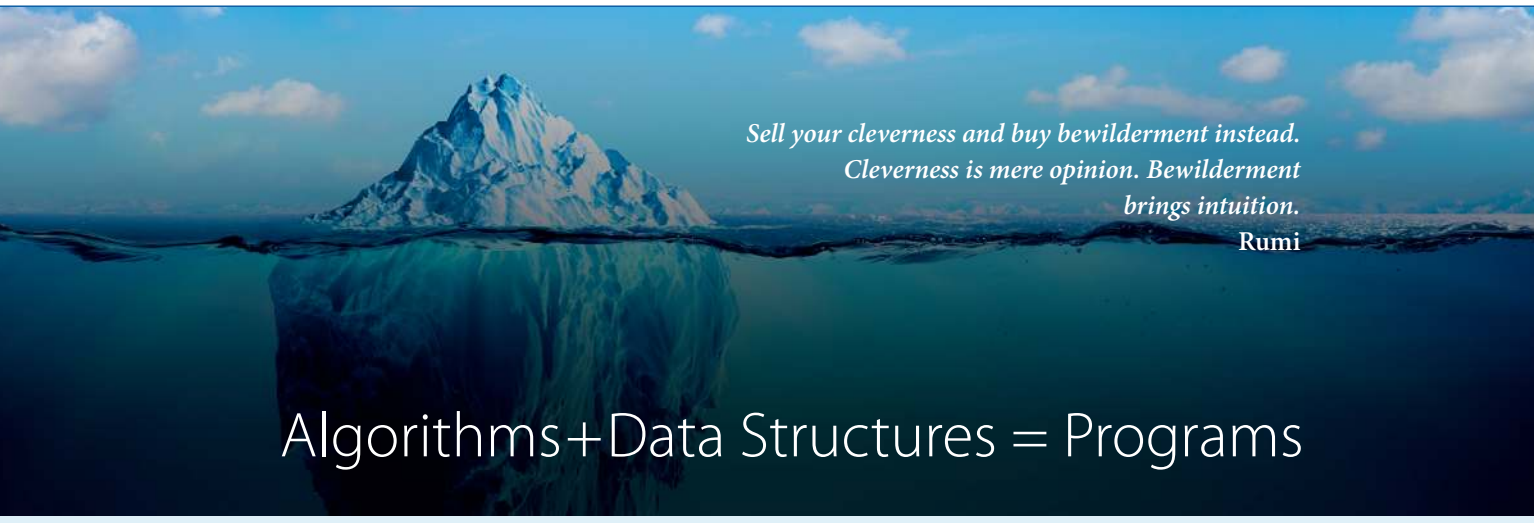


Sell your cleverness and buy bewilderment instead.
Cleverness is mere opinion. Bewilderment
brings intuition.
Rumi

Algorithms+Data Structures = Programs

OVERVIEW

There's a perception that computer science is constantly changing, and in some respects that's true: There are always new languages, frameworks, and application domains rising up and old ones sinking down. All of this change that we see around us, though, is like the *top part of an iceberg*. The most visible elements of our field are built upon and supported by a deeper layer of knowledge that's mostly invisible to the casual observer. This book is about what's under the water, *the fundamental things that make programming possible*, even if we don't see them right away.

LEARNING OBJECTIVES

This chapter introduces the topics ahead. By the end of it, you'll understand:

- How algorithms and data structures work together to create programs.
- Key points from the history of algorithms that are still relevant to us today and the definition of an algorithm.
- Why data structures and algorithms matter to real-world programmers.
- Suggestions for improving your own learning and tips for teaching yourself.

Goals

Consider just a few examples of challenging real-world programming problems:

- predicting moves for a game-playing program;
- implementing a compiler for a new programming language;
- creating digital special effects for an animated movie;
- generating procedural content in an open-world game;
- routing data through the network of devices that make up the Internet; and
- building a large language AI model like ChatGPT.

These problems are all different, but in order to solve them you need an understanding of both **algorithms** and **data structures**. You've probably encountered both of these terms before, even if you haven't studied them in depth yet. An algorithm, in plain words, is a precisely specified procedure for solving a problem. Computer programs are algorithms, but the term could be applied to any series of steps that accomplishes a goal, from assembling modular furniture to baking a key lime pie. Data structures are standard ways of organizing a program's information so that it can be easily accessed and manipulated. You've probably already encountered some standard data structures, like arrays in Java or C, or lists and dictionaries in Python.

With that in mind, this book focuses on three major topics:

- A set of *fundamental data structures* that are essential for every computer scientist and programmer. Each of the data structures we'll cover – including lists, stacks, queues, hash tables, trees, and graphs – represents a particular way of arranging and manipulating data, and they each have their own strengths and important applications. These are the most important data structures that occur over and over again throughout computer science.
- A collection of *standard algorithmic techniques* that are building blocks of more complex programs, including recursion, sorting, backtracking, and hashing.
- A *framework for comparing algorithms*. If two methods solve the same problem, can we say that one is "better" in a rigorous way? This question is addressed by the techniques of **algorithm analysis**, which we'll use to evaluate the quality of our solutions.

Finally, it's not enough to simply understand data structures theoretically: you have to use them! Each chapter of this book will show you how to implement data structures and use them in Java applications. Some chapters feature larger stand-alone projects that you can feature on a resume or project portfolio.

A Brief History of Algorithms

The concept of an algorithm has existed since ancient times, although the term itself did not come into its modern use until the nineteenth century. Clay tablets from Mesopotamia (approximately 2000–1800 BCE) discuss procedures for performing arithmetic by hand and solving equations that are relevant to agriculture and building (Knuth, 1972). Greek mathematicians described methods for finding roots and divisors, estimating π, and solving some classes of equations.

What, though, is an algorithm, really? Scientists have wrangled over a formal definition, but it's generally agreed that a procedure must meet some criteria in order to be called an algorithm (Shaffer, 1997; Cormen et al., 2022):

- It must have *well-defined inputs and outputs*.
- Each step must be *concrete and feasible*. That is, there should be no ambiguity about what is to be performed at each step and each step must be something that a computer (or other agent executing the algorithm) can actually do. Further, the order of the steps must be unambiguous.

- It must be *correct*. For its given inputs, the algorithm must produce correct outputs.
- It should have only a *finite* number of instructions. Programs can contain loops or other control structures, but it has to be possible to write the algorithm down in a form that allows it to be executed.
- It must *terminate*. The running time may be infeasibly large, but the method must eventually end and produce an output.

Try It Yourself

The physicist Richard Feynman is credited with the following algorithm for solving any problem:

1. Write down the problem.
2. Think very hard.
3. Write down the solution.

Explain why Feynman's method, clever though it may be, is not really an algorithm.

The term **algorithm** itself is derived from the ninth-century Islamic Persian scholar Muḥammad ibn Mūsā al-Khwārizmī. The name al-Khwārizmī means "from Khwarazm," which is located in present-day Uzbekistan. He was a true polymath, active in all the scientific fields of his day, but his legacy is connected to writing on mathematics. His most important work is *al-Kitāb al-Mukhtaṣar fī Ḥisāb al-Jabr wal-Muqābalah*, translated as *The Compendious Book of Calculation by Completion and Balancing*, which provided general methods for solving quadratic equations. The book named the field of algebra, from *al-Jabr*, meaning "completion," which refers to the process of moving terms between the two sides of an equation (Gandz, 1926). al-Khwārizmī also wrote texts describing arithmetic using the now-standard system of Indian-Arabic numerals, which were compiled and translated into Latin as *Algoritmi de numero Indorum* – "Algoritmi on the Indian numbers." In Europe, the term *algorismus* came to refer to the techniques for doing calculation on decimal numbers; by the nineteenth century it had acquired its modern form and meaning.

Although mathematicians and engineers continued to develop new computational techniques, algorithms didn't emerge as a distinct field of study until the post-World War II period and the development of electronic computers. The early pioneers of computer science began to investigate not just algorithms for specific problems, but the design and evaluation of algorithms as its own field of study. The study of data structures as a distinct subject developed as computer science and software engineering matured in the 1960s.

Programming in the earliest days was done in low-level machine languages that gave programmers a great deal of control but lacked support for abstractions such as variables. As a result, it was often hard to reason about the correctness of programs, and debugging was painful and time-consuming. By the late 1960s, a group of computer scientists led by Edsger Dijkstra began to advocate for **structured programming** in "high-level" languages like Fortran, ALGOL, and C. These languages were more abstract than machine language and allowed programmers to think more about the *meaning* of a program and how that meaning should be best expressed

in code. Along with these ideas came an increased focus on the relationship between a program's code and the data that it operates on. The Swiss computer scientist Niklaus Wirth described it like this (Wirth, 1976):

> *Programs*, after all, are concrete formulations of abstract *algorithms* based on particular representations and structures of *data*.

Data structures are important because they provide the basis on which the program's algorithm executes. This insight led to considerable research into the best ways to organize information in programs, which by the 1980s had become a core part of the computer science curriculum.

Why Do Data Structures and Algorithms Matter to Real-World Programmers?

"That sounds interesting," you may say, "but do I need to learn this material to be a highly paid software professional?" I'll tell you the truth, reader: *You don't have to learn any of this stuff to be a working software developer*. Even if you can get paid without reading this book, though, there are still good reasons to spend time engaging with this material, whether you're doing that in a formal course or for self-study.

- Algorithms and data structures are *foundational to all of programming*. Foundational knowledge, in any field, is important because it's transferable. If you understand, for example, how to use a data structure like a hash table,[1] that knowledge can then be applied to any language, framework, or problem. Mastering the material in this book will make it easier for you to see the connections and patterns that reoccur over and over again throughout computer science.
- Second, as your career advances, you'll eventually encounter hard problems. The closer you get to the cutting edge of the field – in areas like AI, scientific computing, or programming language design – the less your success depends upon knowing a specific language or tool and the more it depends on having *strong core computer science skills*.
- If you play sports, you've probably spent time in the gym lifting weights or stretching as preparation for training on the field. Athletes train muscles and movements that aren't part of their sport because they want to be strong and injury-free. In the same way, knowing algorithms and data structures will help you *avoid common design mistakes*. In particular, algorithm analysis will help you avoid wasting time from choosing bad, inefficient solutions that can't scale.
- Finally, like a jazz musician building chops by practicing technical exercises and solos by other musicians, working through the projects in this book will *make you better at programming* and prepare you for larger projects.

1 Covered in Chapters 14–16.

Learning from This Book

If you're using this book for a course, your instructor will provide you with a schedule of suggested readings and assignments. If you're a working programmer using it to teach yourself, then you have more freedom to explore the material at your own pace. In either case, here are some tips for getting the most out of this (or any) text.

Orient yourself. Read the introduction and section headings for a chapter and read the learning goals in the introduction, which will give you an overview of the intended outcomes. When you finish a chapter, return and review its outcomes.

Focus on the problem. Data structures and algorithms exist to solve interesting problems. When you're learning a new technique, always ask yourself, "What applications benefit from this?"

Use active learning. Every chapter contains *Try It Yourself* sections integrated into the text that allow you to think about the parts you've just read. When you encounter these, stop and think about them. Don't immediately move on to the solution without trying the question for a few minutes. Read the example programs carefully and reflect on how they work.

Implement the projects. The larger example projects are a key element of this book. If you're a relatively new Java programmer, they'll show you how to use the language's features and develop your coding style and organization skills. Resist the temptation to simply copy the project code and then run it – build the projects step by step and focus on the reasoning behind each implementation.

Do the exercises but don't get stuck. Each chapter ends with several exercises and extensions you can try. These are helpful, but it isn't necessary to do every one. Pick the ones that seem interesting. I recommend doing most of the "Understand" questions, several of the "Apply" questions, and at least a few of the more difficult "Extend" questions. If you get stuck on a question, give it a fair try then move on to something else. You'll likely find that returning to it later, after you have some more practice, will help you get unstuck. If you're using this book as part of a class, don't be afraid to ask your professor or teaching assistants for help, even on questions that aren't assigned to you.[2]

Revisit and compare. As you work through each topic, think about how it relates to the other topics you've already seen. For example, the early chapters of the book cover arrays and lists, which are similar to each other, but used for different applications. As you complete each new topic, read back through the introduction and ending sections of earlier chapters and think about how what you've just learned is similar to and different from the earlier material. Doing this will help you build a richer mental map of the important concepts and their relationships.

Try It Yourself

Think about your previous programming experience. What worked well for you when you were learning new concepts? What didn't work well?

2 We like it when you do this.

SUMMARY

This chapter has introduced our major theme: data structures and algorithms working together to make programs possible. As you're reading, consider the following points:

- There is often no "best" solution to a particular programming problem, but rather multiple solutions with their own trade-offs. Understanding data structures and algorithm analysis will help you evaluate these trade-offs in an intelligent way.
- Even if you use higher-level libraries and frameworks, understanding the fundamentals of computer science will improve your programming and help you learn.
- The concepts from this book are relevant to every area of computer science, including programming language design, systems, AI, and networking.

We'll start the next chapter with an introduction to Java.

EXERCISES

Understand

1. List three examples of algorithms that aren't programming or cooking related.
2. Give an example of a procedure that isn't an algorithm by our criteria.
3. Other than execution time, what are some qualities we might consider in determining whether an algorithm is good or not?
4. Sorting data is an important problem and many sorting algorithms have been developed. What is a real-world problem that requires sorting data?
5. Consider your previous programming experience. What data structures and algorithms have you already studied?
6. Do some research on the difference between active and passive learning. What makes learning active?

Apply

7. I wrote an algorithm that uses a `random` function to make a choice. Explain why the program is still an algorithm, even if the output is not the same every time it runs.
8. I wrote another program that runs and terminates but doesn't display any output or save any results. Is that an algorithm?
9. Think about a classic 2D video game like *Pac-Man*. What data do you need to keep track of for the different elements on the screen? What about a modern 3D game?
10. Many languages support a basic data type for character strings. List some common operations that you would expect to perform on character strings.
11. Explain why any program executing on a computer is automatically composed of concrete, feasible steps. Tip: Think about what happens when a program runs. What is the CPU actually doing?

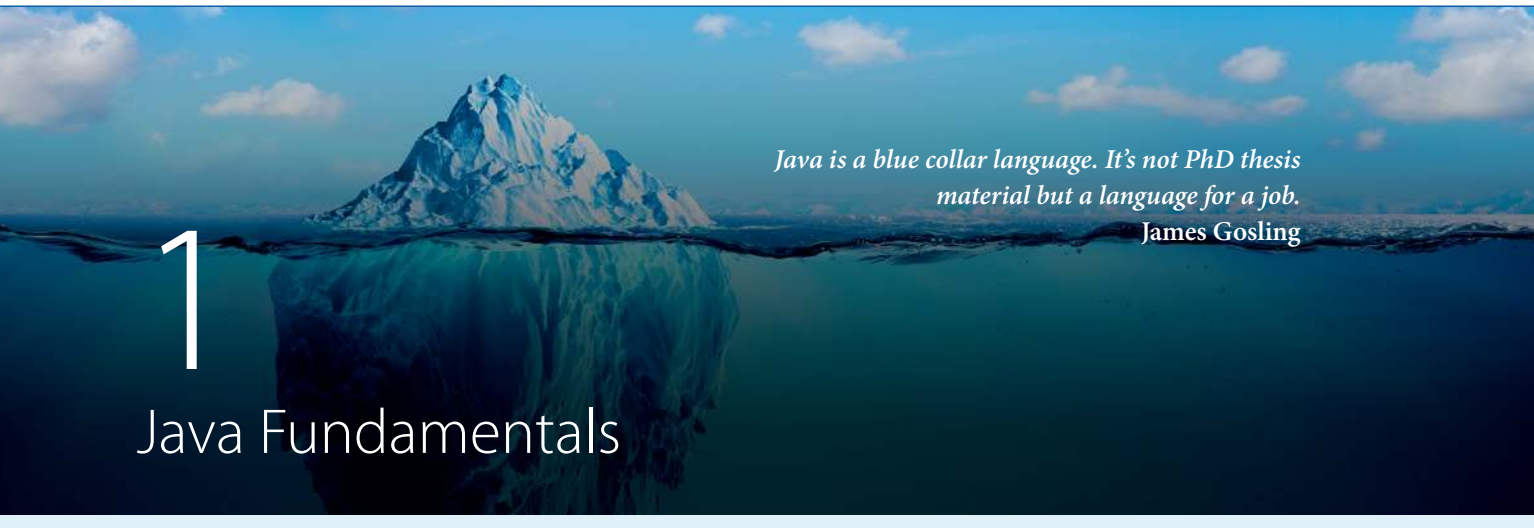
12. Do some research on genealogies. What kind of data structure could you consider using to model a person's family history?
13. Do some research on social networks. What kind of data structure might you use to model the connections between users on a social media platform?

Extend

14. **Bloom's taxonomy** is a model for structuring learning goals based on their complexity. It arranges engagement with a topic into a hierarchy, where the lowest level is memorizing basic facts and the highest is creating new original knowledge. Do some research on Bloom's taxonomy, then give an example of a question or project about algorithms that fits into each of its categories.
15. Do some research on difficult problems. Identify and describe one problem for which there is no known efficient algorithm.
16. Look up the word "metacognition." How does metacognition apply to learning?
17. Think about your own learning process in programming or another area. What lessons have you learned about how **you** learn best?

NOTES AND FURTHER READING

The title of this chapter is a reference to Niklaus Wirth's book, *Algorithms + Data Structures = Programs* (Wirth, 1976). Wirth designed Pascal, one of the most important programming languages of the 1970s and 1980s, and contributed to the development of object-oriented programming with the Modula family of languages. The book was influential and widely used in education. It contains a great overview of building a tiny compiler for a small Pascal-like language. After you finish this book, there are a number of other excellent resources you can read. Cormen et al.'s *Introduction to Algorithms* is a classic upper-level book that covers algorithm design, analysis, and advanced data structures (Cormen et al., 2022). Skiena's *Algorithm Design Manual* features a catalog of important problems and approaches for tackling each one, along with a number of entertaining stories about his experience designing algorithms for real-world problems (Skiena, 1998).



Java is a blue collar language. It's not PhD thesis material but a language for a job.
James Gosling

1 Java Fundamentals

INTRODUCTION

Java is one of the world's most popular programming languages. Widely used in enterprise software development, Java's strengths lie in its combination of performance and portability, as well as its large, robust library of built-in features, which allow developers to create complex applications entirely within the language. Java was developed in the early 1990s by a team from Sun Microsystems led by James Gosling. Initially called Oak (after a tree outside Gosling's office), the new language was intended to be a development environment for interactive TV, but pivoted to the emerging World Wide Web after its public release in 1995. Since then, Java has expanded into almost every area of software development. It is the default programming language for Android mobile devices, the Hadoop large-scale data processing system, and *Minecraft*. Java is one of the most well-known **object-oriented** programming languages.

This chapter surveys the core elements of Java programming, assuming some familiarity with programming in any language. If you already have Java experience, it will be a refresher on important points. If your experience is with Python, JavaScript, or other languages, this chapter will help you understand how Java does things differently.

LEARNING OBJECTIVES

By the end of this chapter you'll be able to

- Write programs using the core elements of Java: variables, types, conditionals, loops, and methods.
- Use built-in classes from the standard library to represent text, read input, and do calculations.
- Combine these features to implement simulation programs and historical cryptographic algorithms.

The next chapter extends these fundamentals and focuses on object-oriented programming. After completing both chapters, you'll be well-prepared to move forward with the rest of the book.

1.1 Hello, Java!

Let's write some code! This section will show you how to write your first Java program.

1.1.1 The First Program

The traditional first program prints "Hello, World!" to the screen. This may seem trivial, but simply coding and running this program verifies that it's possible to compile and run a valid program that produces output. Copy the code below to a file named `HelloWorld.java` in your Java environment, run it, and verify that it produces the expected output.

```
1  // The first program: print a hello message
2
3  // All Java code is contained in a class
4  public class HelloWorld {
5
6    // Main is the entry point for the program
7    public static void main(String[] args) {
8
9      // Print a message to the standard output
10     System.out.println("Hello, World!");
11   }
12 }
```

Every Java program is enclosed in a `class` block. The class name is the name of the program, and the name of the class must match the name of the `.java` file that contains it. This class is named `HelloWorld`, so it must be in a file named `HelloWorld.java`. By convention, *class names always start with an uppercase letter*. Multi-word names are created by capitalizing each word. The top of the program also illustrates Java's basic comment: Two forward slashes tells the compiler to ignore everything on the same line. Java also supports multiline comments; we'll see an example shortly.

Every Java application must contain one method called `main`, which is the *entry point* for the program. When `HelloWorld` executes, it begins at the first line of `main`. Java is more verbose than Python, and programming the `main` method requires chanting an invocation to the Java verbosity gods. That invocation is:

```
public static void main(String[] args)
```

The signature for `main` must contain all of these keywords. For now, don't worry about the meaning of `public`, `static`, and `void` – we'll come to them soon.[1] The `main` method always takes one input argument, a `String[]` called `args`, used to pass command-line arguments into

1 `void` will be discussed in Section 1.6; `public` and `static` will be covered in Chapter 2.

the program.[2] Note that `args` is required even if the program doesn't use any command-line arguments.

Like its C/C++ ancestors, Java uses curly braces to mark blocks of code. We prefer to place the left curly brace on the same line as the block declaration, but other style guides place it on the next line (Google, 2022a). Whitespace is not syntactically significant in Java – unlike Python, Java does not require spacing to indicate the structure of your program – but you should always *indent each new block* to show the logical structure of the program.

The basic printing method is `System.out.println`, which outputs a string to the terminal and then moves to the next line. The input is a text string, denoted using *double quotes*; Java does not allow single-quoted strings.

```
System.out.println("Hello, World!");
```

Every Java statement is terminated by a semicolon. `System` is a special built-in Java object that provides access to the computer's operating system and lower-level utility methods. Every Java program automatically has access to `System` and its methods. Notice that `System` starts with a capital `S`, which is required.

Try It Yourself

- Write a new program called `Haiku.java` that contains a class called `Haiku`. Use three print statements to output this haiku by the poet Kobayashi Issa, famous for his poems about insects and small creatures:

  little snail,
  inch by inch –
  climb Mount Fuji!

- Java's printing supports the standard set of special characters: `\n` for a newline, `\t` for a tab, `\"` for a literal double quote within a string, and `\\` for a literal backslash. Use multiple print statements and `\"` to print this version of *The Raven* as a limerick (Doctorow, 2007). Put your program in a file named `Raven.java` in a class named `Raven`.

  There once was a girl named Lenore,
  And a bird, and a bust, and a door,
  And a guy with depression,
  And a whole lot of questions,
  And the bird always says, "Nevermore."

2 Programs that run in a Linux shell may get inputs this way.