the previous two projects, we've used the tournament selection method for choosing parents for reproduction. We also discussed a couple of other selection methods: roulette wheel and rank-based selection.

For this exercise, develop a new function called `rouletteWheel()` based on the roulette wheel selection algorithm, and incorporate that in the `runGA()` function to solve the Jonah's ark problem. Use the same problem-specific parameters used in Project 34.

Based on the results, reflect on whether using roulette wheel selection results in a quicker convergence to the global optimum. In this context, quicker means finding the global optimum with fewer iterations.

Project 35: Optimize a Multivariate Function with the Genetic Algorithm

In this final project, we'll learn how to apply the genetic algorithm to multivariate function optimization. The only requirement for the function is that it be defined in terms of the independent variables within the decision space. In contrast to gradient-based algorithms, this function does not have to be smooth or differentiable.

We'll use a sufficiently challenging two-dimensional function known as the Eggholder function, defined by two independent variables: x_1 and x_2.

$$f(\boldsymbol{x}) = -(x_2 + 47)\sin\left(\sqrt{\left|\frac{x_1}{2} + x_2 + 47\right|}\right) - x_1\sin\sqrt{\left|x_1 - (x_2 + 47)\right|} \qquad (8.2)$$

We'll find the minimum value of this function in the decision space defined as follows:

$$x_i \in [-512, 512] \text{ for } i = 1, 2$$

As shown earlier in Figure 8-3, the Eggholder function has a very complex shape with numerous peaks and troughs. Because of this, deterministic gradient-based algorithms will have a hard time finding the global minimum. Deterministic search attempts will usually get stuck at local optima unless we use a hybrid approach that incorporates some random search features. In contrast, given enough diversity (population size) and time (number of generations), a genetic algorithm can locate the global minimum fairly quickly for this problem.

The Strategy

To implement function optimization in a genetic algorithm, the decision variables are treated as individual genes, meaning the two-variable Eggholder function will have two genes. This time, however, these genes will be represented as real numbers, including fractions, instead of characters (as in Project 33) or binary values (as in Project 34).

We also need to address the fact that this is a minimization problem, not a maximization problem as in the previous two projects. In those cases, the goal was to find a solution with the greatest fitness, whereas now we want to find the solution with the smallest fitness. Fortunately, we can easily handle this case by multiplying the objective function by –1. This adjustment allows us to continue using the existing code developed for maximization problems. Notably, if we were to switch back to a maximization problem, we could use the same code without needing to multiply the objective function by –1.

Finally, we need a new way to implement mutation for function optimization. In previous projects, we introduced mutation by randomly replacing a character or a binary value, but that approach does not make sense for a real number. Digits in a real number cannot be arbitrarily replaced, as their relative position within the number has additional significance. Therefore, for real-valued genes, mutation is introduced as a small noise that is randomly added to or subtracted from the genes (we still use a probability threshold). The magnitude of the noise is calculated as a small fraction of the range for a specific gene (decision variable). By doing so, we can properly scale the magnitude of the noise or mutation without having to worry about the underlying units used for the corresponding decision variable in the function.

The Code

The code for function optimization has the same general structure as Projects 33 and 34. It is worth reiterating that for minimization of the objective function, we'll have to multiply it by –1, whereas for maximization no alteration to the objective function is needed.

Problem Definition and Global Parameters

This code segment includes the import block, a data class, global parameters, and several collections of mutable lists:

```
// import block
import kotlin.math.sin
import kotlin.math.sqrt
import kotlin.math.abs
import kotlin.math.pow
import kotlin.math.min
import kotlin.math.max
import kotlin.random.Random

// Define required data classes.
data class Solution(
  ❶ val chromosome: DoubleArray,
    val fitness: Double)

// global parameters and declarations
❷ val getFitness = :: eggHolder

❸ val chromosomeLength = 2    // number of independent variables
  val bounds = arrayOf(doubleArrayOf(-512.0, 512.0),
                       doubleArrayOf(-512.0, 512.0))
  val varRange = doubleArrayOf(bounds[0][1] - bounds[0][0],
                               bounds[1][1] - bounds[1][0])
  val POP_SIZE = 100
  val MAX_GEN = 200
❹ val MUTATION_THRESHOLD = 0.5    // On average, 1 of 2 genes will mutate.
  val MUTATION_FACTOR = 0.02
  val ELITISM = 0.1
  val eliteSize = (POP_SIZE * ELITISM).toInt()

  val population: MutableList<Solution> = mutableListOf()
  val nextgen: MutableList<Solution> = mutableListOf()
  val bestSolutions: MutableList<Solution> = mutableListOf()
```

The code segment begins by importing the necessary math functions to calculate the function value or fitness. Remember to import only the methods you need, instead of importing all of them using an `import kotlin.math.*` statement. We then declare the chromosome to be of type `DoubleArray` ❶ to deal with real-coded genes.

We also define `getFitness` as a variable and assign it a reference to the Eggholder function ❷. This approach allows us to define other functions later. And to use those, we simply need to reassign `getFitness` to the desired function.

Because the Eggholder function is a function of two independent variables (x_1 and x_2), we will need two real-coded genes per chromosome ❸.

The next two lines set the bounds for the decision variables and calculate the range for each. For real-coded genes, the magnitude of mutation is

typically set to a small value relative to the range of the decision variables. This approach has the benefit of being scale independent.

The remaining part of the code segment is similar to that of previous projects. This time, the population is composed of 100 individuals (`POP_SIZE`), and it will evolve for 200 generations (`MAX_GEN`). The `MUTATION_THRESHOLD` is now set to 0.5 ❹, in line with the practice of setting the mutation probability equal to the inverse of the number of genes.

Of course, we could've tried many other combinations of parameter values. The values used in this code segment were chosen based on a number of trials to ensure that the global minima for the Eggholder function can be found quickly.

Initializing Population and Fitness Evaluation

The general organization of this code snippet is very similar to that of the two previous projects, with a few problem-specific adjustments:

```
fun initPopulation() {
    // Initialize a population of valid solutions (genes within bounds).
    // Each solution is represented by a chromosome.

    for (person in 0 until POP_SIZE) {
      ❶ val x = DoubleArray(chromosomeLength)
        for (i in 0 until chromosomeLength) {
            // The first argument is inclusive, but the second one is not.
            // It's possible to add a small bias term to the upper bounds.
          ❷ x[i] = Random.nextDouble(bounds[i][0], bounds[i][1])
        }
        population += Solution(x, getFitness(x))
    }

    // Sort the population (in place) in descending order.
    population.sortByDescending {it.fitness}

    println("\nBest solution from initial population:")
  ❸ println(population[0].toString())
    println("\n... initPopulation done ...\n")
}

fun eggHolder(x: DoubleArray): Double {
    val c1 = (x[1] + 47)
    val c2 = sin(sqrt(abs(0.5 * x[0] + c1)))
    val c3 = x[0] * sin(sqrt(abs(x[0] - c1)))

    // Multiply by -1 ONLY for minimization problems.
  ❹ return -1.0 * (-c1 * c2 - c3)
}
```

We generate the chromosome as a `DoubleArray` with two elements ($x[0]$ will be gene 1, and $x[1]$ will be gene 2) ❶. We then initialize the genes randomly, ensuring they stay within their respective bounds (defined by the decision space) ❷. The rest of the code segment assigns the solutions to the

mutable list `population`, sorts the population in descending order, and prints the best solution from the initial population ❸.

As mentioned earlier, this code doesn't include a `getFitness` function; instead, we had pointed `getFitness` to the `eggHolder()` function, which returns the value of the objective function (fitness). For convenience, we've broken down the objective function given by Equation 8.2 into three parts, which are later combined to calculate the fitness value ❹. Notice that we're multiplying the fitness by –1 before returning the value to `getFitness`. Doing so enables us to use the code developed for maximization problems to solve a minimization problem.

We'll skip reviewing the `runGA()` function as it is identical to the one used for Project 34. The same goes for the `selectElites()` and `tournament()` functions. Therefore, we'll move straight to the `crossover()` and `mutation()` functions.

Operator Functions for Crossover and Mutation

We'll now move on to the two key operator functions performing crossover and mutation to examine the differences introduced for the real-coded genes.

```
fun crossover(parent1: Solution, parent2: Solution): Solution {
    // Select a random weight within (0-1).
    // This could be generated separately for x- and y-components.
  ❶ val s = (0..1000).random()/1000.0

    // Generate randomly weighted genes.
    var x1 = parent1.chromosome[0]*s + parent2.chromosome[0]*(1-s)
    var x2 = parent1.chromosome[1]*s + parent2.chromosome[1]*(1-s)

    // Check that new genes stay within bounds (decision space).
    x1 = min(max(x1, bounds[0][0]), bounds[0][1])
    x2 = min(max(x2, bounds[1][0]), bounds[1][1])

    // Compose the new chromosome.
  ❷ val xNew = doubleArrayOf(x1, x2)
    // Mutate the new chromosome.
  ❸ mutation(xNew)

  ❹ return Solution(xNew, getFitness(xNew))
}

fun mutation(xNew: DoubleArray) {
    for (i in 0 until chromosomeLength) {
        if (((0..1000).random() / 1000.0) <= MUTATION_THRESHOLD) {
            // Get the random sign factor.
          ❺ val sign = if ((0..100).random()/100.0 <= 0.5) -1 else 1
          ❻ xNew[i] += sign * varRange[i] * MUTATION_FACTOR
            xNew[i] =  min(max(xNew[i], bounds[i][0]), bounds[i][1])
        }
    }
    // nothing to return
}
```

The purpose of the crossover function for real-valued genes remains the same: to produce a new chromosome for the child by using genetic materials from the parents. Several methods are available for creating the new chromosome or genes. In this example, we're using a random-weighted scheme based on a randomly selected value *s* between 0 and 1 ❶. (If we used a fixed weight, `s=0.5`, that would be equivalent to using the arithmetic average of the gene values from two parents to create a new gene.)

We use the weighted average scheme to generate two new genes (`x1` and `x2`) and ensure that these values are within the bounds of the decision variables. We then compose the new chromosome `xNew` as a `DoubleArray`, with two genes as its elements ❷.

Next, we call the `mutation()` function to mutate this newly created chromosome in place ❸. Since arrays are passed by reference (memory location), mutations can be directly applied to the elements (genes) of the array, and we don't need to return anything to the calling function. Once mutation is applied, a new child (`Solution`) is created and returned using the newly created chromosome and its fitness ❹.

The `mutation()` function, similar to Project 34, scans each gene and mutates it if a random number between 0 and 1 is less than `MUTATION_THRESHOLD`. It randomly picks the sign of the mutation (positive or negative) ❺ and calculates the value as the sign times the decision variable's range times `MUTATION_FACTOR` ❻. It also ensures that the mutated genes are within the bounds of the corresponding decision variables.

Before we proceed to the `main()` function, we need to adjust the `printSolution()` function from Project 34. It now takes a solution with a chromosome of type `DoubleArray` instead of an `IntArray`. Use the following updated function in your code:

```
fun printSolution(generation: Int, solution: Solution) {
    val str1 = "%04d".format(generation).padEnd(10, ' ')
    val (c, f) = solution
    val str2 = "%5.7f".format(c[0]).padEnd(14, ' ')
    val str3 = "%5.7f".format(c[1]).padEnd(14, ' ')

    // Multiply f (fitness) by -1 for minimization (for display only).
    val str4 = "%5.4f".format(-f)

    println(str1 + str2 + str3 + str4)
}
```

You can review this code and compare it with the program output as an exercise, since you are familiar with these helper functions.

The main() Function

The code snippet for the `main()` function, including the `printBestSolution()` function, is likewise similar to the `main()` functions in previous projects.

```
fun main() {
    println("\n*** Real-valued function optimization using " +
            "the genetic algorithm ***\n")
    println("Number of dimensions: $chromosomeLength")
    println("Population size: $POP_SIZE, Generations: $MAX_GEN")
    println("Elitism: $ELITISM")
    println("Mutation threshold: $MUTATION_THRESHOLD")
    println("Mutation factor: $MUTATION_FACTOR")

    // Initialize the population.
    initPopulation()
    // Run the genetic algorithm.
    runGA()
    // Print the best overall solution.
    printBestSolution()
}

fun printBestSolution() {
    // Sort the bestSolutions to get the best-so-far solution.
    bestSolutions.sortByDescending {it.fitness}
    println("\nBest solution found after $MAX_GEN generations:")

    // Deconstruct for printing with formatting.
    val (chromosome, fitness) = bestSolutions[0]

    // Format and print the best-so-far properties.
    for (i in chromosome.indices) {
        print("chromosome[$i]: ")
        println("%5.8f".format(chromosome[i]))
    }
    println("Fitness: " + "%5.5f".format(-fitness))
}
```

The `main()` function starts by printing key global parameters. It then calls `initPopulation()` to initialize the population and launches the driver function `runGA()` to carry out function minimization using a genetic algorithm.

In the `printBestSolution()` function, we format and print the two real-valued genes on the same line by using a `for` loop. Finally, we print the negative fitness value to get the correct sign for the minimum fitness.

The Result

We're now ready to run the code and examine the results. If you use the same global parameter values that I have used for this project, you are likely to get the global optimal solution within five to seven attempts. Let's look at a sample output:

```
*** Real-valued function optimization using a genetic algorithm ***

Number of dimensions: 2
Population size: 100, Generations: 200
Elitism: 0.1
```

```
Mutation threshold: 0.5
Mutation factor: 0.02

Best solution from initial population:
[439.9192360610284, 466.3475628354653] -809.6304961876202

... initPopulation done ...

0001      439.91923606   466.34756284   -809.63050
0002      439.91923606   466.34756284   -809.63050
0003      439.91923606   466.34756284   -809.63050
0004      421.26042117   431.81770471   -838.23597
0005      421.26042117   431.81770471   -838.23597

--snip--

0190      512.00000000   404.23184036   -959.64066
0191      512.00000000   404.23184036   -959.64066
0192      512.00000000   404.23184036   -959.64066
0193      512.00000000   404.23184036   -959.64066
0194      512.00000000   404.23184036   -959.64066
0195      512.00000000   404.23184036   -959.64066
0196      512.00000000   404.23184036   -959.64066
0197      512.00000000   404.23184036   -959.64066
0198      512.00000000   404.23184036   -959.64066
0199      512.00000000   404.23184036   -959.64066
0200      512.00000000   404.23184036   -959.64066

Best solution found after 200 generations:
chromosome[0]: 512.00000000
chromosome[1]: 404.23184036
Fitness: -959.64066
```

The first section of the results shows the global parameters used for solving this problem—population size (100) and number of generations (200). Elitism is set to 0.1, or 10 percent. We used a mutation threshold of 0.5 because we have two genes, but we could have used a lower threshold if this threshold caused the best solutions to oscillate rather than converge. Due to the presence of many near-optimal solutions within the decision space of this problem, a higher-than-usual mutation threshold may have helped the algorithm to get out of local minima and explore other regions.

The initial best fitness value was –809.63, which is not that close to the global minimum of –959.64 located after 117 iterations (not shown in the partial output above). Once this value was reached, the best solution remained unchanged until the program ended after completing the maximum number of iterations.

We can see from the last part of the results that the optimal solution is located at $x_1 = 512.0$ and $x_2 = 404.23$. Figure 8-9 shows this point as a white half-circle near the top-right corner of the contour plot of the Eggholder function.

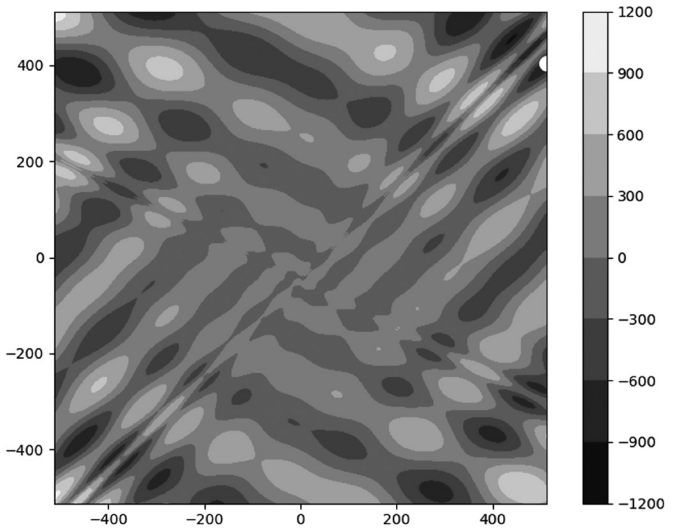


Figure 8-9: The contour plot of the Eggholder function

In this case, the global solution is literally on the right-hand boundary of the decision space in Figure 8-10. The grayscale bar indicates that the darker regions are troughs and the lighter regions are peaks. Clearly, the fitness values are close to the global minima in many cases (based on the darkness of the shade). This is why it is so difficult to find the global minima for the Eggholder function.

Figure 8-10 shows the convergence pattern for this problem. The fitness value improves in a stepwise manner with the number of generations (iterations) until it reaches the global minima.

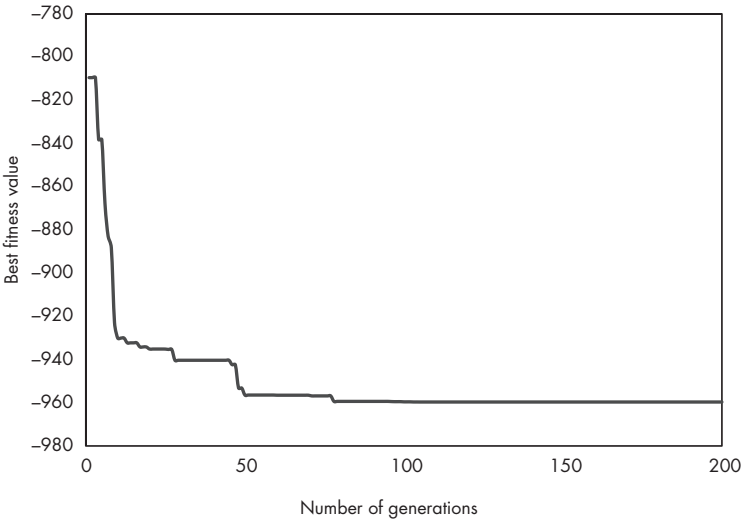


Figure 8-10: The convergence pattern for the Eggholder function using the genetic algorithm

We'll revisit this problem using particle swarm optimization (another NIA) and draw the corresponding convergence pattern in the next chapter. You'll see that while both methods are capable of identifying the global optima, particle swarm optimization will do that much quicker!

Stopping Condition for Genetic Algorithms

When solving real-world optimization problems, we often lack knowledge of the global optimal solution. Consequently, we cannot directly use it as a stopping condition. To address this challenge, we can employ several strategies. In this section, I'll discuss commonly used approaches for defining stopping conditions in genetic algorithms.

First, the stopping condition can be implemented as the maximum number of generations, as we've done for all three projects on the genetic algorithm. In general, you wouldn't know how many iterations it might take to solve a previously unsolved problem. This will depend on the nature of the problem, the global parameter values, and the specific schemes used for various operator functions. You'll have to gradually adjust the number of iterations (along with other parameters) to find a combination of values that works for the problem at hand. Interestingly, we can use the genetic algorithm to find optimal combinations of global parameters. In the field of deep learning, the genetic algorithm has been used to optimize global parameters and quickly train neural networks that produce high-quality results.

Second, you could stop the algorithm from iterating if the best solution's fitness does not show noticeable improvements for several generations (for example, very little or no improvement over the past 30 or 50 generations). This will require additional coding to track improvements dynamically, but this can be a strategy to let the algorithm stop automatically, even when the maximum number of iterations has not been reached.

Third, for certain types of problems, you may be able to set a target for the fitness and have the program terminate once that target is reached (recall that we had a text-matching target for the first project). When the target is difficult to reach, you could also terminate the program when a predetermined percentage of that target is reached (instead of matching the target exactly).

A lot more could be said about genetic algorithms, and researchers are frequently developing new adaptive or hybrid strategies and finding new applications. If you are interested in the state of the art, I suggest reviewing recent journal articles on the application of the genetic algorithm in your field of interest.

EXERCISE

One well-known crossover strategy used in genetic algorithms is called blend crossover (BLX). Given two genes x_1 and x_2 from parent 1 and parent 2, respectively, and assuming that $x_1 < x_2$, BLX randomly creates a gene in the range of $[x_1 - \alpha * (x_2 - x_1), x_2 + \alpha * (x_2 - x_1)]$. The value of α lies between 0 and 1 (inclusive). When $\alpha = 0$, this strategy becomes the same as the random weighted scheme used in Project 35. When $\alpha = 0.5$, the random value will be chosen from a range of $2 * (x_2 - x_1)$. This is the recommended value based on empirical results. Finally, $\alpha = 1$ will result in choosing the random value from a range of $3 * (x_2 - x_1)$.

For this exercise, develop a new function called `crossoverBLX()` based on this scheme (use $\alpha = 0.5$). Solve the Eggholder problem using this new crossover function for the set of parameters used in Project 35. Does the use of BLX result in a quicker convergence to the global optimum compared with the random weighted scheme?

In Project 35, we revised the code developed for function maximization to handle a minimization problem. However, changing the code back and forth can easily lead to errors. Therefore, in the next exercise you will develop new code to directly handle function minimization problems.

EXERCISE

Modify the previously developed code so that it can directly handle a minimization problem. As a hint, you'll have to change the selection method that chooses the solution with the smallest fitness instead of choosing the one with the greatest fitness. Additionally, whenever you sort the population, such as to facilitate elitism or to identify the best solution from a generation, you must sort in ascending order and pick the first solution which will have the smallest fitness value.

If you want to use the roulette wheel scheme for a minimization problem, you'll need to add code to deal with negative fitness values that may arise. This is because relative fitness values, which are calculated from actual fitness values, are used as proxy probabilities in the roulette wheel scheme, and probabilities cannot be negative. This issue can be handled by rescaling the fitness values so that the smallest fitness value becomes zero (or a small positive number). One alternative to using the roulette wheel or tournament selection is to use a rank-based selection where the chromosome with the smallest fitness value will be assigned the highest rank.

Happy coding!

Summary

In this chapter, you explored the fascinating world of nature-inspired algorithms, computational methods that mimic natural phenomena to solve complex problems. One key feature of these algorithms is that they are stochastic in nature: they exploit built-in randomness to tackle problems that are intractable or too complex for conventional methods. You learned about the benefits and challenges of using nature-inspired algorithms and focused on one of the most popular and powerful examples: the genetic algorithm.

The genetic algorithm is inspired by the process of natural evolution and uses a population of candidate solutions that undergo selection, crossover, and mutation to find the best solution for a given problem. You learned several ways to implement these operations and adjust the parameters of the algorithm to achieve the best performance. You also applied genetic algorithms to three different projects in order to:

- Generate a target string from a random population of characters
- Maximize the value of items in a knapsack with a limited capacity
- Find the global optimum solution for a real-valued and highly complex multivariate objective function

In addition, you completed a set of exercises that cover additional techniques for the crossover operation and dedicated methods for solving minimization problems directly. By the end of this chapter, you gained a solid understanding of the theory and practice of genetic algorithms, and how they can be used to solve various types of optimization problems.

Resources

Brownlee, Jason. *Clever Algorithms: Nature-Inspired Programming Recipes*. Electronic version, June 16, 2012. *https://github.com/clever-algorithms/CleverAlgorithms*.

Buontempo, Frances. *Genetic Algorithms and Machine Learning for Programmers*. Raleigh, NC: The Pragmatic Bookshelf, 2019.

Gen, Mitsuo, and Runwei Cheng. *Genetic Algorithms and Engineering Optimization*. New York: John Wiley & Sons, 2000.

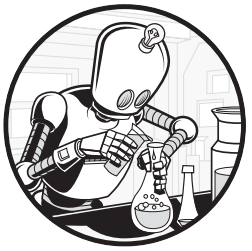
Goldberg, David. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley Professional, 1989.

Haupt, Randy L., and Sue Ellen Haupt. *Practical Genetic Algorithms*. 2nd ed. Hoboken, NJ: John Wiley & Sons, 2004.

Yang, Xin-She. *Nature-Inspired Optimization Algorithms*. 2nd ed. London: Academic Press, 2021.

9

AGENT-BASED ALGORITHMS



In this chapter, we'll continue to explore NIAs, focusing on two algorithms based on the collective behavior of social animals: particle swarm optimization and ant colony systems. These algorithms are designed for *agent-based models*, in which a swarm of simple agents work together and interact with their surroundings to create outcomes that benefit the whole colony.

We'll explain the key concepts and principles behind these algorithms and implement them using pseudocode. We'll also put these algorithms to the test by developing two Kotlin applications to solve real-world problems. The first is function minimization, which uses particle swarm optimization to find the global minimum of a given function. The second is the traveling salesman problem, which uses ant colony systems to find the shortest route that connects a large number of cities.

An Overview of Particle Swarm Optimization

The particle swarm optimization (PSO) algorithm has been used to solve a wide range of optimization problems. PSO has a few similarities with the genetic algorithm. Both methods involve working with a population (of chromosomes or particles), members of which help us look for an optimal solution until a stopping condition is met. Both methods are also stochastic and rely on underlying processes with built-in elements of randomness.

That's where the similarities end. Unlike the genetic algorithm, the PSO algorithm does not depend on genetic operators such as selection, crossover, and mutation. Instead, it is driven by autonomous agents that update their positions in the decision space based on their current and past positions, as well as the best position identified by the swarm. PSO involves a deliberate effort to continuously move toward a better solution, which is very different from the passive selection-driven upgrading of chromosomes in the genetic algorithm. Individual chromosomes do not have any ability to sense their neighborhood or make decisions to update their composition as the particles do in PSO. Further, PSO is conceptually simpler and easier to implement, involves fewer parameters, and tends to converge more quickly on the global optima, compared with the genetic algorithm.

At its core, the PSO algorithm consists of three key steps:

1. Initialize the position and velocity of the particles.
2. Proceed through the time steps, and update particle velocities based on the current velocity, the best-so-far individual position, and the global best position identified by the swarm up to that time step.
3. Update the current position by moving to a better position.

These steps are repeated for a set number of times or until a stopping condition is met.

Let $\boldsymbol{x}_i^0$ and $\boldsymbol{v}_i^0$ be the position and velocity vectors at time $t = 0$, where $i \in \mathbf{I}$ denotes the ith particle in a swarm of size N ($\mathbf{I} = [1, 2, \ldots, N]$). Also, let $\boldsymbol{x}_{max}$ and $\boldsymbol{x}_{min}$ be the upper and lower bounds for the position vector and r be a random value between 0 and 1 selected from a uniform distribution.

The first step of PSO is to initialize the position and velocity vectors as shown in Equations 9.1 and 9.2.

$$\boldsymbol{x}_i^0 = \boldsymbol{x}_{min} + r(\boldsymbol{x}_{max} - \boldsymbol{x}_{min}) \tag{9.1}$$

$$\boldsymbol{v}_i^0 = 0 \tag{9.2}$$

In Equation 9.2, for the purposes of this chapter, we set the initial velocities to zero. The alternative is to set them to small random values. In most cases, the swarm will quickly move away from the randomly assigned initial position, and the choice of initial velocity will not have a noticeable impact on the convergence rate as long as the magnitude of the velocity stays within the decision space.

The second step of PSO is to update the velocity vector $\boldsymbol{v}_i^{t+1}$ for the particles by using the following equation:

$$\boldsymbol{v}_i^{t+1} = w\,\boldsymbol{v}_i^t + \text{c1}\,\text{r1}\,(\boldsymbol{b}_i^t - \boldsymbol{x}_i^t) + \text{c2}\,\text{r2}\,(\boldsymbol{g}_s^t - \boldsymbol{x}_i^t) \tag{9.3}$$

The variables and parameters in Equation 9.3 stand for the following:

w	Inertia factor
c1	Particle memory/cognitive factor
c2	Swarm memory/social factor
r1, r2	Random values between 0 and 1 chosen from a uniform distribution
$\boldsymbol{b}_i^t$	Best position vector found by particle i up to time t
$\boldsymbol{g}_s^t$	Best position vector found by the swarm up to time t
$\boldsymbol{v}_i^t$, $\boldsymbol{v}_i^{t+1}$	Velocity vectors of particle i at time t and $t+1$, respectively
$\boldsymbol{x}_i^t$	Position vector of particle i at time t

Figure 9-1 provides a visual and more intuitive interpretation of Equation 9.3.

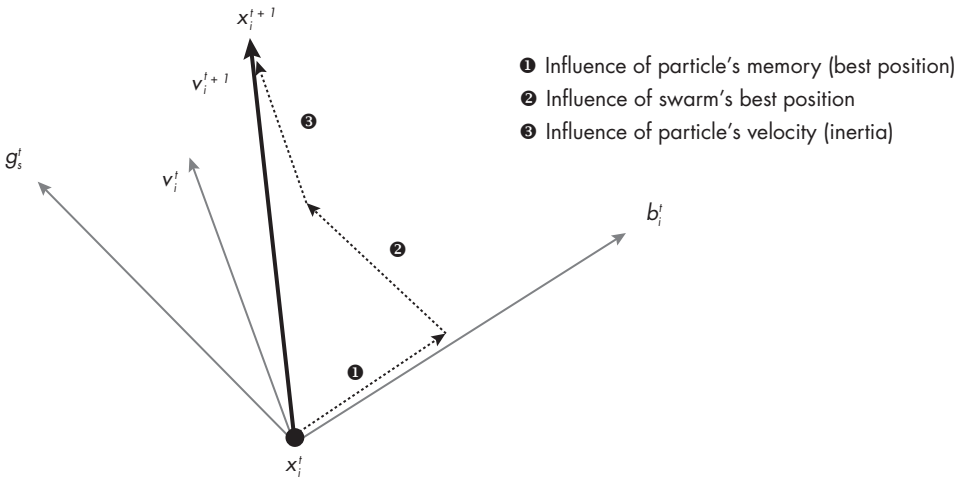


Figure 9-1: A graphical representation of the strategy for updating velocity in PSO

Let's consider an arbitrary particle i whose current position is x_i^t (for simplicity's sake, we'll consider a one-dimensional problem). Because of inertia, the particle will tend to move toward the direction of its current velocity, v_i^t. However, PSO relies on learning from both particle- and swarm-level best solutions found so far—b_i^t and g_s^t. As a result, the particle incorporates this information by moving a bit toward b_i^t ❶ and then toward g_s^t ❷, as well as toward its own velocity v_i^t ❸. Figure 9-1 shows the result of all these movements as v_i^{t+1}. Equation 9.3 captures the same movements symbolically in multiple dimensions, along with the relative weights assigned to each of these components.

The inertia factor w is typically assigned a value between 0 and 1, where a value of 0 would imply no impact of $\boldsymbol{v}_i^t$ on $\boldsymbol{v}_i^{t+1}$ and 1 would imply full

impact. It is also possible to adjust the value of w over time, which can lead to better convergence properties. This scheme is implemented by initially setting w to ≥ 1 and gradually decreasing it per Equation 9.4.

$$w^t = \mathrm{w}_{\max} - \left(\frac{t}{t_{\max}}\right) (\mathrm{w}_{\max} - \mathrm{w}_{\min}) \tag{9.4}$$

The c1 and c2 factors in Equation 9.3 are also called *acceleration coefficients*. Along with the random variables r1 and r2, these coefficients determine the degree of influence of the particle-best and swarm-best positions on the updated velocity of a particle.

As is evident from Equation 9.3, when both c1 and c2 = 0, particles will keep moving at constant velocities until they hit the boundaries of the decision space. If c1 > 0 and c2 = 0, particles will behave as if they're independent (no information gathered from the swarm). When c1 = 0 and c2 > 0, the entire swarm will move toward the best position found collectively so far.

While the optimal values for these parameters are likely to be problem specific, the commonly used values found in the literature range from 0.5 to 2.5 for both c1 and c2. It is also common practice to keep c1 equal to c2 and ensure both values are relatively small to allow for a thorough exploration of the decision space without causing *velocity explosion*, which refers to excessive velocity and large jumps across the decision space. To avoid this, an upper limit for the velocity of particle i along dimension j is set as follows (for all time steps):

$$-v_j^{max} \leq v_{i,j} \leq v_j^{max} \tag{9.5}$$

where

$$v_j^{max} = \lambda_j (x_j^{max} - x_j^{min}), \qquad \lambda_j \in (0, 1) \tag{9.6}$$

The final step for PSO is to update the position vector of particle i for the next time step $t + 1$ as follows:

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1} \tag{9.7}$$

As with velocities, the updated position vectors will also have to be checked against the specified bounds for the decision variables.

Implementing PSO for Function Minimization

We can use the PSO algorithm to either maximize or minimize a function. In this chapter, we'll apply PSO to function minimization. Consequently, when aiming to minimize a multivariate function by using this implementation, we don't need to multiply the objective function value by –1, as we did for function minimization in Chapter 8.

In contrast to the other algorithms covered in this book, the PSO algorithm involves more interlinked steps, which makes it difficult to

understand and code the algorithm without a thorough overview of the entire process. To address this, I'll provide pseudocode outlining the entire process to guide us through actual code development. *Pseudocode* is a high-level description of an algorithm or a computer program. It's written in plain language that closely resembles the structure of a programming language, but it is not meant to be executed on a computer. It allows programmers to plan out and communicate the logic of a program without getting bogged down in the details of a specific programming language.

Here's the pseudocode for the PSO algorithm. Notice that we've used boldface to mark where loops and conditional blocks begin and end, as well as to emphasize specific tasks carried out by code segments.

```
Initialize swarm:
for i = 0 .. < swarm size
        for j = 0 .. < number of dimensions
                initialize position to random values per Equation 9.1
                initialize velocity to 0 per Equation 9.2
        end for
        create pBest[i] (with same position and fitness)
        create swarm[i]
        if fitness(swarm[i]) < fitness(gBest)
                update gBest
        end if
end for

Iterate over a preset number of time steps:
for time step = 0 .. < tmax
        update w per Equation 9.4
        for i = 0 .. < swarm size
                for j = 0 .. < number of dimensions
                        update velocity per Equation 9.3
                        check velocity within bounds per Equation 9.5
                        update position per Equation 9.7
                        check position within bounds
                end for
                update fitness of swarm[i]
                if fitness(swarm[i]) < fitness(pBest[i])
                        update pBest[i]
                end if
                if fitness(swarm[i]) < fitness(gBest)
                        update gBest
                end if
        end for
end for
```

We'll follow this pseudocode closely as we develop the code for implementing the PSO algorithm in the next project.

Project 36: Optimize a Multivariate Function with a Particle Swarm

For this project, we'll revisit the Eggholder function optimization problem defined in Chapter 8, this time solving it with the PSO algorithm rather than with the genetic algorithm.

The Code

We'll develop the code for the PSO algorithm following the pseudocode provided earlier and discuss its implementation in four segments: problem definition and global parameters, initializing the swarm, the `runPSO()` driver function, and the `main()` function.

Problem Definition and Global Parameters

This segment is composed of an import block, a collection of classes, global variables and parameters, and lists to save particle states and best solutions.

```
import kotlin.math.sin
import kotlin.math.pow
import kotlin.math.abs
import kotlin.math.sqrt
import kotlin.math.min
import kotlin.math.max

❶ data class Solution(
      var pos: DoubleArray,
      var fitness: Double
  )

❷ data class Particle(
      val id: Int,
      val pos: DoubleArray,
      val vel: DoubleArray,
      var fitness: Double,
      val pBest: Solution
  )

  // problem definition
❸ val getFitness = ::eggHolder
  val nDim = 2    // number of variables in the cost function
  val xBbounds = arrayOf(doubleArrayOf(-512.0, 512.0),
                         doubleArrayOf(-512.0, 512.0))
  val xRange = doubleArrayOf(xBbounds[0][1] - xBbounds[0][0],
                             xBbounds[1][1] - xBbounds[1][0])
  val lambda = 0.5
  val vMax = doubleArrayOf(lambda*xRange[0], lambda*xRange[1])

  // global parameters
  val TMAX = 50
  val SWARMSIZE = 30
  val wmax = 1.2
  val wmin = 0.5
```

```
val wt = (wmax - wmin)/TMAX
var w = wmax
val c1 = 2.0       // cognitive coefficient
val c2 = 2.0       // social coefficient

// global objects and collections
val swarm = mutableListOf<Particle>()
val BestSolution = Solution(doubleArrayOf(0.0, 0.0), Double.MAX_VALUE)
```

The first block in the code snippet imports a number of math functions from the standard Kotlin library. The next two blocks define the data classes used for this project. The `Solution()` class is used to store particle-best and swarm-best solutions ❶. The `Particle()` class is the primary class that is used to create a swarm ❷. Each particle has an identification number (`id`), position and velocity vectors (`pos` and `vel`), a `fitness` property, and a `Solution` property. The latter is used to store information on the best solution identified by the particle up to a certain point in time.

The next code block defines the `eggHolder` function ❸ and its dimensions in `nDim` (equal to 2 for the Eggholder function). It also defines the bounds and ranges for `pos` and `vel` and then defines `vMax` per Equation 9.5 to ensure that updated velocities remain within the set bounds.

The final code block defines PSO-specific global parameters. Currently, the maximum number of time steps (iterations) is set to 50, and the swarm size is set to 30. We'll also dynamically adjust the inertia factor w per Equation 9.4 for which `wmax` and `wmin` have been set to 1.2 and 0.5, respectively. The cognitive and social factors `c1` and `c2` have been set to 2.0. These parameter values were chosen based on the recommendations found in the relevant literature.

The code block ends by creating a mutable list (`swarm`) and initializing `BestSolution`, which we'll use to store the swarm-best solution. Since we're framing the problem as a minimization problem, the fitness value of the `BestSolution` has been set to `Double.MAX_VALUE`, which is the maximum possible of type `Double`.

Initializing the Swarm

The `initSwarm()` function is responsible for initializing the PSO algorithm by creating and initializing individual particles and adding them to the collection of particles (the swarm).

```
fun initSwarm() {
    println("\nStarting initialization...")

  ❶ for (i in 0 until SWARMSIZE) {
        // Define local objects.
        val pos = DoubleArray(nDim)
        val vel = DoubleArray(nDim)
        val fitness: Double
        val pBest: Solution
```

```
            // Set initial positions (random, within bounds).
        ❷ for (j in 0 until nDim) {
                pos[j] = xBbounds[j][0] + (xBbounds[j][1] - xBbounds[j][0]) *
                            (0..1000).random() / 1000.0
                vel[j] = 0.0
            }

            // Add new particles to the swarm.
        ❸ fitness = getFitness(pos)
            pBest = Solution(pos.copyOf(), fitness)
        ❹ swarm += Particle(i, pos, vel, fitness, pBest)

            // Update BestSolution.
        ❺ if (fitness < BestSolution.fitness) {
                BestSolution.pos = pos.copyOf()
                BestSolution.fitness = fitness
            }
        }
        println("\nBest solution after initialization:")
        println(BestSolution.toString())
}
```

The function begins by printing a message indicating the start of the initialization process. It then iterates using a `for` loop over a specified swarm size (`SWARMSIZE`) ❶, creating individual particles with random initial positions within predefined bounds for each dimension ❷. The initial velocities for each particle are set to zero, per Equation 9.2.

The `getFitness` function allows us to calculate the fitness value for each particle ❸. Notice that during initialization, the personal best fitness is the same as its current fitness, meaning that `pBest` initially has the same position and fitness as the particle. These particles are characterized by identification number (`id`), position (`pos`) and velocity (`vel`) vectors, a fitness value (`fitness`), and a personal best solution (`pBest`). We use these attributes to create the particles and add them to the swarm ❹.

The best solution across the entire swarm is updated if a particle's fitness is better than the current best ❺. After initialization, the function prints the best solution.

The Driver Function

The code block for the `runPSO()` driver function carries out all the core tasks of PSO, including updating the velocity and position vectors and tracking the personal- and swarm-level best solutions.

```
fun runPSO() {
  ❶ for (timeStep in 0 until TMAX) {
        // Update inertia factor as a function of time.
        val w = wmax - timeStep * wt

        // random coefficients for cognitive and social components
        val r1 = (0..100).random()/100.0
        val r2 = (0..100).random()/100.0
```

```
        // Iterate over each particle of the swarm.
      ❷ for (i in swarm.indices) {
            // Update velocity and position vectors.
            for (j in 0 until nDim) {
                // Update velocity vector, and implement bounds.
                val C1 = w * swarm[i].vel[j]
                val C2 = c1 * r1 * (swarm[i].pBest.pos[j]-swarm[i].pos[j])
                val C3 = c2 * r2 * (BestSolution.pos[j] - swarm[i].pos[j])
                val vel = C1 + C2 + C3

                // Implement velocity bounds.
                swarm[i].vel[j] = min(max(vel, -vMax[j]), vMax[j])

                // Update position vector, and implement bounds.
                swarm[i].pos[j] += swarm[i].vel[j]
                swarm[i].pos[j] =
                     min(max(swarm[i].pos[j], xBbounds[j][0]), xBbounds[j][1])
            }

            // Evaluate particle fitness.
          ❸ swarm[i].fitness = getFitness(swarm[i].pos)

            // Update the particle's best solution (pBest).
          ❹ if (swarm[i].fitness < swarm[i].pBest.fitness) {
                swarm[i].pBest.pos = swarm[i].pos.copyOf()
                swarm[i].pBest.fitness = swarm[i].fitness
            }

            // Update the global best solution.
          ❺ if (swarm[i].fitness < BestSolution.fitness) {
                BestSolution.pos = swarm[i].pos.copyOf()
                BestSolution.fitness = swarm[i].fitness
            }
        }
    }
}
```

Unlike the `runGA()` function for implementing the genetic algorithm, the `runPSO()` function is self-sufficient and doesn't rely on any helper functions other than `getFitness()`, which simply calculates the value of the function being minimized. It iterates over a fixed number of time steps (`TMAX`) ❶, beginning each iteration by initializing the inertia factor (`w`) and the random factors `r1` and `r2`.

For each time step, the code loops over each particle in the swarm ❷ and updates its velocity and position vectors according to the PSO formula. The code also implements bounds for the velocity and position values, using the minimum and maximum values defined in the arrays `vMax` and `xBbounds`.

The code evaluates the fitness of each particle by using the `getFitness()` function ❸, which takes the position vector as an input and returns the corresponding fitness as a scalar value.

The code then compares each particle's current fitness with its personal best fitness (`pBest`) and updates the latter if the former is lower ❹. It also compares the current fitness with the global best fitness (`BestSolution`) and

updates the latter if the former is lower ❺. The personal and global best solutions store both the position and the fitness values.

The function terminates when it completes `TMAX` iterations. It doesn't return anything, since the best overall solution is saved as the global object `BestSolution`.

The main() Function

This function is a short code block that prints values of key global parameters, calls other functions to initialize the swarm and run the PSO driver function, and prints the best solution found.

```
fun main() {
    println("\n*** Real-valued function optimization using PSO ***\n")
    println("Function dimensions: $nDim")
    println("Swarm size: $SWARMSIZE, Max time steps: $TMAX")
    println("w_max: $wmax  w_min: $wmin")
    println("Cognitive factor (c1): $c1")
    println("Social factor (c2): $c2")

    // Initialize the swarm.
    initSwarm()
    // Run PSO algorithm.
    runPSO()

    // Print final results.
    println("\nBest solution after $TMAX iterations:")
    println(BestSolution.toString())
}
```

The `main()` function is the entry point of the program that uses PSO to perform real-valued function optimization. It prints some information about the problem parameters, such as the function dimensions, the swarm size, the maximum time steps, and the PSO coefficients.

The function then calls two other functions, `initSwarm()` and `runPSO()`. The first function initializes the swarm of particles with random positions and velocities and evaluates their initial fitness values. The second function runs the PSO algorithm for a fixed number of iterations, updating the particles' velocities, positions, and fitness values, as well as the personal and global best solutions.

The `main()` function finally prints the best solution found by the PSO algorithm after the specified number of iterations, showing both the position vector and the fitness value of the global best solution.

The Result

I've deliberately kept the output of the PSO program brief and to the point. By now, you should be comfortable with writing your own additional lines of code to print or save other intermediate results for further analysis or visualization. If you run the code with the same parameters used in this example, the output might look like this:

```
*** Real-valued function optimization using PSO ***

Function dimensions: 2
Swarm size: 30, Max time steps: 50
w_max: 1.2  w_min: 0.5
Cognitive factor (c1): 2.0
Social factor (c2): 2.0

Starting initialization...

BestSolution after initialization:
Solution(pos=[-429.056, 374.784], fitness=-742.3993203916232)

BestSolution after 50 iterations:
Solution(pos=[512.0, 404.2263191597745], fitness=-959.640628508424)
```

The first segment of the output shows the values of key global parameters. Next, it shows the best solutions at the start and the end of the iterations. The PSO algorithm achieved a near-optimal solution for the Eggholder function within the given decision space, matching the result obtained by the genetic algorithm in Chapter 8.

Figure 9-2 shows the convergence behavior of the PSO algorithm over time when applied to the Eggholder function. Unlike the genetic algorithm discussed in Chapter 8, PSO achieves optimal solutions more rapidly, requiring fewer iterations.

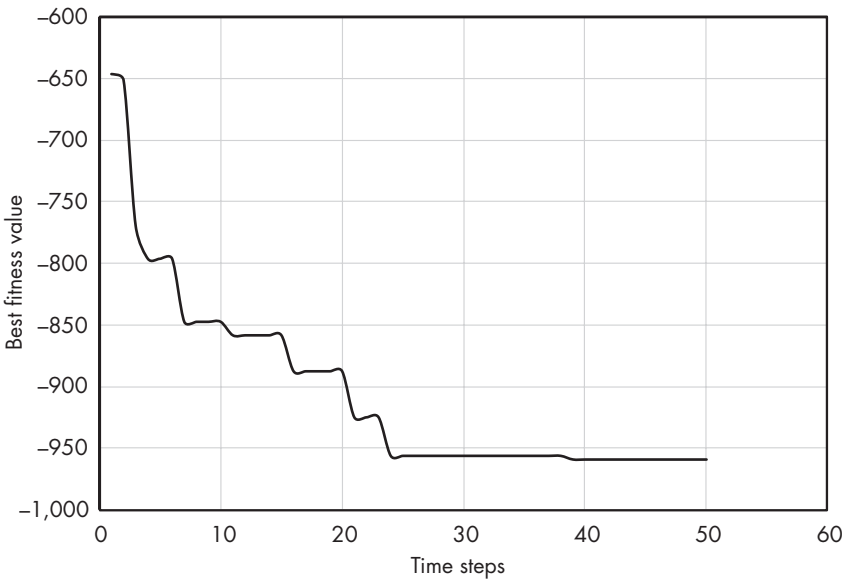


Figure 9-2: The convergence pattern for the Eggholder function, using the particle swarm algorithm

PSO seems to have an advantage over the genetic algorithm for this problem. Though the PSO algorithm had a worse initial global best fitness value of around –742, compared to –810 for the genetic algorithm, it reached the global optimum in about 40 iterations, while the genetic algorithm took 117 iterations. This suggests that the PSO algorithm can explore and exploit the search space more efficiently than the genetic algorithm for the Eggholder function. This efficiency likely stems from PSO's unique approach to exploring the solution space through collaborative particle interactions.

I encourage you to use this code to solve other known test problems and further investigate how the PSO algorithm performs vis-à-vis the genetic algorithm.

EXERCISE

The Mishra Bird function is given by the following equation:

$$f(x_1, x_2) = \sin x_1 e^{(1-\cos x_2)^2} + \cos x_2 e^{(1-\sin x_1)^2} + (x_1 - x_2)^2$$

You'll find the global minima for this function as well as the optimal solution (x_1*, x_2*) within a decision space bounded by [–2π, 2π] in both x_1 and x_2 dimensions. The function's global minimum value within the decision space is approximately –106.7645.

Hint: Define a new function `mishraBird()` and assign that to `getFitness`. Change the `xBounds` accordingly. You can increase `TMAX` to 100 for a quicker convergence without changing any other parameter values.

Notice that within this decision space are two global optimal solutions for this function: (4.701056, 3.152946) and (–1.582142, –3.130247). The global minimum fitness value is –106.764537 (all values rounded to six decimal places). If you run the revised code multiple times, the algorithm will likely identify both global optimal solutions at random.

Ant Colony Optimization

Ant colony optimization (ACO) refers to a family of algorithms that are based on lessons learned from real-world ants, especially from their foraging behavior. The original algorithm, known as the ant system (AS), was proposed by Marco Dorigo in 1992. Since then, the algorithm has been modified several times to help it more effectively solve a class of problems that requires finding the least-cost tour through all nodes of a weighted graph. In discrete mathematics, a graph is a set of nodes or vertices that are related, and the imaginary or real line connecting a pair of nodes is called

an *edge*. (You can revisit Chapter 7 for a review of graphs and conventional graph-search algorithms.)

To demonstrate the basic concept of ACO, let's review the simple illustration in Figure 9-3 of ants exploring the best paths to a source of food.

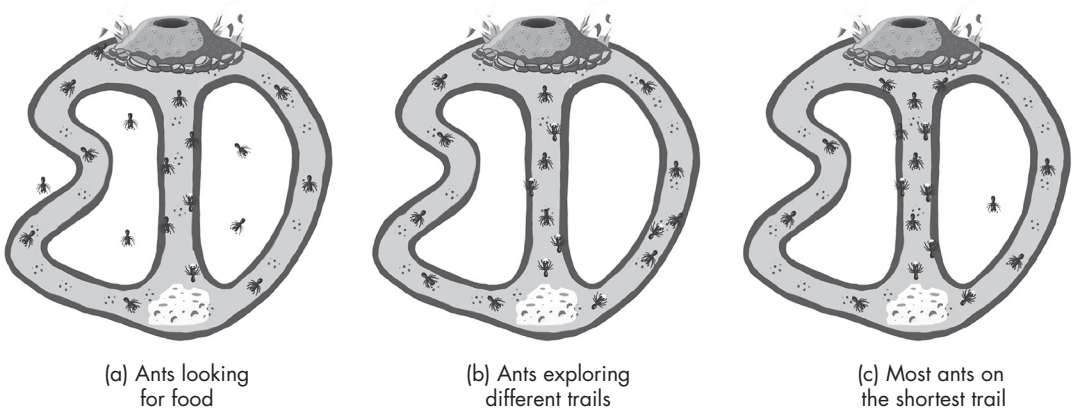


Figure 9-3: Ants exploring different trails leading to the food source

When ants start looking for a food source, they initially disperse randomly in all directions, as shown in Figure 9-3(a). As they move, they lay down a scent (pheromone) to mark their trails. Once an ant finds a food source, it picks up a piece of food and brings that back to the nest by following its scent mark. It also lays down more pheromones as it returns to the nest.

In the meantime, other ants notice the trail. Given more than one source, each trail will develop a scent mark of greater or lesser intensity, depending on how many ants are traveling back and forth along that trail and how far the source is from the nest. In general, the stronger the scent mark of a trail, the greater the number of ants following that trail will be. Once a food source is found, ants will therefore follow the most well-defined trails, as shown in Figure 9-3(b).

Crucially, pheromones are not permanent—they tend to decay or evaporate over time. If a trail is not frequently visited, it gradually becomes less traceable and eventually is forgotten. Additionally, the closest food source will take the least amount of time to visit, which means the corresponding trail will be traveled more frequently, which will result in a stronger concentration of pheromone—which will create a positive feedback loop and attract even more ants to choose the shortest route. Over time, most of the ants will start to use this shortest route (the optimal path), as shown in Figure 9-3(c). The ant colony will have used a very simple rule to solve a very difficult combinatorial optimization problem!

In this ingenious natural scheme, individual agents (ants) do not have any idea about the impact of their actions on the entire colony. And yet by repeating their simple routine, they enable the colony to find its food sources as efficiently as if the whole search process were centrally coordinated.

The ACS Algorithm

Based on the conceptual model presented in the previous section, we can envision the key components of ACO as creating a colony of artificial ants, moving ants from one node to the next based on pheromone intensity and the distance between the nodes, and updating the pheromone trails until the shortest route is found.

In this chapter, we'll explore an enhanced version of ACO called the ant colony system (ACS). The ACS algorithm is implemented in three key steps: constructing a tour, updating a local pheromone trail, and updating a global pheromone trail. We'll discuss each of these steps in detail shortly.

We'll use the ACS algorithm to solve a particular case of the traveling salesman problem (TSP), which belongs to a class of hard-to-solve problems called the NP-hard problems. Mathematically speaking, a problem is *NP-hard* if it is at least as hard as the hardest problem in NP, a class of problems for which a solution can be verified in polynomial time. It is beyond the scope of this book to discuss the NP-hard problems further; instead, we will focus on how to use ACS to solve TSPs.

Solving a TSP entails answering the following question:

> Given a list of nodes and the distances between every pair of those nodes, what is the shortest possible route a traveler can take that passes through each node exactly once and brings the traveler back to the start node?

In other words, we're looking for a closed-loop solution that goes through each node and has the shortest possible length. Notice that it is theoretically possible for multiple routes to have the same shortest length of travel.

We'll make two additional assumptions regarding the form of TSP we'll try to solve:

- The network of nodes (graph) is fully connected, meaning a traveler can visit from any particular node to all other remaining nodes (during implementation, we'll exclude the nodes that have already been visited).
- The distance between any pair of nodes is symmetric, meaning the distance does not change with the direction of travel (a pair of nodes are connected by a single, unique path—an edge).

Symbols and Their Meanings

ACS is a fairly complex algorithm with many parameters and variables, listed in Table 9-1 along with the symbols used to represent them.

Table 9-1: Symbols Used in the ACS Algorithm

Symbol	Interpretation
$\mathbf{N}_i^k$	The neighborhood of ant k when it is at node i; it is a set of nodes that an ant is allowed to visit given its current location.
τ_{ij}	The pheromone intensity of edge $[i, j]$ connecting node i to node j.

Symbol	Interpretation
τ_0	Initial pheromone level for all edges, set to $1/(nC^{nn})$; n is the total number of nodes in the nearest-neighbor tour and C^{nn} is the tour length.
d_{ij}	The length of the edge from node i to node j (distance between these nodes). Also, $d_{ij} = d_{ji}$.
η_{ij}	Heuristic information defined as $1/d_{ij}$.
α	Parameter, set to 1 for ACS.
β	Parameter [2.0–5.0], used as the exponent of η.
q	A uniformly distributed random variable in [0, 1].
q_0	A parameter in (0, 1); an ant explores the learned knowledge based on the intensity of pheromone trails and heuristics when $q \leq q_0$.
p_{ij}	The normalized probability for choosing edge [i, j] during roulette wheel selection if $q > q_0$.
ζ	Parameter, set to a small value such as 0.005; used as the weighting factor for updating the local pheromone trail.
ρ	Parameter, set to the recommended value of 0.1 for ACS; used as the weighting factor for updating the best-so-far global pheromone trail.
C^{nn}	Tour length for the nearest-neighbor tour used for estimating initial pheromone concentration.
C^{bs}	Tour length for the best-so-far solution or tour.
T^{bs}	The best-so-far tour (collection of edges that constitute the tour).

In addition to listing the parameters and variables, Table 9-1 also provides short descriptions of those elements. (You may need to revisit these descriptions as you read the rest of this section.)

The Steps of ACS

In this section, we'll explore the key steps of the ACS algorithm, expressed mathematically. This will include three steps: tour construction, updating the local pheromone trail, and updating the global pheromone trail.

Step 1: Tour Construction

The first step of ACS entails applying a pseudorandom proportional rule used by an ant to choose its next location j given its current location i, defined in Equation 9.8.

$$j = \begin{cases} argmax_{l \in N_i^k} \{\tau_{il}\eta_{il}^{\beta}\}, & \text{if } q \leq q_0 \\ J, & \text{otherwise} \end{cases} \tag{9.8}$$

The *argmax* function in Equation 9.8 chooses an argument l from the feasible set of nodes that can be visited from node i, for which the expression inside the curly brackets is maximized. This value of l is set to j as the next destination for the ant provided $q \leq q_0$. The parameter q_0 allows us to

control the degree to which learned knowledge (pheromone trails and heuristics) is prioritized over random exploration of new routes.

When $q > q_0$, the next node j is set to J, which is a random variable selected using a roulette wheel scheme. Equation 9.9 estimates the normalized probabilities for the feasible paths from node i to node j for ant k:

$$p_{ij}^{k} = \frac{\tau_{ij}\,\eta_{ij}^{\beta}}{\sum_{l \in N_i^k} \tau_{il}\,\eta_{il}^{\beta}} \qquad j \in \mathbf{N}_i^k \tag{9.9}$$

For a refresher on the roulette wheel scheme, please review "Selection" on page 312 in Chapter 8. This time, we'll implement the scheme in our final coding project.

Step 2: Updating the Local Pheromone Trail

As soon as an ant moves from node i to node j, the weighted average scheme in Equation 9.10 is applied to update the local pheromone trail.

$$\tau_{ij} = (1 - \zeta)\,\tau_{ij} + \zeta\,\tau_0 \tag{9.10}$$

In the original ACO, no local updates occur, which allows one to implement the tour construction either sequentially or concurrently. When tours are built concurrently (in parallel), they can result in significant computational time savings for large real-world problems. Due to the local updating rule of ACS, it is implied that the process will be implemented sequentially. This is because the tour created by a specific ant is influenced by the preceding ants' local updates to the pheromone trails.

Step 3: Updating the Global Pheromone Trail

Once all ants are done building their tours, the global pheromone update rule is applied once per iteration and only along the edges of the best-so-far tour, T^{bs}, as shown in Equation 9.11.

$$\tau_{ij} = (1 - \rho)\,\tau_{ij} + \rho\left(\frac{1}{C^{\text{bs}}}\right), \qquad \text{for all } (i,j) \in T^{\text{bs}} \tag{9.11}$$

Equation 9.11 implies that T^{bs} will need to be compared with the best solution identified by the colony at the end of each iteration and then be updated as needed.

The Pseudocode

The following pseudocode shows how the components of the ACS algorithm come together to form a sophisticated heuristic algorithm:

```
// preprocessing - read and initialize
read nodes from file (name, number, coordinates)
calculate edge lengths (Euclidian distance)
calculate nearest-neighbor tour length (Cnn)
initialize pheromone levels to τ0 = 1/(nCnn)

// Iteratively apply ACS steps.
while stopping condition not met:
        initialize ants (assign id numbers and initial positions)
        for ant = 0 .. < number of ants
                set cities to visit for each ant
                while number of cities to visit > 0:
                        construct tour per Equation 9.8 & Equation 9.9
                        perform local pheromone update per Equation 9.10
                end while
                // Complete the loop.
                return to the start node
                update ant properties
                perform local pheromone update per Equation 9.10
        end for
        update best-so-far solution
        perform global pheromone update per Equation 9.11
end while

// postprocessing
print results
```

We'll follow this pseudocode to develop a complete ACS application in Kotlin in the next project.

Project 37: Solve the Traveling Salesman Problem with an Ant Colony System

In this project, we'll solve the well-known test problem Berlin52. This is a combinatorial optimization problem that involves finding the shortest route through 52 destinations in Berlin, Germany. The dataset for this problem was retrieved from TSPLIB, which is a collection of traveling salesman problems with known global optimal solutions. See "Resources" on page 377 for the download link for these problems, which you can try solving by using the ACS algorithm and other NIAs discussed in this book.

The Code

The Berlin52 TSP requires datasets for 52 different locations. It's more convenient to create a separate datafile for this project (for example, a file in CSV format) and read the location data from the file at runtime. So we'll adopt that approach for this project.

We'll discuss the code segments in four primary blocks: problem definition and global parameters, the `main()` block, the `runACS()` driver function and its helper functions, and additional intermediate and postprocessing of results.

Problem Definition and Global Declarations

This segment specifies the import block and defines the input file location, global variables and parameters, and a collection of classes, lists, and arrays required to implement the ASC algorithm.

```
// import block
import java.io.File
import kotlin.math.pow
import kotlin.math.sqrt

// Input file location: change the datafile location as needed.
❶ const val datafile = "berlin52.csv"

❷ // global parameters

val numCities = 52     // Set the number of cities.
val numAnts = 30       // Typically set to 10-30.
val rho = 0.1          // rho = 0.1 is recommended for ACS.
var pheromone0 = 0.0   // 1/(n*Cnn) for ACS
val q0 = 0.8           // argmax parameter (0.5-0.9)
val zeta = 0.005       // Set to a small value.
val alpha = 1.0        // alpha = 1 for ACS.
val beta = 2.0         // Set to 2-5.
val iterMax = 300      // maximum number of iterations
val maxRounds = 50     // number of times the entire process is repeated

❸ // classes and collections

data class City(
    val name: String,
    val node: Int,
    val x: Double,
    val y:Double
)

class Ant(val id: Int, val start: Int) {

    var currentNode = start

    val citiesToVisit = mutableListOf<Int>()
    val pathNodes = ArrayList<Pair<Int, Int>>()
    val pathSegments = mutableListOf<Double>()

    // Set fitness to a very high value for function minimization.
    var fitness = Double.POSITIVE_INFINITY

    fun setCitiesToVisit() {
        for (i in 0 until numCities) {
            if (i != this.start)
                this.citiesToVisit += i
        }
    }
}
```

```
data class ArgMax(
    val index: Int,
    val value: Double
)

data class Solution(
    val iteration: Int,
    val antID: Int,
    val pathNodes: ArrayList<Pair<Int, Int>>,
    val segments: List<Double>,
    val fitness: Double
)

val cities = mutableListOf<City>()
val ants = mutableListOf<Ant>()
val antSolutions  = mutableListOf<Solution>()
val bestSolutions = mutableListOf<Solution>()

val edges = Array(numCities) {DoubleArray(numCities)}
val pheromone = Array(numCities) {DoubleArray(numCities)}
val prob = Array(numCities) {DoubleArray(numCities)}
var bestOverallTour = ArrayList<Pair<Int, Int>>()

var bestOverallFitness = Double.POSITIVE_INFINITY
var optimaCount = 0
```

The code segment begins by importing the required methods from the standard Kotlin and Java libraries. We'll use `java.io.File` to read data from an input file from a specified location (in this case, from *berlin52.csv*) ❶. (The input file you'll use will likely have a different location, so you must change the input file location as needed.)

The input file for the Berlin52 TSP follows a set format. The first value in the initial line contains a brief title describing the problem, while subsequent values serve as column headers for the data points. From the second line to the last, city-specific information is presented in groups of four comma-separated values (hence the file extension *.csv*): city name, city identification, x-coordinate, and y-coordinate, respectively. Each row contains data for a particular city or location.

In this case, the datafile consists of 53 lines, including the introductory line that provides descriptive information. Table 9-2 shows how the file will look when you open it with Microsoft Excel or another spreadsheet program.

Table 9-2: Input File Format for the Berlin52 TSP

Berlin52	ID	X	Y
C1	0	565	575
C2	1	25	185
C3	2	345	750
C4	3	945	685

(continued)

Table 9-2: Input File Format for the Berlin52 TSP *(continued)*

Berlin52	ID	X	Y
C5	4	845	655
C6	5	880	660
. . .	. . .	. . .	. . .
C50	49	595	360
C51	50	1340	725
C52	51	1740	245

The next code block declares all the global parameters ❷. For example, the number of cities (`numCities`) is set to 52, the number of ants (`numAnts`) is set to 30, the maximum number of iterations per round (`iterMax`) is set to 300, and the number of times the entire process is repeated (`maxRounds`) is set to 50. The comments next to the parameters indicate suggested ranges or values for these parameters.

Next, we define the classes and collections used in this project ❸. The `City` data class stores information on the locations to visit, including their names, ID numbers, and coordinates.

The `Ant` class is used to create the ant colony, which is at the heart of ACS. The `Ant` class has several properties required for managing and monitoring ant movements. In particular, `citiesToVisit` dynamically keeps track of the remaining cities to visit, `pathNodes` stores the start and the end nodes for each path (edge) already traversed, and `pathSegments` stores the corresponding edge lengths. This class also has a method called `setCitiesToVisit()` that defines the initial list of cities each ant can visit.

The `ArgMax` class is used during the tour construction phase. The `Solution` class stores information on completed tours, including the nodes and edges comprising a tour and its fitness value (length of the tour).

These classes are followed by a block that initializes a number of collections, arrays, and parameters. For example, `cities` is used to build a list of locations to visit, and `ants` is used to create the ant colony. Others are used to store ant solutions (`antSolutions`), best solutions (`bestSolutions`), and the best overall tour (`bestOverallTour`).

We use two-dimensional arrays to store information on the edges (`edges`) and corresponding pheromone levels (`pheromone`) and edge probabilities (`prob`). The final two lines set the default values for the best overall fitness (`bestOverallFitness`) and the counter variable `optimaCount`.

The main() Block

The `main()` block is a minimal block with a few function calls and print functionalities.

```
fun main() {
    println("\n*** Solving Berlin52 TSP Using the Ant Colony System ***\n")
    // Conduct preprocessing.
```

```
    readCities()
    calculateEdges()
    calculatePheromone0()

    // Repeat the process maxRounds number of times.
  ❶ for (round in 1..maxRounds) {
      ❷ initializePheromone()
            runACS()
        processInterimResults(round)
        // Prepare for next iteration.
      ❸ bestSolutions.clear()
    }

    // Print the best overall solution.
    printBestOverallFitnessAndTour()
    println("\noptimaCount: $optimaCount")
}
```

The function starts by printing a problem-specific message on the console and then moves on to the preprocessing block. It calls the `readCities()` function to read off the location data from the input file. The `calculateEdges()` function uses the location coordinates to estimate the distances between pairs of nodes. The `calculatePheromone0()` function finds the nearest-neighbor tour length C^{nn} and uses that to estimate the initial pheromone levels τ_0 for all edges.

Next, we introduce a `for` loop ❶ to carry out the entire ACS process a set number of times (`maxRounds`). The Berlin52 TSP is a challenging problem, and the ACS algorithm may not be able to locate the global optimal solution during each round of the ACS process, which involves `iterMax` attempts.

Inside the `for` loop, the `initializePheromone()` function ❷ sets the initial pheromone level for all edges of the graph. Next, the driver function `runACS()` carries out the search for the optimal route. The `processInterimResults()` function updates the best overall fitness and best overall tour values, then prints the best solution for each round of search. Finally, the list of best solutions is cleared before starting the next round ❸.

The success rate of the ACS algorithm depends on the combination of global parameter values and the random initialization of the ants and their start nodes (in addition to the random selection of edges that happens during the implementation of the algorithm). In general, when a heuristic algorithm gets stuck at a local optimum even after attempting to find the global optimal solution a reasonable number of times, it may be beneficial to restart the entire process by resetting the initial conditions and changing the global parameters if needed, rather than increasing the number of iterations. The `for` loop in the `main()` block helps us automatically reset and restart the ACS process and carry it out `maxRounds` times.

Notice that we need to read the input data, calculate edge lengths, and calculate the initial pheromone level only once. Thus, those tasks are completed before initiating the `for` loop inside the `main()` block. However, we need to reset the pheromone levels to τ_0 each time we call the `runACS()` function.

The `main()` block ends with printing the best overall fitness and tour (from all rounds and iterations). It also prints the number of times the algorithm was successful in finding the global optimal solution.

The readCities() Function

The sole purpose of this function is to read the location data from an input file.

```
fun readCities() {
    // Open input file and read location data.
  ❶ val file = File(datafile)
  ❷ val lines = file.readLines().filterNot{it.isEmpty()}

    for (i in lines.indices) {
      ❸ val items = lines[i].split(",")
        if (i == 0) {
            println("Reading data for " + items[0] + "-TSP")
        } else {
            // Read Name, ID, x, y.
          ❹ cities += City(
                items[0],
                items[1].trim().toInt(),
                items[2].trim().toDouble(),
                items[3].trim().toDouble()
            )
        }
    }
}
```

The Java `File` class ❶ opens the input file from a location specified in the datafile string. The entire content of the datafile is read into memory as an array of strings (`lines`) ❷. We are applying `.filterNot{it.isEmpty()}` to `file.readLines()` to make reading the file safer with respect to empty lines.

Then, each line is split by using a comma (`,`) as the separator ❸ (recall that the input file was created as a comma-separated value, or CSV, file). Finally, different parts of the split line are used to create a list of nodes (`cities`) by using the `City` class ❹.

The calculateEdges() Function

The purpose of this function is to calculate and save the edge lengths (path segments) of a tour.

```
fun calculateEdges() {
    // Assume symmetry: edges[i][j] = edges[j][i].
    for (i in 0 until cities.size) {
        for (j in i until cities.size) {
            if (i == j) {
              ❶ edges[i][j] = 0.0
            } else {
```

```
            ❷ edges[i][j] = sqrt((cities[i].x - cities[j].x).pow(2) +
                    (cities[i].y - cities[j].y).pow(2))
            ❸ edges[j][i] = edges[i][j]
          }
        }
    }
}
```

The edge lengths are saved in a two-dimensional array, where the diagonal elements (`i = j`) are set to `0.0` ❶ (the distance of a node from itself is zero), and the off-diagonal elements are set to Euclidian distances between a pair of nodes ❷. We're assuming that a pair of nodes is connected by a single edge or path. This allows us to calculate only the upper triangle of the matrix and set the lower triangle values by using the property of symmetry ❸.

The calculatePheromone0() Function

This function calculates the nearest-neighbor tour length C^{nn} and uses that to estimate the initial pheromone level, `pheromone0`.

```
fun calculatePheromone0() {
    // Start at node 0 (first city in the dataset).
  ❶ var i = 0 // Start node for the nearest-neighbor tour.
    val citiesToVisitList = (1 until numCities).toMutableList()
    var nearestNeighborTourLength = 0.0

    // Build the nearest-neighbor tour.
  ❷ while (citiesToVisitList.size > 0) {
        // Set initial search parameters.
        var nearestNode = -9999 // Use an unlikely value.
        var nearestNodeDistance = Double.MAX_VALUE

        for (j in citiesToVisitList) {
            if (edges[i][j] < nearestNodeDistance) {
                nearestNodeDistance = edges[i][j]
                nearestNode = j
            }
        }

      ❸ nearestNeighborTourLength += nearestNodeDistance
      ❹ citiesToVisitList.remove(nearestNode)
        i = nearestNode
    }
    // Add the edge connecting the last city visited and the starting city.
  ❺ nearestNeighborTourLength += edges[i][0]

    // Calculate initial pheromone value per ACS.
  ❻ pheromone0 = 1.0/(numCities * nearestNeighborTourLength)
}
```

This function finds the nearest-neighbor tour by starting the tour from node 0 ❶ and moving to the nearest nodes with a `while` loop ❷ until all

nodes or cities are visited. At each step, the shortest edge is identified and added to the nearest-neighbor tour length ❸. Once the node is added to the tour, it is removed from the list of cities to visit ❹. The tour is closed by connecting the last node visited to the start node and adding the corresponding edge length to the total tour length ❺. Finally, the initial pheromone level `pheromone0` is calculated as equal to $1/(nC^{nn})$ ❻.

The initializePheromone() Function

This function sets initial pheromone levels for all edges of the graph to `pheromone0`.

```
fun initializePheromone() {
    // All edges have the same initial pheromone level.
    for (i in 0 until numCities) {
        for (j in i until numCities)
            if (i != j) {
                pheromone[i][j] = pheromone0
                pheromone[j][i] = pheromone0
            } else pheromone[i][j] = 0.0
    }
}
```

The pheromone levels are stored in a two-dimensional array (a matrix), and the property of symmetry is used to calculate the lower triangle elements of the matrix by setting $\tau_{ji} = \tau_{ij}$. Notice that when $i = j$, the node is simply referring to itself, and the corresponding diagonal elements are set to 0.0. These values are not required or used by the ACS algorithm.

The runACS() Function

The `runACS()` function creates the ant colony, coordinates the tour construction for individual ants, saves intermediate results, and implements the global pheromone update rule. We'll begin by providing an overview of the key elements of this function, which will be followed by discussions on the individual helper functions.

```
fun runACS() {
    var iter = 1
    while(iter <= iterMax) {
        // Create a new ant colony.
      ❶ initializeAnts()

        // Generate tours for all ants.
        for (ant in ants) {
          ❷ ant.setCitiesToVisit()
          ❸ buildAntTour(ant)
          ❹ antSolutions += Solution(
                                iter,
                                ant.id,
                                ant.pathNodes,
                                ant.pathSegments,
```

```
                                    ant.fitness)
        }

        // Get the solution with minimum fitness.
      ❺ val bestAntSolution =
            antSolutions.minWith(compareBy {it.fitness})
      ❻ bestSolutions += bestAntSolution
      ❼ globalPheromoneUpdate()

        // Clear previously saved states of ants and solutions.
        ants.clear()
        antSolutions.clear()

        // Advance the counter.
      ❽ iter += 1
    }
}
```

The code segment begins by setting the iteration counter `iter` to `1`. A `while` loop is used to repeat the search `iterMax` times. Inside the loop, `initializeAnts()` creates the ant colony for a specific iteration ❶.

Once the ant colony is initiated, a tour for each ant is constructed inside a `for` loop. The process starts with setting a list of cities to visit for each ant ❷ by invoking the `setCitiesToVisit()` method of the `Ant` class. The actual tour is built by the `buildAntTour()` function ❸.

Once the optimal tour for an individual ant is completed, relevant information is saved in `antSolutions` ❹. The best of all ant solutions (for the current iteration) is found by using the `minWith()` function of Kotlin ❺, which is then added to the `bestSolutions` list ❻. At this stage, the `globalPheromoneUpdate()` function is called to apply the global pheromone update rule ❼. Next, `ants` and `antSolutions` are cleared in preparation for the next iteration. Finally, `iter` is incremented by 1 ❽; the process will exit the `while` loop when `iter` > `iterMax`.

The initializeAnts() Function

The code snippet for the `initializeAnts()` function is very short.

```
fun initializeAnts() {
    // Create a list of nodes (cities) to set start nodes for ants.
    val cityList = (0 until numCities).toList()
    // While creating a new Ant, set its start node randomly.
    for (i in 0 until numAnts) ants += Ant(i, cityList.random())
}
```

This code begins by creating a list of indices, each designating a city or node. Next, it creates the ants one by one and assigns each ant a start node selected randomly from the list of cities to visit.

This scheme allows multiple ants to have the same start node, while some nodes may not have any ants assigned to them. This flexibility is beneficial when the number of ants differs from the number of nodes.

The buildAntTour() Function

This function identifies the next node to visit, updates relevant ant properties to reflect that choice, and recursively builds the entire tour. It also calculates the ant fitness when the tour is complete and applies the local pheromone update for each edge traveled.

```
fun buildAntTour(ant: Ant) {

  ❶ var i = ant.currentNode

    while (ant.citiesToVisit.size > 0) {
        // Find the next node to visit per ACS.
      ❷ val nextNode = selectNodeToVisit(i, ant)

        // edge-specific local pheromone update per ACS
      ❸ pheromone[i][nextNode] =
                (1- zeta) * pheromone[i][nextNode] +
                (zeta * pheromone0)
        ant.currentNode = nextNode
        ant.pathNodes += Pair(i, nextNode)
        ant.pathSegments += edges[i][nextNode]
        i = ant.currentNode
      ❹ ant.citiesToVisit.remove(nextNode)
    }

    // Close the loop by adding the last Pair() of nodes to the path
    // and adding the last path segment to the tour.
    ant.pathNodes += Pair(ant.currentNode, ant.start)
    ant.pathSegments += edges[ant.currentNode][ant.start]

    // Calculate the fitness of the entire loop (closed path).
  ❺ ant.fitness = ant.pathSegments.sum()

    // edge-specific local pheromone update for the last path segment
  ❻ pheromone[ant.currentNode][ant.start] =
        (1- zeta) * pheromone[ant.currentNode][ant.start] +
        (zeta * pheromone0)
}
```

The function starts with setting the current location index `i` to the ant's `currentNode` property (which is initially the same as `startNode`) ❶. Subsequent nodes to visit are found by using a `while` loop until the list of cities to visit is exhausted. The process of selecting the next node is quite involved and is implanted by `selectNodeToVisit()` ❷, a function we'll discuss in more detail shortly.

After locating `nextNode`, the local pheromone update rule is applied ❸, and the `currentNode` property of the ant is set to `nextNode`. At this time, relevant ant properties are updated based on the move from node `i` to `nextNode`. Before repeating the iteration for the next node or city to visit, the current node index `i` is updated to `currentNode` (that is, to the most recent `nextNode`) and then `nextNode` is removed from the list of cities to visit ❹.

Once the ant is done visiting all the cities it is allowed to visit, the tour is closed by connecting the last city visited to the city from which the ant started its tour. This is done by updating the ant's `pathNodes` and `pathSegments` properties. When the tour is complete, its fitness (length) is calculated ❺, and the local pheromone update rule is applied one more time for the last segment of the tour ❻.

The selectNodeToVisit() Function

This function implements the most mathematically involved part of the ACS algorithm that uses both an argmax type function and the roulette wheel scheme to decide which node to visit next.

```
fun selectNodeToVisit(i: Int, ant: Ant): Int {

    var chosenNode = -9999 // Use an unlikely value.
    val argmaxList = mutableListOf<ArgMax>()

    // Calculate edge probabilities and argMaxList elements.
    var sum = 0.0
  ❶ for (j in ant.citiesToVisit) {
        prob[i][j] = (pheromone[i][j]).pow(alpha) /
                     (edges[i][j]).pow(beta)
        sum += prob[i][j]
        argmaxList += ArgMax(j, prob[i][j])
    }

    // Calculate normalized values of the edge probabilities.
  ❷ for (j in ant.citiesToVisit) {
        prob[i][j] = prob[i][j] / sum
    }

    // Use argmax or roulette wheel scheme to select j.
  ❸ val q = (0 until 1000).random()/1000.0

    if (q <= q0) {
        // Use accumulated experience more greedily per ACS.
        val maxArgMax = argmaxList.maxWith(compareBy {it.value})
      ❹ chosenNode = maxArgMax.index
    } else {
      ❺ // Use roulette wheel scheme.
        val spin = (0 until 1000).random()/1000.0
        var sumProb = 0.0
        for (j in ant.citiesToVisit) {
            sumProb += prob[i][j]
          ❻ if (spin <= sumProb) {
                chosenNode = j
                break
            }
        }
    }
  ❼ return chosenNode
}
```

The function first creates local variables for saving the chosen node (`chosenNode`, initially set to an unlikely value) and a mutable list to which an argmax operation will be applied per Equation 9.8. Next, edge-specific raw probabilities are calculated inside the first `for` loop ❶. This loop also populates the `argmaxList`, which stores the possible destination node index *j* as its `index` property and the corresponding `prob[i][j]` (before being normalized) as its `value`. The second `for` loop ❷ converts the raw probabilities to normalized probabilities.

After the initial processing, a random number q is drawn from a uniform distribution ❸. If $q <= q_0$, the argmax rule is used to choose the next node index ❹. Otherwise, Equation 9.9 is used to find the next node index by using the roulette wheel scheme ❺. In particular, when the `spin` value is less than or equal to the sum of normalized probabilities up to index *j*, we set `chosenNode` equal to *j* ❻ and break out of the loop. Finally, the value of `chosenNode` is returned ❼.

The globalPheromoneUpdate() Function

This function applies the global pheromone update rule once all ants finish building their tours for a particular iteration.

```
fun globalPheromoneUpdate() {
    val bestSoFar =
        bestSolutions.minWith(compareBy {it.fitness})
    for (pair in bestSoFar.pathNodes) {
        val (i,j) = pair
        pheromone[i][j] = (1 - rho) * pheromone[i][j] + rho/bestSoFar.fitness
    }
}
```

This function has two key steps. First, it identifies the best-so-far solution since the beginning of the iterations inside the `while` loop of `runACS()`. Next, pheromone levels are updated only for the edges (path segments) that belong to the best-so-far tour.

Other Functions in the main() Block

The other functions in the `main()` block are not part of the ACS algorithm. Instead, we use these functions to monitor the convergence of the algorithm and to print final values of the best overall fitness and corresponding solution at the end.

The `processInterimResults()` function helps save and print intermediate results after the completion of each round of calculations inside the `for` loop of the `main()` function.

```
fun processInterimResults(round: Int) {
  ❶ val bestSoFar =
        bestSolutions.minWith(compareBy {it.fitness})
  ❷ if (bestSoFar.fitness < bestOverallFitness) {
        bestOverallFitness = bestSoFar.fitness
```

```
        bestOverallTour = bestSoFar.pathNodes
    }

❸ // Print interim results.
    println("round: $round iter: ${bestSoFar.iteration}" +
            "ant: ${bestSoFar.antID}")
    println("bestSoFar.fitness: ${bestSoFar.fitness}")

    // Count the number of times global optima are found.
❹ if (bestSoFar.fitness - 7544.3659 < 0.0001)
        optimaCount += 1
    }
```

This function first sorts the `bestSolutions` to find the best-so-far solution based on the fitness values of the solutions ❶. Next, it updates the value of `bestOverallFitness`, the best fitness found from all rounds up to this point ❷. The current number of rounds, the iteration number, and the `antID` are then printed along with the best-so-far fitness ❸. This helps us monitor how the algorithm is doing as it proceeds through the number of rounds (as mentioned earlier, the maximum number of rounds is set by `maxRounds`).

Finally, we check to see if the fitness of `bestSoFar` matches the known global optimal fitness for the Berlin52 problem (7544.3659) and count the number of such matches ❹ (which is later printed from the `main()` function).

The final function called from `main()` is `printBestOverallFitnessAndTour()`, which prints the optimal function value and solution found by the ACS algorithm.

```
fun printBestOverallFitnessAndTour() {
    println("\nbestOverallFitness:  $bestOverallFitness")
    println("\nbestOverallTour: ")

    for (i in bestOverallTour.indices) {
        print("${bestOverallTour[i]}".padEnd(10))
        if ((i+1) % 5 == 0) println()
    }
    println()
}
```

The first line inside the function prints the value of the best overall fitness. The optimal solution in this case is a list of `Pairs`, where each pair consists of the start and the end nodes for the edges that belong to the best overall tour. We use a `for` loop and an `if` statement to print five pairs of nodes per line so that the entire solution can be examined easily in the console.

The Result

Following is a sample output from a test run of the ACS application. I encourage you to compare this output with the various print statements and functions used in the entire ACS code.

```
*** Solving Berlin52 TSP Using the Ant Colony System ***

Reading data for Berlin52 TSP

round: 1 iter: 245 ant: 11
bestSoFar.fitness: 7994.43559098265
round: 2 iter: 105 ant: 0
bestSoFar.fitness: 7544.365901904086
round: 3 iter: 270 ant: 2
bestSoFar.fitness: 7842.717484844844

--snip--

round: 49 iter: 186 ant: 14
bestSoFar.fitness: 7544.365901904086
round: 50 iter: 226 ant: 9
bestSoFar.fitness: 7721.297918696817

bestOverallFitness:  7544.365901904086

bestOverallTour:
(14, 5)   (5, 3)    (3, 24)   (24, 11)  (11, 27)
(27, 26)  (26, 25)  (25, 46)  (46, 12)  (12, 13)
(13, 51)  (51, 10)  (10, 50)  (50, 32)  (32, 42)
(42, 9)   (9, 8)    (8, 7)    (7, 40)   (40, 18)
(18, 44)  (44, 31)  (31, 48)  (48, 0)   (0, 21)
(21, 30)  (30, 17)  (17, 2)   (2, 16)   (16, 20)
(20, 41)  (41, 6)   (6, 1)    (1, 29)   (29, 22)
(22, 19)  (19, 49)  (49, 28)  (28, 15)  (15, 45)
(45, 43)  (43, 33)  (33, 34)  (34, 35)  (35, 38)
(38, 39)  (39, 36)  (36, 37)  (37, 47)  (47, 23)
(23, 4)   (4, 14)

optimaCount: 5
```

We can see that on this occasion, the first instance of global optima was found in the second round (during `iter` = 105 and by `ant` number 0). A near-optimal solution with a fitness of 7548.99 was found multiple times (not shown). The best overall solution had a fitness of 7544.3659, which is the known shortest tour length for the Berlin52 problem.

All the nodes that belong to the optimal (best overall) tour are also shown in the output. Notice that the optimal tour is a closed loop, and it returns to the same node it starts from. The sequence of the nodes in the optimal solution may differ when you run the code. This will not affect the tour length (therefore, its fitness will remain the same).

The final item in the output, `optimaCount`, indicates that during the entire process, the global optimal solution was found 5 times out of 50 rounds (although each of those rounds might have found the global optimal solution more than once). If you plot the nodes that belong to the best overall tour by using their x- and y-coordinates, the optimal tour will look like the path shown in Figure 9-4.

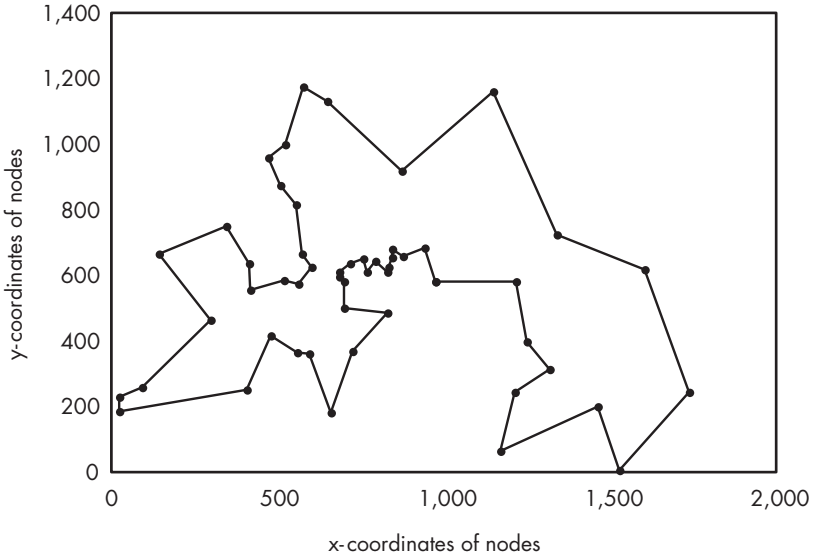


Figure 9-4: The optimal tour for the Berlin52 traveling salesman problem

To visualize the convergence patterns for the rounds that found the global optima, you can add a few additional lines of code to save the relevant data from intermediate steps and plot the data. A typical convergence plot will look like the patterns shown in Figure 9-5.

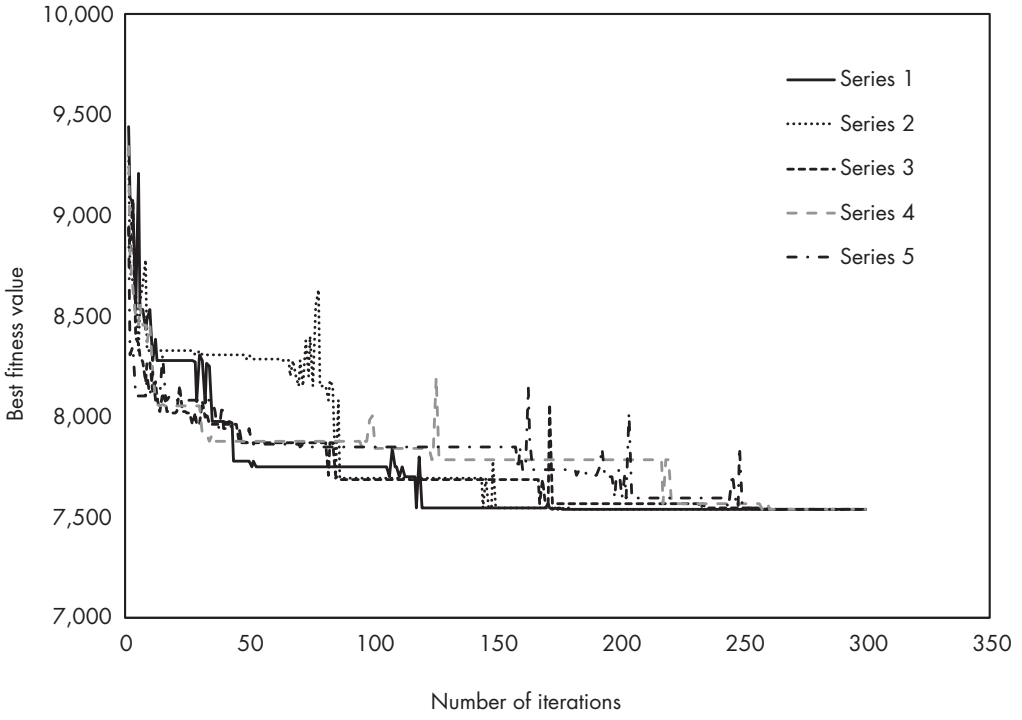


Figure 9-5: Convergence patterns for the Berlin52 problem

Before concluding this project, I want to make a couple of points regarding the success rate of the ACS algorithm and the accuracy of the solution generated by the code we developed. These comments will clarify some questions that you may have when you run the code on your device or compare the results with the same published elsewhere.

First, recalling that the NIAs used in this book are stochastic, the `optimaCount` will vary each time you run the program. For the given set of values for the global parameters, I found the average `optimaCount` to be around 5 (based on 10 runs). However, if you change the values of the global parameters, this average success rate will change. I encourage you to play with those parameters to develop an understanding of their relative influence in finding the global optima.

Second, you may find in the literature that the optimal (shortest) tour length for the Berlin52 problem is 7542, which is slightly different from the optimal value we found, 7544.3659. This does not indicate any issues with the ACS algorithm or with the code developed in this project; it is due to the fact that some algorithms convert the nodal (intercity) distances to the nearest integer values before solving the problem, for mathematical efficiency. Therefore, those methods essentially solve a slightly different problem. However, our ACS application has identified the exact same optimal route as reported in the literature.

Ant colony optimization is an area of active research, just like other NIAs. New modifications are being proposed and tested to improve the convergence and accuracy of this algorithm. I strongly encourage you to consult recently published literature if you are interested in using ACS or similar algorithms to solve large real-world routing problems.

Summary

This chapter completes your introduction to the fascinating world of NIAs and their applications. You discovered two more powerful tools—particle swarm optimization and ant colony systems—and you learned how to harness the power of these algorithms in Kotlin. You put your skills to the test with two real-world optimization problems: finding the global minimum of a complex mathematical function and solving the traveling salesman problem for a network of 52 locations in Berlin. You explored how the algorithms converged to the optimal solutions over time and how to measure their performance. And of course, you challenged yourself with exercises to reinforce your learning.

Resources

Brownlee, Jason. *Clever Algorithms: Nature-Inspired Programming Recipes.* Electronic version, June 16, 2012. *https://github.com/clever-algorithms/CleverAlgorithms.*

Clerc, Maurice. *Particle Swarm Optimization*. London: ISTE, 2006.

Dorigo, Marco, and Thomas Stützle. *Ant Colony Optimization*. Cambridge, MA: MIT Press, 2004.

Olsson, Andrea E., ed. *Particle Swarm Optimization: Theory, Techniques and Applications*. New York: Nova Science, 2011.

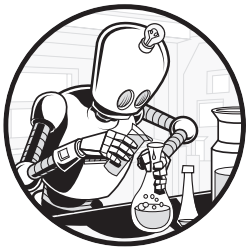
Parsopoulos, Konstantinos E. "Particle Swarm Methods." In *Handbook of Heuristics*, edited by Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende, 639–685. Cham, Switzerland: Springer, 2018.

Solnon, Christine. *Ant Colony Optimization and Constraint Programming*. London: ISTE, 2013.

TSPLIB. Symmetric Traveling Salesman Problem (TSP). Accessed June 15, 2024. *http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/*.

Yang, Xin-She. *Nature-Inspired Optimization Algorithms*. 2nd ed. London: Academic Press, 2021.

AFTERWORD



I sincerely hope this book has sparked your curiosity and passion for coding and problem-solving. However, this is just the beginning of your journey in the fascinating world of Kotlin and its applications. You can use Kotlin to tackle practical problems in almost any field of study, from science and engineering to art and entertainment.

This book has given you a solid foundation in Kotlin's core features. You're now ready to dive deep and explore more advanced topics, such as Java interoperability, generics, extension functions, and coroutines. If your goal is to become an Android app developer, you will also have to learn how to use Android Studio and Jetpack Compose for developing apps with modern and multiplatform user interfaces.

As before, your primary source of information is the official documentation at *https://kotlinlang.org*. However, you can choose from many excellent books that address your specific needs. No matter where your coding journey takes you next, you'll benefit from learning about the practice of clean

coding and design patterns for developing more complex, fully functional applications. I've listed some related resources below that could be valuable additions to your library.

So keep coding, keep learning, and keep a curious mind!

Resources

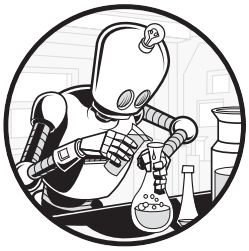
Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship.* Boston: Pearson, 2008.

Mayer, Christian. *The Art of Clean Code.* San Francisco: No Starch Press, 2022.

Petzold, Charles. *Code: The Hidden Language of Computer Hardware and Software.* 2nd ed. Published with the authorization of Microsoft Corporation by Pearson Education, Inc., 2023.

Soshin, Alexey. *Kotlin Design Patterns and Best Practices.* 3rd ed. Birmingham, UK: Packt, 2024.

APPENDIX



This appendix provides an overview of basic computing terms and concepts you'll need to understand before creating programs and apps. It also walks you through setting up your computer so you can run the Kotlin code in projects throughout this book.

If you are new to programming, I recommend reviewing this appendix to be sure you have the necessary background knowledge and workspace setup.

Key Definitions

In this section, I'll introduce a few key terms that are fundamental to understanding what a computer program is and how it works. The code we write relies on a bigger ecosystem of software (other programs such as the operating system and the compiler) and hardware (the computer) to be able to run and generate the desired output.

An *operating system (OS)* is a software program that serves as an intermediary between computer hardware and user applications. Popular operating systems include Microsoft Windows, macOS, Linux, and Android, each tailored to specific devices and computing environments. An OS manages and controls various hardware resources—such as memory, processors, input/output devices, and storage—which provide a standardized interface for software programs to interact with the underlying hardware. It enables multiple applications to run concurrently, ensuring efficient and secure utilization of system resources.

A *computer program*, often referred to as *software* or an *app*, is a set of instructions that directs a computer to perform specific tasks. Programmers write these instructions to solve problems, automate processes, or execute desired computations. Once written, the program is typically translated into *machine code* or an *intermediate code* by a compiler or interpreter to make it executable on a computer. Examples of computer programs include word processors, web browsers, games, and operating systems.

To write computer programs, we use a *programming language*, a set of rules and symbols that can be understood and executed by a computer system. Examples include Kotlin, Python, Java, C/C++, JavaScript, Rust, and Go. Programming languages differ in syntax (the rules for writing programs), semantics (the meaning and behavior of programs), and level of abstraction (how close the language is to the format that the hardware can follow). Each language has its advantages and limitations, making it more or less suitable for different tasks such as web development, system programming, data analysis, or artificial intelligence.

A *library* is a collection of prewritten code modules or functions that programmers can use to perform common tasks. A *standard library* is an integral part of a programming language, bundled with its core distribution to provide the standardized functionality expected to be available on any system. It includes essential modules for tasks like input/output, data manipulation, printing, and networking. Libraries can facilitate reusing code, save time, and allow developers to build on existing solutions.

A *text editor* is a software application designed for creating, modifying, and formatting plain text files. Unlike word processors, which often include advanced formatting and layout features, text editors are focused on the fundamental task of working with unformatted text. Text editors are commonly used by software developers for writing code, as they provide a lightweight and distraction-free environment. Popular text editors include Notepad on Windows, TextEdit on macOS, and Vim on Linux. Also available are editors that have been customized for multiple platforms, such as Notepadd++ and Sublime Text.

An *integrated development environment (IDE)* is a comprehensive software tool designed to streamline and enhance the process of software development. It typically provides a unified environment that integrates various essential tools and features, including a text editor, a debugger, a compiler, and build automation tools. Modern IDEs also include features such as syntax highlighting, code completion, and project management. Popular IDEs include Visual Studio Code, Eclipse, and IntelliJ IDEA, each catering to

specific programming languages and development environments. We'll use IntelliJ IDEA for completing all coding tasks in this book because it offers built-in support for Kotlin.

A *compiler* is a software tool that translates source code written in a high-level programming language into machine code or an intermediate code that can be executed directly by a computer or in various environments, such as the Java Virtual Machine (which we'll discuss shortly). The compilation process involves several stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation.

Debugging is the process of finding and fixing errors (bugs) in a computer program. It involves analyzing the code and using debugging tools and techniques to find the cause of unexpected or incorrect results. Debugging is a systematic and iterative task that requires technical skills, problem-solving abilities, and a good grasp of the software's logic.

The *Java Development Kit (JDK)* is a software development kit used by Java and Kotlin developers for building, testing, and deploying applications. The JDK includes a set of tools and utilities that facilitate programming, including a compiler, a debugger, and other development tools. It also includes the *Java Runtime Environment (JRE)*, which is necessary for running Java/Kotlin applications. The JRE lacks a compiler and thus is used only to run precompiled Java or Kotlin programs. To create new programs, you'll need a JDK.

The *Java Virtual Machine (JVM)* is a crucial component of the JRE and the JDK, serving as an abstraction layer between Java/Kotlin applications and the underlying hardware and operating system. It enables the execution of Java *bytecode*, the compiled form of Java or Kotlin source code. The JVM provides platform independence, allowing Java/Kotlin programs to run on any device with a compatible JVM. It manages memory and security features, as well as provides an execution environment for compatible applications by interpreting bytecode or using just-in-time compilation for optimized performance.

An *executable file* is a stand-alone file that can run on a computer and perform certain tasks. An executable file contains the machine code that the computer's hardware can understand and execute, as well as information about how the code and data are organized in memory. Unlike a text file, an executable file is not human readable, because it is compiled from other files, such as source code files written in languages like Fortran, C, or C++. Examples of executable files are EXE files on Windows and APP files on macOS.

Workflow for Creating an App

Creating a Kotlin app typically involves the following steps:

1. **Set up the project.** Use an IDE like IntelliJ IDEA to create a new Kotlin project. Choose the target platform (such as the JVM) and build system (such as Maven or Gradle). A build system automates compiling,

testing, and packaging code into an executable form. Most of the time, the IDE's default choices work well.

2. **Code in Kotlin.** Use Kotlin to write the app logic. The IDE will typically autosave the code file in a designated folder before running the code or exiting.
3. **Build and compile.** Compile the Kotlin code into bytecode. You can choose to build and run at the same time or to just build the project to check for any build-time errors.
4. **Test.** Check the app's quality and reliability with unit and integration testing. Unit testing ensures that each unit (for example, a function, a method, or a class) works as expected and promotes modular and clear code. Integration testing verifies the interactions and interfaces between components or systems in a larger app.
5. **Debug.** Use the IDE's debugging tools to identify and fix any issues in the code. Set breakpoints, inspect variables, and step through the code to understand how it works. You can insert print statements in various code segments to ensure that the program logic is working as expected.
6. **Deploy.** Package the compiled code and any resources into a deployable format. For this, you can create a Java Archive file from your app and its dependencies. A *Java Archive (JAR)* file is a collection of files that can be run as a Java application on a JVM.

Setting Up Shop

Before you can start working through the chapters in this book and building your own Kotlin applications, you'll need to complete these three steps.

Step 1: Download and Install IntelliJ IDEA

Go to the IntelliJ IDEA download page (since the link to this page can change, find the current site with a quick internet search). Select the **Community Edition**, which is free to use for noncommercial purposes.

Download the *.exe* file for Windows, the *.dmg* file for macOS, or the *.tar.gz* file for Linux. The file will be downloaded to your default download folder. Windows and Mac users can double-click the downloaded file to start the installation; the installation wizard will guide you through the process. Installing IntelliJ IDEA on a Linux device requires additional steps, depending on the type of Linux powering your device. You can look up installation instructions for your specific system online.

Step 2: Download and Set Up the JDK

Next, you'll need to install a JDK. I recommend using a version of the Azul Zulu JDK that comes bundled with the JavaFX graphics library, which will let you work on this book's visualization projects without any further installation steps. I'll guide you through this process and show you an example of how to do it on Windows devices. The steps for Mac devices should be very similar, but you can always look online for more detailed instructions if you need to.

Find the Download Azul JDKs page via a quick internet search. Choose the latest Long-Term Support (LTS) version for Windows. Choose the type of device (CPU)—the most common type at the time of writing is x86-64-bit—and choose **JDK FX** as the Java package type.

WARNING *Be sure to choose JDK FX and and not just JDK as the package type or the installation won't include JavaFX.*

Select ***.msi*** as the file type for downloading. Once you select the file type, the download process should start automatically, and the file should be saved in your default download folder.

Finally, double-click the downloaded file to start the installation. The installation wizard will take you through the process. You don't have to make any changes to the default settings.

Another JDK package that comes bundled with JavaFX is BellSoft Liberica Full JDK. You can install it the same way. Alternatively, you can separately install the core Java SDK and OpenFX, an open source version of JavaFX. Refer to the respective websites for detailed instructions on the installation process and how to configure your IDE to use these installations.

Step 3: Create a New Project

You're now ready to create your very first "Hello, world!" app in Kotlin. For this, you'll need to set up a new project. This section shows you how to proceed on a Windows device. The steps for other systems are very similar.

Launch IntelliJ IDEA by clicking the desktop icon or from the Start menu (you can use the search box to locate the program). In the left panel, Projects should be selected by default (if not, select this). Next, click the **New Project** button in the top right of the window. Your screen should now look similar to Figure A-1.

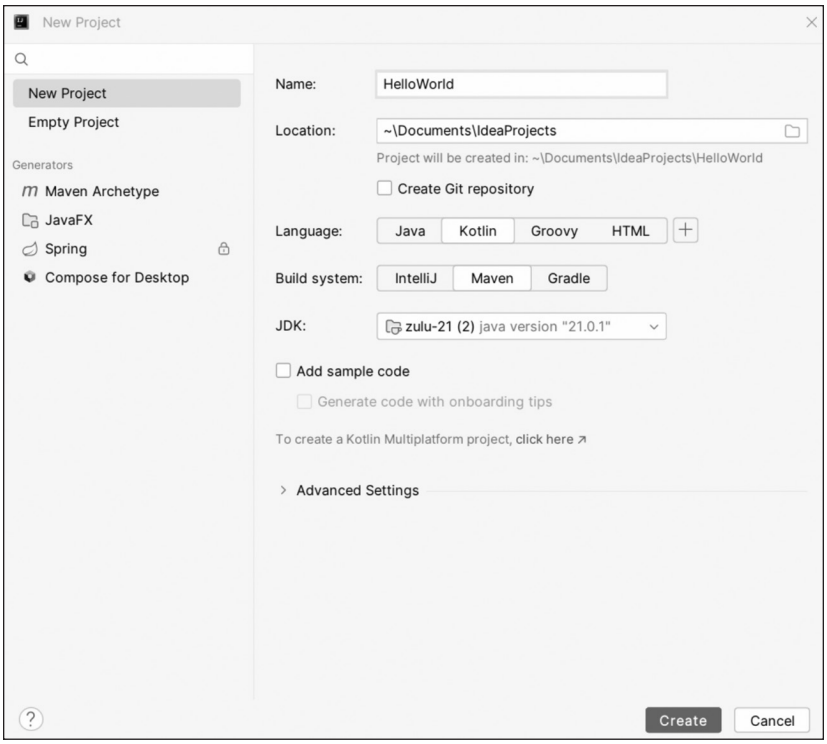


Figure A-1: Setting up a new Kotlin project

Enter **HelloWorld** as the project name (no spaces), and then select **Kotlin** as the Language and **Maven** or **Gradle** as the Build system.

You must also select the JDK required for compiling and running your code. If the installed Azul Zulu JDK is not preselected, open the drop-down menu by clicking the downward-pointing triangle, navigate to the folder where the JDK is installed, and select that folder. The location of the JDK might look like this: *C:\Program Files\Zulu\zulu-xx* (the last two digits represent the version number).

Click **Create** to make the IDE configure a new Kotlin project for you. This should bring you to the next screen, which shows the newly created project "HelloWorld" in the left panel.

Expand the Project tree by repeatedly clicking **>** next to the project and its branch names until you see an empty folder named *kotlin*. Right-click ***kotlin*** and select **New ▸ Kotlin class/File ▸ File**. Your screen should look similar to Figure A-2.

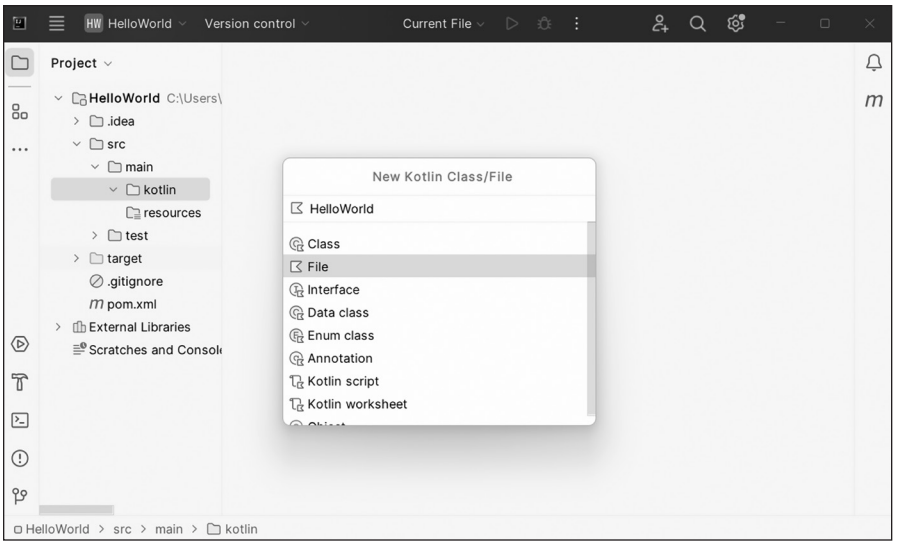


Figure A-2: Creating a Kotlin file

Enter **HelloWorld** in the filename box and press ENTER (the IDE should add *.kt* as an extension to the filename). You should be taken to the IDE code editor window, as shown in Figure A-3.

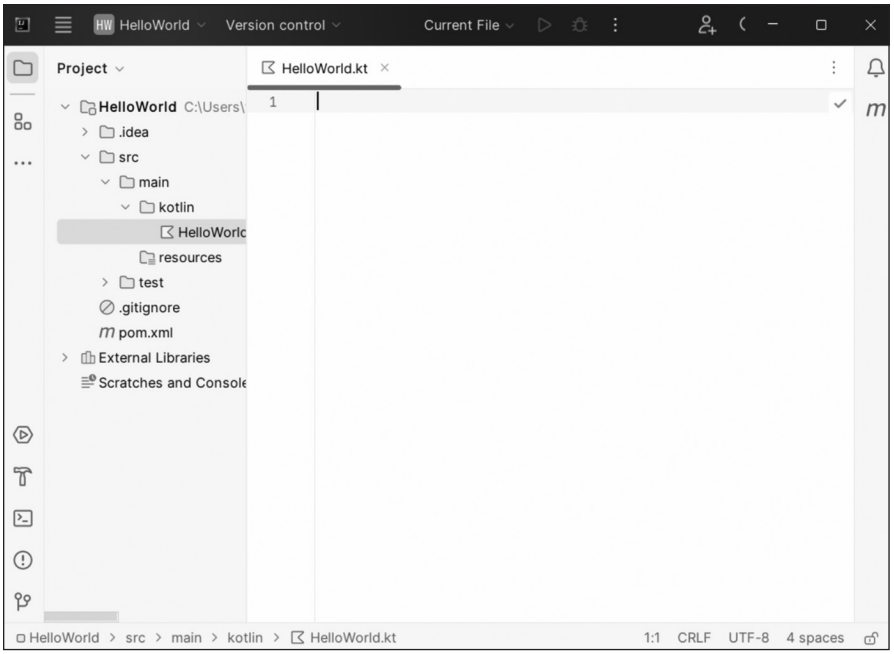


Figure A-3: Starting to code in the code editor window

You're only a few steps away from writing and running your first Kotlin code. Enter the following lines of code in the code editor:

```
fun main() {
    // Print "Hello, world!" in the console.
    println("Hello, world!")
}
```

This code creates the `main()` function, the entry point for any Kotlin program. We use the `fun` keyword to declare a function and define the function's body within a pair of curly brackets.

Inside the function, we use `//` to write a *comment*, a block of text that is ignored by the compiler but tells the human user what a particular line or block of code will do. Finally, we use the `println()` method, which is part of the standard Kotlin library, to print a message in the *console* (a text-based output window).

You can run this code by pressing SHIFT-F10 or by clicking the right-pointing green triangle next to Current File in the top panel. You can also use the Run menu to run this program. Why not try out all these options? Once you've run the program, the screen should look something like that shown in Figure A-4.

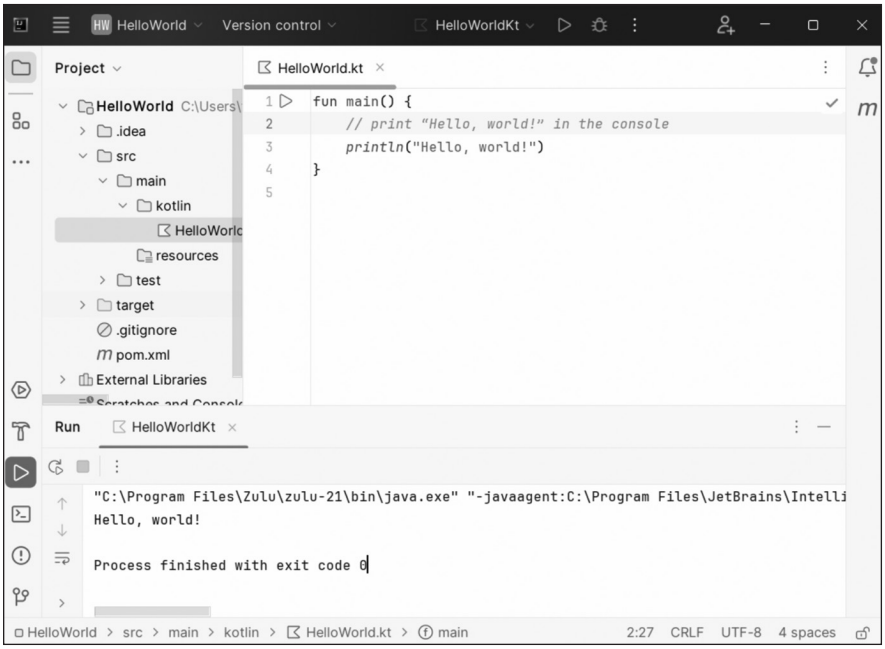


Figure A-4: Printing "Hello, world!" in Kotlin

Notice the console along the bottom of the window. It displays the single line of output produced by this program: `Hello, world!`

Congratulations on successfully composing and running your first Kotlin application! You're now ready to start exploring the world of Kotlin.

INDEX

Page numbers in italics refer to figures and tables.

B

C

D

E

F

G

H

I

M

N

O

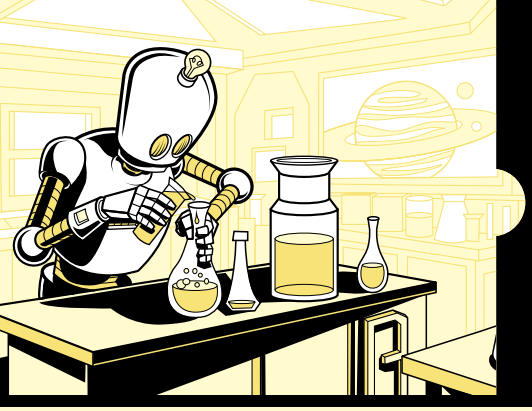
P

Q

R

S

T



LEARN KOTLIN THROUGH 37 PROJECTS

Kotlin isn't just for building Android apps. As you'll learn in *Kotlin from Scratch*, it's also a general programming language for crafting both elegant and efficient code.

With the aid of 37 hands-on projects, you'll move quickly through the language basics while building your problem-solving skills, even tackling advanced concepts like fractals, dynamic systems, and nature-inspired algorithms. You'll explore the way Kotlin handles variables, control structures, functions, classes, and data structures, and you'll learn to create visualizations using Kotlin and the JavaFX graphics library. Then you'll build increasingly sophisticated apps to practice what you've learned while tackling challenges from math and science to algorithms and optimization.

As you progress through the book, you will:

- Simulate physical systems, like the intricate dance of binary stars
- Implement the classic Hill cipher for encryption and decryption
- Generate beautiful fractals with recursive algorithms
- Program classic computer science algorithms for sorting and searching
- Solve the infamous Berlin52 traveling salesman problem

Expand your language repertoire and improve your computational thinking with *Kotlin from Scratch*.

ABOUT THE AUTHOR

Dr. Faisal Islam brings a wealth of interdisciplinary expertise to *Kotlin from Scratch*. With an MS in civil engineering and a PhD in resource economics, he offers a unique perspective on applying computational thinking to real-world challenges. Dr. Islam has over 20 years of experience in coding across multiple languages (C, Java, Python, and Kotlin) to solve complex problems and an extensive background in simulation, modeling, and optimization.

Covers Kotlin 2.*x*

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™

nostarch.com