ATOMIC KOTLIN

Bruce Eckel

Svetlana Isakova

Atomic Kotlin

Bruce Eckel and Svetlana Isakova

This book is for sale at http://leanpub.com/AtomicKotlin

This version was published on 2021-11-22

ISBN 978-0-9818725-4-4

Contents

About this Sample

This is a free sample of the book *Atomic Kotlin*. The book is available in both eBook and print form at www.AtomicKotlin.com[1].

This sample contains a limited number of full atoms (chapters) and partial atoms.

Leanpub

Before purchasing the book, you are strongly encouraged to install this sample on your reading device of choice to ensure that you have no problems.

This sample (and the complete book) is available in multiple eBook formats:

- PDF for reading on computers; includes hyperlinked references to other atoms.
- MOBI for reading on Kindle devices, or the Kindle app.
- EPUB for reading on iPads and other Apple devices (or computer EPUB readers).
- You can also read the book, in your browser, directly from the Leanpub site.

Leanpub provides instructions on how to install an eBook onto your device:

- Apps and Devices for Reading Leanpub Books[2]
- Overview[3]
- Kindle[4]
- iPad/iOS[5]

[1] https://www.AtomicKotlin.com
[2] http://help.leanpub.com/en/articles/3045130-how-can-i-read-a-leanpub-ebook-what-apps-or-devices-do-you-recommend
[3] http://help.leanpub.com/en/articles/117470-after-i-buy-a-leanpub-book-how-can-i-get-it-onto-my-device
[4] http://help.leanpub.com/en/articles/110746-how-do-i-get-leanpub-books-on-my-kindle
[5] http://help.leanpub.com/en/articles/3972511-how-can-i-read-a-leanpub-book-on-apple-s-books-app

We have tested the eBook on these platforms:

- iPad (Open the EPUB file using the Safari browser, then select "Open in iBooks").
- Kindle Fire (See above link).
- Kindle Paperwhite (See above link).
- Google Play Books[6] (Add the EPUB file to your Books library).

Stepik

Stepik provides a separate, browser-based eBook product which allows you to interactively edit, compile and run the inline book examples directly within your browser, as long as you have an Internet connection. The interactive eBook (with its own free sample) can be found here: Stepik[7]. Please note that the Stepik product is not part of the Leanpub product.

Complete Table of Contents

[6] https://play.google.com/books
[7] https://stepik.org/course/15001

- Looping & Ranges
- The `in` Keyword
- Expressions & Statements
- Summary 1
- Section II: Introduction to Objects
- Objects Everywhere
- Creating Classes
- Properties
- Constructors
- Constraining Visibility
- Packages
- Testing
- Exceptions
- Lists
- Variable Argument Lists
- Sets
- Maps
- Property Accessors
- Summary 2
- Section III: Usability
- Extension Functions
- Named & Default Arguments
- Overloading
- `when` Expressions
- Enumerations
- Data Classes
- Destructuring Declarations
- Nullable Types
- Safe Calls & the Elvis Operator
- Non-Null Assertions
- Extensions for Nullable Types
- Introduction to Generics
- Extension Properties

- `break` & `continue`
- Section IV: Functional Programming
- Lambdas
- The Importance of Lambdas
- Operations on Collections
- Member References
- Higher-Order Functions
- Manipulating Lists
- Building Maps
- Sequences
- Local Functions
- Folding Lists
- Recursion
- Section V: Object-Oriented Programming
- Interfaces
- Complex Constructors
- Secondary Constructors
- Inheritance
- Base Class Initialization
- Abstract Classes
- Upcasting
- Polymorphism
- Composition
- Inheritance & Extensions
- Class Delegation
- Downcasting
- Sealed Classes
- Type Checking
- Nested Classes
- Objects
- Inner Classes
- Companion Objects
- Section VI: Preventing Failure

- Exception Handling
- Check Instructions
- The `Nothing` Type
- Resource Cleanup
- Logging
- Unit Testing
- Section VII: Power Tools
- Extension Lambdas
- Scope Functions
- Creating Generics
- Operator Overloading
- Using Operators
- Property Delegation
- Property Delegation Tools
- Lazy Initialization
- Late Initialization
- Appendices
- Appendix A: AtomicTest
- Appendix B: Java Interoperability

Copyright

Atomic Kotlin

By Bruce Eckel, President, MindView, LLC, and Svetlana Isakova, JetBrains sro.

eBook ISBN 978-0-9818725-4-4

Version 1.0: December 2020

Version 1.1: November 2021

Print Book ISBN 978-0-9818725-5-1

First printing: January 2021

Second printing: November 2021

November 2021 updates include adjustments for Kotlin 1.5, and corrections.

The eBook ISBN covers the Leanpub and Stepik eBook distributions, both available through *www.AtomicKotlin.com*.

Please purchase this book through *www.AtomicKotlin.com*, to support its continued maintenance and updates.

Created in Crested Butte, Colorado, USA, and Munich, Germany.

Text printed in the United States.

Cover design by Daniel Will-Harris, www.Will-Harris.com[8]

Visit us at *www.AtomicKotlin.com*.

Source Code

All the source code for this book is available as copyrighted freeware, distributed via Github[10]. To ensure you have the most current version, this is the official code distribution site. You may use this code in classroom and other educational situations as long as you cite this book as the source.

The primary goal of this copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code without permission. (As long as this book is cited, using examples from the book in most media is generally not a problem.)

In each source-code file you find a reference to the following copyright notice:

[8] http://www.Will-Harris.com
[9] https://kotlinlang.org/foundation/kotlin-foundation.html
[10] https://github.com/BruceEckel/AtomicKotlinExamples

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

Section I: Programming Basics

There was something amazingly enticing about programming—**Vint Cerf**

This section is for readers who are learning to program. If you're an experienced programmer, skip forward to Summary 1 and Summary 2.

Introduction

> This book is for dedicated novices and experienced programmers.

You're a novice if you don't have prior programming knowledge, but "dedicated" because we give you just enough to figure it out on your own. When you're finished, you'll have a solid foundation in programming and in Kotlin.

If you're an experienced programmer, skip forward to Summary 1 and Summary 2, then proceed from there.

The "Atomic" part of the book title refers to atoms as the smallest indivisible units. In this book, we try to introduce only one concept per chapter, so the chapters cannot be further subdivided—thus we call them *atoms*.

Concepts

All programming languages consist of features. You apply these features to produce results. Kotlin is powerful—not only does it have a rich set of features, but you can usually express those features in numerous ways.

If everything is dumped on you too quickly, you might come away thinking Kotlin is "too complicated."

This book attempts to prevent overwhelm. We teach you the language carefully and deliberately, using the following principles:

1. **Baby steps and small wins**. We cast off the tyranny of the chapter. Instead, we present each small step as an *atomic concept* or simply *atom*, which looks like a tiny chapter. We try to present only one new concept per atom. A typical atom contains one or more small, runnable pieces of code and the output it produces.
2. **No forward references**. As much as possible, we avoid saying, "These features are explained in a later atom."

3. **No references to other programming languages**. We do so only when necessary. An analogy to a feature in a language you don't understand isn't helpful.
4. **Show don't tell**. Instead of verbally describing a feature, we prefer examples and output. It's better to see a feature in code.
5. **Practice before theory**. We try to show the mechanics of the language first, then tell why those features exist. This is backwards from "traditional" teaching, but it often seems to work better.

If you know the features, you can work out the meaning. It's usually easier to understand a single page of Kotlin than it is to understand the equivalent code in another language.

Where Is the Index?

This book is written in Markdown and produced with Leanpub. Unfortunately, neither Markdown nor Leanpub supports indexes. However, by creating the smallest-possible chapters (atoms) consisting of a single topic in each atom, the table of contents acts as a kind of index. In addition, the eBook versions allow for electronic searching across the book.

Cross-References

A reference to an atom in the book looks like this: Introduction, which in this case refers to the current atom. In the various eBook formats, this produces a hyperlink to that atom.

Formatting

In this book:

- *Italics* introduce a new term or concept, and sometimes emphasize an idea.

- `Fixed-width font` indicates program keywords, identifiers and file names. The code examples are also in this font, and are colorized in the eBook versions of the book.
- In prose, we follow a function name with empty parentheses, as in `func()`. This reminds the reader they are looking at a function.
- To make the eBook easy to read on all devices and allow the user to increase the font size, we limit our code listing width to 47 characters. At times this requires compromise, but we feel the results are worth it. To achieve these widths we may remove spaces that might otherwise be included in many formatting styles—in particular, we use two-space indents rather than the standard four spaces.

"Pause"

Occasionally you will see:

- -

This indicates a pause, or a kind of small reset. In this book it often appears before a brief summary of the current subsection, but where a "Summary" subhead would be overkill. Some books use a mechanism like this to indicate that an idea is complete and we are starting something new, but it's still within the same topic and not big enough to warrant a subsection or a new section. The markdown in Leanpub is quite limited, and using one or more dots (my original attempt) isn't possible. Putting two dashes in the markdown produces a dot and a dash. There might be a better way to do this but I haven't found it, so I settled on that.

Sample the Book

We provide a free sample of the electronic book at *AtomicKotlin.com*. The sample includes the first two sections in their entirety, along with several subsequent atoms. This way you can try out the book and decide if it's a good fit for you.

The complete book is for sale, both as a print book and an eBook. If you like what we've done in the free sample, please support us and help us continue our work

by paying for what you use. We hope the book helps, and we appreciate your sponsorship.

In the age of the Internet, it doesn't seem possible to control any piece of information. You'll probably find the electronic version of this book in numerous places. If you are unable to pay for the book right now and you do download it from one of these sites, please "pay it forward." For example, help someone else learn the language once you've learned it. Or help someone in any way they need. Perhaps in the future you'll be better off, and then you can pay for the book.

Exercises and Solutions

Most atoms in *Atomic Kotlin* are accompanied by a handful of small exercises. To improve your understanding, we recommend solving the exercises immediately after reading the atom. Most of the exercises are checked automatically by the JetBrains IntelliJ IDEA integrated development environment (IDE), so you can see your progress and get hints if you get stuck.

You can find the following links at http://AtomicKotlin.com/exercises/[11].

To solve the exercises, install IntelliJ IDEA with the Edu Tools plugin by following these tutorials:

1. Install IntelliJ IDEA and the EduTools Plugin[12].
2. Open the Atomic Kotlin course and solve the exercises[13].

In the course, you'll find solutions for all exercises. If you're stuck on an exercise, check for hints or try peeking at the solution. We still recommend implementing it yourself.

If you have any problems setting up and running the course, please read The Troubleshooting Guide[14]. If that doesn't solve your problem, please contact the support team as mentioned in the guide.

If you find a mistake in the course content (for example, a test for a task produces the wrong result), please use our issue tracker to report the problem with this pre-filled

[11] http://AtomicKotlin.com/exercises/
[12] https://www.jetbrains.com/help/education/install-edutools-plugin.html
[13] https://www.jetbrains.com/help/education/learner-start-guide.html?section=Atomic%20Kotlin
[14] https://www.jetbrains.com/help/education/troubleshooting-guide.html

form[15]. Note that you'll need to log in into YouTrack. We appreciate your time in helping to improve the course!

Seminars

You can find information about live seminars and other learning tools at *AtomicKotlin.com*.

Conferences

Bruce creates *Open-Spaces* conferences such as the Winter Tech Forum[16]. Join the mailing list at *AtomicKotlin.com* to stay informed about our activities and where we are speaking.

Support Us

This was a big project. It took time and effort to produce this book and accompanying support materials. If you enjoy this book and want to see more things like it, please support us:

- **Blog, tweet, etc. and tell your friends**. This is a grassroots marketing effort so everything you do will help.
- **Purchase an eBook or print version** of this book at *AtomicKotlin.com*.
- **Check** *AtomicKotlin.com* for other support products or events.

About Us

Bruce Eckel is the author of the multi-award-winning *Thinking in Java* and *Thinking in C++*, and a number of other books on computer programming including

[15] https://youtrack.jetbrains.com/newIssue?project=EDC&summary=AtomicKotlin%3A&c=Subsystem%20Kotlin&c=
[16] http://www.WinterTechForum.com

Atomic Scala[17]. He's given hundreds of presentations throughout the world and puts on alternative conferences and events like the Winter Tech Forum[18] and developer retreats. Bruce has a BS in applied physics and an MS in computer engineering. His blog is at www.BruceEckel.com[19] and his consulting, training and conference business is Mindview LLC[20].

Svetlana Isakova began as a member of the Kotlin compiler team, and is now a developer advocate for JetBrains. She teaches Kotlin and speaks at conferences worldwide, and is coauthor of the book *Kotlin in Action*.

Acknowledgements

- The Kotlin Language Design Team and contributors.
- The developers of Leanpub, which made publishing this book *so* much easier.
- James Ward for converting the Gradle build to Kotlin, and being generally awesome.

Dedications

For my beloved father, E. Wayne Eckel. April 1, 1924—November 23, 2016. You first taught me about machines, tools, and design.

For my father, Sergey Lvovich Isakov, who passed away so early and who we will always miss.

About the Cover

Daniel Will-Harris[21] designed the cover based on the Kotlin logo.

[17] http://www.atomicscala.com/
[18] http://www.WinterTechForum.com
[19] http://www.BruceEckel.com
[20] https://www.mindviewllc.com/
[21] http://www.will-harris.com

Why Kotlin?

> *Programs must be written for people to read, and only incidentally for machines to execute.*—**Harold Abelson**, coauthor, *Structure and Interpretation of Computer Programs.*

This atom is an overview of the historical development of programming languages so you can understand where Kotlin fits and why you might want to learn it. We introduce some topics which, if you are a novice, might seem too complicated right now. Feel free to skip this atom and come back to it after you've read more of the book.

Programming language design is an evolutionary path from serving the needs of the machine to serving the needs of the programmer.

A programming language is invented by a language designer and implemented as one or more programs that act as tools for using the language. The implementer is usually the language designer, at least initially.

Early languages focused on hardware limitations. As computers become more powerful, newer languages shift toward more sophisticated programming with an emphasis on reliability. These languages can choose features based on the psychology of programming.

Every programming language is a collection of experiments. Historically, programming language design has been a succession of guesses and assumptions about what will make programmers more productive. Some of those experiments fail, some are mildly successful and some are very successful.

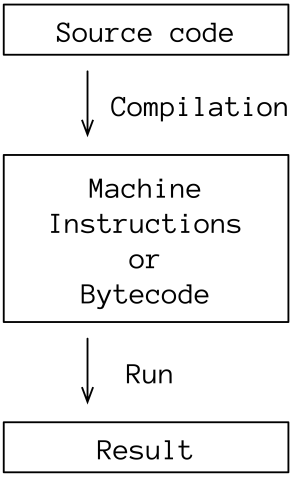
We learn from the experiments in each new language. Some languages address issues that turn out to be incidental rather than essential, or the environment changes (faster processors, cheaper memory, new understanding of programming and languages) and that issue becomes less important or even inconsequential. If those ideas become obsolete and the language doesn't evolve, it fades from use.

The original programmers worked directly with numbers representing processor machine instructions. This approach produced numerous errors, and *assembly language* was created to replace the numbers with mnemonic *opcodes*—words that programmers could more easily remember and read, along with other helpful tools. However, there was still a one-to-one correspondence between assembly-language instructions and machine instructions, and programmers had to write each line of assembly code. In addition, each computer processor used its own distinct assembly language.

Developing programs in assembly language is exceedingly expensive. Higher-level languages help solve that problem by creating a level of abstraction away from low-level assembly languages.

Compilers and Interpreters

The instructions of an interpreted language are executed directly by a program called an *interpreter*. Kotlin is *compiled* rather than *interpreted*. The source code of a compiled language is converted into a different representation that runs as its own program, either directly on a hardware processor or on a *virtual machine* that emulates a processor:



Languages such as C, C++, Go and Rust compile into *machine code* that runs directly on the underlying hardware *central processing unit* (CPU). Languages like Java and Kotlin compile into *bytecode* which is an intermediate-level format that doesn't run directly on the hardware CPU, but instead on a *virtual machine*, which is a program that executes bytecode instructions. Programs produced by the JVM version of Kotlin run on the *Java Virtual Machine* (JVM).

Portability is an important benefit of a virtual machine. The same bytecode can run on every computer that has a virtual machine. Virtual machines can be optimized for particular hardware and to solve speed problems. The JVM contains many years of such optimizations, and has been implemented on many platforms.

At *compile time*, the code is checked by the compiler to discover *compile-time errors*. (IntelliJ IDEA and other development environments highlight these errors when you input the code, so you can quickly discover and fix any problems). If there are no compile-time errors, the source code will be compiled into bytecode.

A *runtime error* cannot be detected at compile time, so it only emerges when you run the program. Typically, runtime errors are more difficult to discover and more expensive to fix. *Statically-typed languages* like Kotlin discover as many errors as possible at compile time, while *dynamic languages* perform their safety checks at runtime (some dynamic languages don't perform as many safety checks as they might).

Languages that Influenced Kotlin

Kotlin draws its ideas and features from many languages, and those languages were influenced by earlier languages. It's helpful to know some programming-language history to gain perspective on how we got to Kotlin. The languages described here are chosen for their influence on the languages that followed them. All these languages ultimately inspired the design of Kotlin, sometimes by being an example of what *not* to do.

FORTRAN: FORmula TRANslation (1957)

Designed for use by scientists and engineers, Fortran's goal was to make it easier to encode equations. Finely-tuned and tested Fortran libraries are still in use today, but

they are typically "wrapped" to make them callable from other languages.

LISP: LISt Processor (1958)

Rather than being application-specific, LISP embodied essential programming concepts; it was the computer scientist's language and the first *functional* programming language (You'll learn about functional programming in this book). The tradeoff for its power and flexibility was efficiency: LISP was typically too expensive to run on early machines, and only in recent decades have machines become fast enough to produce a resurgence in the use of LISP. For example, the GNU Emacs editor is written entirely in LISP, and can be extended using LISP.

ALGOL: ALGOrithmic Language (1958)

Arguably the most influential of the 1950's languages because it introduced syntax that persisted in many subsequent languages. For example, C and its derivatives are "ALGOL-like" languages.

COBOL: COmmon Business-Oriented Language (1959)

Designed for business, finance, and administrative data processing. It has an English-like syntax, and was intended to be self-documenting and highly readable. Although this intent generally failed—COBOL is famous for bugs introduced by a misplaced period—the US Department of Defense forced widespread adoption on mainframe computers, and systems are still running (and requiring maintenance) today.

BASIC: Beginners' All-purpose Symbolic Instruction Code (1964)

BASIC was one of the early attempts to make programming accessible. While very successful, its features and syntax were limited, so it was only partly helpful for people who needed to learn more sophisticated languages. It is predominantly an interpreted language, which means that to run it you need the original code for the program. Despite that, many useful programs were written in BASIC, in particular as

a scripting language for Microsoft's "Office" products. BASIC might even be thought of as the first "open" programming language, as people made numerous variations of it.

Simula 67, the Original Object-Oriented Language (1967)

A *simulation* typically involves many "objects" interacting with each other. Different objects have different characteristics and behaviors. Languages that existed at the time were awkward to use for simulations, so Simula (another "ALGOL-like" language) was developed to provide direct support for creating simulation objects. It turns out that these ideas are also useful for general-purpose programming, and this was the genesis of Object-Oriented (OO) languages.

Pascal (1970)

Pascal increased compilation speed by restricting the language so it could be implemented as a *single-pass compiler*. The language forced the programmer to structure their code in a particular way and imposed somewhat awkward and less-readable constraints on program organization. As processors became faster, memory cheaper, and compiler technology better, the impact of these constraints became too costly.

An implementation of Pascal, Turbo Pascal from Borland, initially worked on CP/M machines and then made the move to early MS-DOS (precursor to Windows), later evolving into the Delphi language for Windows. By putting everything in memory, Turbo Pascal compiled at lightning speeds on very underpowered machines, dramatically improving the programming experience. Its creator, Anders Hejlsberg, later went on to design both C# and TypeScript.

Niklaus Wirth, the inventor of Pascal, created subsequent languages: Modula, Modula-2 and Oberon. As the name implies, Modula focused on dividing programs into modules, for better organization and faster compilation. Most modern languages support *separate compilation* and some form of module system.

C (1972)

Despite the increasing number of higher-level languages, programmers were still writing assembly language. This is often called *systems programming*, because it is done at the level of the operating system, but it also includes embedded programming for dedicated physical devices. This is not only arduous and expensive (Bruce began his career writing assembly language for embedded systems), but it isn't portable—assembly language can only run on the processor it is written for. C was designed to be a "high-level assembly language" that is still close enough to the hardware that you rarely need to write assembly. More importantly, a C program runs on any processor with a C compiler. C decoupled the program from the processor, which solved a huge and expensive problem. As a result, former assembly-language programmers could be vastly more productive in C. C has been so effective that recent languages (notably Go and Rust) are still attempting to usurp it for systems programming.

Smalltalk (1972)

Designed from the beginning to be purely object-oriented, Smalltalk significantly moved OO and language theory forward by being a platform for experimentation and demonstrating rapid application development. However, it was created when languages were still proprietary, and the entry price for a Smalltalk system could be in the thousands. It was interpreted, so you needed a Smalltalk environment to run programs. Open-source Smalltalk implementations did not appear until after the programming world had moved on. Smalltalk programmers have contributed great insights benefitting later OO languages like C++ and Java.

C++: A Better C with Objects (1983)

Bjarne Stroustrup created C++ because he wanted a better C and he wanted support for the object-oriented constructs he had experienced while using Simula-67. Bruce was a member of the C++ Standards Committee for its first eight years, and wrote three books on C++ including *Thinking in C++*.

Backwards-compatibility with C was a foundational principle of C++ design, so C code can be compiled in C++ with virtually no changes. This provided an easy migration path—programmers could continue to program in C, receive the benefits

of C++, and slowly experiment with C++ features while still being productive. Most criticisms of C++ can be traced to the constraint of backwards compatibility with C.

One of the problems with C was the issue of *memory management*. The programmer must first acquire memory, then run an operation using that memory, then release the memory. Forgetting to release memory is called a *memory leak* and can result in using up the available memory and crashing the process. The initial version of C++ made some innovations in this area, along with *constructors* to ensure proper initialization. Later versions of the language have made significant improvements in memory management.

Python: Friendly and Flexible (1990)

Python's designer, Guido Van Rossum, created the language based on his inspiration of "programming for everyone." His nurturing of the Python community has given it the reputation of being the friendliest and most supportive community in the programming world. Python was one of the first open-source languages, resulting in implementations on virtually every platform including embedded systems and machine learning. Its dynamism and ease-of-use makes it ideal for automating small, repetitive tasks but its features also support the creation of large, complex programs.

Python is a true "grass-roots" language; it never had a company promoting it and the attitude of its fans was to never push the language, but simply to help anyone learn it who wants to. The language continues to steadily improve, and in recent years its popularity has skyrocketed.

Python may have been the first mainstream language to combine functional and OO programming. It predated Java with automatic memory management using *garbage collection* (you typically never have to allocate or release memory yourself) and the ability to run programs on multiple platforms.

Haskell: Pure Functional Programming (1990)

Inspired by Miranda (1985), a proprietary language, Haskell was created as an open standard for pure functional programming research, although it has also been used for products. Syntax and ideas from Haskell have influenced a number of subsequent languages including Kotlin.

Java: Virtual Machines and Garbage Collection (1995)

James Gosling and his team were given the task of writing code for a TV set-top box. They decided they didn't like C++ and instead of creating the box, created the Java language. The company, Sun Microsystems, put an enormous marketing push behind the free language (still a new idea at the time) to attempt domination of the emerging Internet landscape.

This perceived time window for Internet domination put a lot of pressure on Java language design, resulting in a significant number of flaws (The book *Thinking in Java* illuminates these flaws so readers are prepared to cope with them). Brian Goetz at Oracle, the current lead developer of Java, has made remarkable and surprising improvements in Java despite the constraints he inherited. Although Java was remarkably successful, an important Kotlin design goal is to fix Java's flaws so programmers can be more productive.

Java's success came from two innovative features: a *virtual machine* and *garbage collection*. These were available in other languages—for example, LISP, Smalltalk and Python have garbage collection and UCSD Pascal ran on a virtual machine—but they were never considered practical for mainstream languages. Java changed that, and in doing so made programmers significantly more productive.

A virtual machine is an intermediate layer between the language and the hardware. The language doesn't have to generate machine code for a particular processor; it only needs to generate an intermediate language (bytecode) that runs on the virtual machine. Virtual machines require processing power and, before Java, were believed to be impractical. The *Java Virtual Machine* (JVM) gave rise to Java's slogan "write once, run everywhere." In addition, other languages can be more easily developed by targeting the JVM; examples include Groovy, a Java-like scripting language, and Clojure, a version of LISP.

Garbage collection solves the problem of forgetting to release memory, or if it's difficult to know when a piece of storage is no longer used. Projects have been significantly delayed or even cancelled because of memory leaks. Although garbage collection appears in some prior languages, it was believed to produce an unacceptable amount of overhead until Java demonstrated its practicality.

JavaScript: Java in Name Only (1995)

The original Web browser simply copied and displayed pages from a Web server. Web browsers proliferated, becoming a new programming platform that needed language support. Java wanted to be this language but was too awkward for the job. JavaScript began as LiveScript and was built into NetScape Navigator, one of the first Web browsers. Renaming it to JavaScript was a marketing ploy by NetScape, as the language has only a vague similarity to Java.

As the Web took off, JavaScript became tremendously important. However, the behavior of JavaScript was so unpredictable that Douglas Crockford wrote a book with the tongue-in-cheek title *JavaScript, the Good Parts*, where he demonstrated all the problems with the language so programmers could avoid them. Subsequent improvements by the ECMAScript committee have made JavaScript unrecognizable to an original JavaScript programmer. It is now considered a stable and mature language.

Web assembly (WASM) was derived from JavaScript to be a kind of bytecode for web browsers. It often runs much faster than JavaScript and can be generated by other languages. At this writing, the Kotlin team is working to add WASM as a target.

C#: Java for .NET (2000)

C# was designed to provide some of the important abilities of Java on the .NET (Windows) platform, while freeing designers from the constraint of following the Java language. The result included numerous improvements over Java. For example, C# developed the concept of *extension functions*, which are heavily used in Kotlin. C# also became significantly more functional than Java. Many C# features clearly influenced Kotlin design.

Scala: SCALAble (2003)

Martin Odersky created Scala to run on the Java virtual machine: To piggyback on the work done on the JVM, to interact with Java programs, and possibly with the idea that it might displace Java. As a researcher, Odersky and his team used Scala as a platform to experiment with language features, notably those not included in Java.

These experiments were illuminating and a number of them found their way into Kotlin, usually in a modified form. For example, the ability to redefine operators like + for use in special cases is called *operator overloading*. This was included in C++ but not Java. Scala added operator overloading but also allows you to invent new operators by combining any sequence of characters. This often produces confusing code. A limited form of operator overloading is included in Kotlin, but you can only overload operators that already exist.

Scala is also an object-functional hybrid, like Python but with a focus on pure functions and strict objects. This helped inspire Kotlin's choice to also be an object-functional hybrid.

Like Scala, Kotlin runs on the JVM but it interacts with Java far more easily than Scala does (see Appendix B). In addition, Kotlin targets JavaScript, the Android OS, and it generates native code for other platforms.

Atomic Kotlin evolved from the ideas and material in Atomic Scala[22].

Groovy: A Dynamic JVM Language (2007)

Dynamic languages are appealing because they are more interactive and concise than static languages. There have been numerous attempts to produce a more dynamic programming experience on the JVM, including Jython (Python) and Clojure (a dialect of Lisp). Groovy was the first to achieve wide acceptance.

At first glance, Groovy appears to be a cleaned-up version of Java, producing a more pleasant programming experience. Most Java code will run unchanged in Groovy, so Java programmers can be quickly productive, later learning the more sophisticated features that provide notable programming improvements over Java.

The Kotlin operators `?.` and `?:` that deal with the problem of emptiness first appeared in Groovy.

There are numerous Groovy features that are recognizable in Kotlin. Some of those features also appear in other languages, which probably pushed harder for them to be included in Kotlin.

[22] http://www.AtomicScala.com

Why Kotlin? (Introduced 2011, Version 1.0: 2016)

Just as C++ was initially intended to be "a better C," Kotlin was initially oriented towards being "a better Java." It has since evolved significantly beyond that goal.

Kotlin pragmatically chooses only the most successful and helpful features from other programming languages—after those features have been field-tested and proven especially valuable.

Thus, if you are coming from another language, you might recognize some features of that language in Kotlin. This is intentional: Kotlin maximizes productivity by leveraging tested concepts.

Readability

Readability is a primary goal in the design of the language. Kotlin syntax is concise—it requires no ceremony for most scenarios, but can still express complex ideas.

Tooling

Kotlin comes from JetBrains, a company that specializes in developer tooling. It has first-class tooling support, and many language features were designed with tooling in mind.

Multi-Paradigm

Kotlin supports multiple programming paradigms, which are gently introduced in this book:

- Imperative programming
- Functional programming
- Object-oriented programming

Multi-Platform

Kotlin source code can be compiled to different target platforms:

- **JVM**. The source code compiles into JVM bytecode (`.class` files), which can then be run on any Java Virtual Machine (JVM).
- **Android**. Android has its own runtime called ART[23] (the predecessor was called Dalvik). The Kotlin source code is compiled into *Dalvik Executable Format* (`.dex` files).
- **JavaScript**, to run inside a web browser.
- **Native Binaries** by generating machine code for specific platforms and CPUs.

This book focuses on the language itself, using the JVM as the only target platform. Once you know the language, you can apply Kotlin to different application and target platforms.

Two Kotlin Features

This atom does not assume you are a programmer, which makes it hard to explain most of the benefits of Kotlin over the alternatives. There are, however, two topics which are very impactful and can be explained at this early juncture: Java interoperability and the issue of indicating "no value."

Effortless Java Interoperability

To be "a better C," C++ must be backwards compatible with the syntax of C, but Kotlin does not have to be backwards compatible with the syntax of Java—it only needs to work with the JVM. This frees the Kotlin designers to create a much cleaner and more powerful syntax, without the visual noise and complication that clutters Java.

For Kotlin to be "a better Java," the experience of trying it must be pleasant and frictionless, so Kotlin enables effortless integration with existing Java projects. You can write a small piece of Kotlin functionality and slip it in amidst your existing Java

[23] https://source.android.com/devices/tech/dalvik

code. The Java code doesn't even know the Kotlin code is there—it just looks like more Java code.

Companies often investigate a new language by building a standalone program with that language. Ideally, this program is beneficial but nonessential, so if the project fails it can be terminated with minimal damage. Not every company wants to spend the kind of resources necessary for this type of experimentation. Because Kotlin seamlessly integrates into an existing Java system (and benefits from that system's tests), it becomes very cheap or even free to try Kotlin to see whether it's a good fit.

In addition, JetBrains, the company that creates Kotlin, provides IntelliJ IDEA in a "Community" (free) version, which includes support for both Java and Kotlin along with the ability to easily integrate the two. It even has a tool that takes Java code and (mostly) rewrites it to Kotlin.

Appendix B covers Java interoperability.

Representing Emptiness

An especially beneficial Kotlin feature is its solution to a challenging programming problem.

What do you do when someone hands you a dictionary and asks you to look up a word that doesn't exist? You could guarantee results by making up definitions for unknown words. A more useful approach is just to say, "There's no definition for that word." This demonstrates a significant problem in programming: How do you indicate "no value" for a piece of storage that is uninitialized, or for the result of an operation?

The *null reference* was invented in 1965 for ALGOL by Tony Hoare, who later called it "my billion-dollar mistake." One problem was that it was too simple—sometimes being told a room is empty isn't enough. You might need to know, for example, *why* it is empty. This leads to the second problem: the implementation. For efficiency's sake, it was typically just a special value that could fit in a small amount of memory, and what better than the memory already allocated for that information?

The original C language did not automatically initialize storage, which caused numerous problems. C++ improved the situation by setting newly-allocated storage to all zeroes. Thus, if a numerical value isn't initialized, it is simply a numerical zero.

This didn't seem so bad but it allowed uninitialized values to quietly slip through the cracks (newer C and C++ compilers often warn you about these). Worse, if a piece of storage was a *pointer*—used to indicate ("point to") another piece of storage—a null pointer would point at location zero in memory, which is almost certainly not what you want.

Java prevents accesses to uninitialized values by reporting such errors at runtime. Although this discovers uninitialized values, it doesn't solve the problem because the only way you can verify that your program won't crash is by running it. There are swarms of these kinds of bugs in Java code, and programmers waste huge amounts of time finding them.

Kotlin solves this problem by preventing operations that might cause null errors at compile time, *before the program can run*. This is the single-most celebrated feature by Java programmers adopting Kotlin. This one feature can minimize or eliminate Java's null errors, saving your project significant amounts of time and money.

An Abundance of Benefits

The two features we were able to explain here (without requiring more programming knowledge) make a huge difference whether or not you're a Java programmer. If Kotlin is your first language and you end up on a project that needs more programmers, it is much easier to recruit one of the many existing Java programmers into Kotlin.

Kotlin has many other benefits, which we cannot explain until you know more about programming. That's what the rest of the book is for.

- -

Languages are often selected by passion, not reason... I'm trying to make Kotlin a language that is loved for a reason.—Andrey Breslav, Kotlin Lead Language Designer.

Hello, World!

> "Hello, world!" is a program commonly used to demonstrate the basic syntax of programming languages.

We develop this program in several steps so you understand its parts.

First, let's examine an empty program that does nothing at all:

```
// HelloWorld/EmptyProgram.kt

fun main() {
  // Program code here ...
}
```

The example starts with a *comment*, which is illuminating text that is ignored by Kotlin. `//` (two forward slashes) begins a comment that goes to the end of the current line:

```
// Single-line comment
```

Kotlin ignores the `//` and everything after it until the end of the line. On the following line, it pays attention again.

The first line of each example in this book is a comment starting with the name of the the subdirectory containing the source-code file (Here, `HelloWorld`) followed by the name of the file: `EmptyProgram.kt`. The example subdirectory for each atom corresponds to the name of that atom.

keywords are reserved by the language and given special meaning. The keyword `fun` is short for *function*. A function is a collection of code that can be executed using that function's name (we spend a lot of time on functions throughout the book). The function's name follows the `fun` keyword, so in this case it's `main()` (in prose, we follow the function name with parentheses).

`main()` is actually a special name for a function; it indicates the "entry point" for a Kotlin program. A Kotlin program can have many functions with many different

names, but `main()` is the one that's automatically called when you execute the program.

The *parameter list* follows the function name and is enclosed by parentheses. Here, we don't pass anything into `main()` so the parameter list is empty.

The *function body* appears after the parameter list. It begins with an opening brace ({) and ends with a closing brace (}). The function body contains *statements* and *expressions*. A statement produces an effect, and an expression yields a result.

`EmptyProgram.kt` contains no statements or expressions in the body, just a comment.

Let's make the program display "Hello, world!" by adding a line in the `main()` body:

```
// HelloWorld/HelloWorld.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

The line that displays the greeting begins with `println()`. Like `main()`, `println()` is a function. This line *calls* the function, which executes its body. You give the function name, followed by parentheses containing one or more parameters. In this book, when referring to a function in the prose, we add parentheses after the name as a reminder that it is a function. Here, we say `println()`.

`println()` takes a single parameter, which is a `String`. You define a `String` by putting characters inside quotes.

`println()` moves the cursor to a new line after displaying its parameter, so subsequent output appears on the next line. You can use `print()` instead, which leaves the cursor on the same line.

Unlike some languages, you don't need a semicolon at the end of an expression in Kotlin. It's only necessary if you put more than one expression on a single line (this is discouraged).

For some examples in the book, we show the output at the end of the listing, inside a *multiline comment*. A multiline comment starts with a `/*` (a forward slash followed

by an asterisk) and continues—including line breaks (which we call *newlines*)—until a `*/` (an asterisk followed by a forward slash) ends the comment:

```
/* A multiline comment
Doesn't care
about newlines */
```

It's possible to add code on the same line *after* the closing `*/` of a comment, but it's confusing, so people don't usually do it.

Comments add information that isn't obvious from reading the code. If comments only repeat what the code says, they become annoying and people start ignoring them. When code changes, programmers often forget to update comments, so it's good practice to use comments judiciously, mainly for highlighting tricky aspects of your code.

Exercises and solutions can be found at www.AtomicKotlin.com.

`var` & `val`

> When an identifier holds data, you must decide whether it can be reassigned.

You create *identifiers* to refer to elements in your program. The most basic decision for a data identifier is whether it can change its contents during program execution, or if it can only be assigned once. This is controlled by two keywords:

- `var`, short for *variable*, which means you can reassign its contents.
- `val`, short for *value*, which means you can only initialize it; you cannot reassign it.

You define a `var` like this:

```
var identifier = initialization
```

The `var` keyword is followed by the identifier, an equals sign and then the initialization value. The identifier begins with a letter or an underscore, followed by letters, numbers and underscores. Upper and lower case are distinguished (so `thisvalue` and `thisValue` are different).

Here are some `var` definitions:

```
// VarAndVal/Vars.kt

fun main() {
  var whole = 11              // [1]
  var fractional = 1.4        // [2]
  var words = "Twas Brillig"  // [3]
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

In this book we mark lines with commented numbers in square brackets so we can refer to them in the text like this:

- **[1]** Create a `var` named `whole` and store `11` in it.
- **[2]** Store the "fractional number" `1.4` in the `var fractional`.
- **[3]** Store some text (a `String`) in the `var words`.

Note that `println()` can take any single value as an argument.

As the name *variable* implies, a `var` can vary. That is, you can change the data stored in a `var`. We say that a `var` is *mutable*:

```
// VarAndVal/AVarIsMutable.kt

fun main() {
  var sum = 1
  sum = sum + 2
  sum += 3
  println(sum)
}
/* Output:
6
*/
```

The assignment `sum = sum + 2` takes the current value of `sum`, adds two, and assigns the result back into `sum`.

The assignment `sum += 3` means the same as `sum = sum + 3`. The `+=` operator takes the previous value stored in `sum` and increases it by `3`, then assigns that new result back to `sum`.

Changing the value stored in a `var` is a useful way to express changes. However, when the complexity of a program increases, your code is clearer, safer and easier to understand if the values represented by your identifiers cannot change—that is, they cannot be reassigned. We specify an unchanging identifier using the `val` keyword instead of `var`. A `val` can only be assigned once, when it is created:

```
val identifier = initialization
```

The `val` keyword comes from *value*, indicating something that cannot change—it is *immutable*. Choose `val` instead of `var` whenever possible. The `Vars.kt` example at the beginning of this atom can be rewritten using `val`s:

```
// VarAndVal/Vals.kt

fun main() {
  val whole = 11
  // whole = 15 // Error   // [1]
  val fractional = 1.4
  val words = "Twas Brillig"
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

- **[1]** Once you initialize a `val`, you can't reassign it. If we try to reassign `whole` to a different number, Kotlin complains, saying "Val cannot be reassigned."

Choosing descriptive names for your identifiers makes your code easier to understand and often reduces the need for comments. In `Vals.kt`, you have no idea what `whole` represents. If your program is storing the number 11 to represent the time of day when you get coffee, it's more obvious to others if you name it `coffeetime` and easier to read if it's `coffeeTime` (following Kotlin style, we make the first letter lowercase).

- -

`vars` are useful when data must change as the program is running. This sounds like a common requirement, but turns out to be avoidable in practice. In general, your programs are easier to extend and maintain if you use `val`s. However, on rare occasions it's too complex to solve a problem using only `val`s. For that reason, Kotlin gives you the flexibility of `var`s. However, as you spend more time with `val`s you'll discover that you almost never need `var`s and that your programs are safer and more reliable without them.

Exercises and solutions can be found at www.AtomicKotlin.com.

Data Types

> Data can have different *types*.

To solve a math problem, you write an expression:

```
5.9 + 6
```

You know that adding those numbers produces another number. Kotlin knows that too. You know that one is a fractional number (`5.9`), which Kotlin calls a `Double`, and the other is a whole number (`6`), which Kotlin calls an `Int`. You know the result is a fractional number.

A *type* (also called *data type*) tells Kotlin how you intend to use that data. A type defines the set of values an expression of that type may produce. A type also defines the operations that can be performed on the data, the meaning of the data, and how values of that type can be stored.

Kotlin uses types to verify that your expressions are correct. In the above expression, Kotlin creates a new value of type `Double` to hold the result.

Kotlin tries to adapt to what you need. If you ask it to do something that violates type rules, it produces an error message. For example, try adding a `String` and a number:

```
// DataTypes/StringPlusNumber.kt

fun main() {
  println("Sally" + 5.9)
}
/* Output:
Sally5.9
*/
```

Types tell Kotlin how to use them correctly. In this case, the type rules tell Kotlin how to add a number to a `String`: by appending the two values and creating a `String` to hold the result.

Now try multiplying a `String` and a `Double` by changing the `+` in `StringPlusNumber.kt` to a `*`:

```
"Sally" * 5.9
```

Combining types this way doesn't make sense to Kotlin, so it gives you an error.

In `var` & `val`, we stored several types. Kotlin figured out the types for us, based on how we used them. This is called *type inference*.

We can be more verbose and specify the type:

```
val identifier: Type = initialization
```

You start with the `val` or `var` keyword, followed by the identifier, a colon, the type, an =, and the initialization value. So instead of saying:

```
val n = 1
var p = 1.2
```

You can say:

```
val n: Int = 1
var p: Double = 1.2
```

We've told Kotlin that `n` is an `Int` and `p` is a `Double`, rather than letting it infer the type.

Here are some of Kotlin's basic types:

```
// DataTypes/Types.kt

fun main() {
  val whole: Int = 11              // [1]
  val fractional: Double = 1.4     // [2]
  val trueOrFalse: Boolean = true  // [3]
  val words: String = "A value"    // [4]
  val character: Char = 'z'        // [5]
  val lines: String = """Triple quotes let
you have many lines
in your string"""                  // [6]
  println(whole)
  println(fractional)
  println(trueOrFalse)
  println(words)
  println(character)
```

```
  println(lines)
}
/* Output:
11
1.4
true
A value
z
Triple quotes let
you have many lines
in your string
*/
```

- **[1]** The `Int` data type is an *integer*, which means it only holds whole numbers.
- **[2]** To hold fractional numbers, use a `Double`.
- **[3]** A `Boolean` data type only holds the two special values `true` and `false`.
- **[4]** A `String` holds a sequence of characters. You assign a value using a double-quoted `String`.
- **[5]** A `Char` holds one character.
- **[6]** If you have many lines and/or special characters, surround them with triple-double-quotes (this is a *triple-quoted `String`*).

Kotlin uses type inference to determine the meaning of mixed types. When mixing `Int`s and `Double`s during addition, for example, Kotlin decides the type for the resulting value:

```
// DataTypes/Inference.kt

fun main() {
  val n = 1 + 1.2
  println(n)
}
/* Output:
2.2
*/
```

When you add an `Int` to a `Double` using type inference, Kotlin determines that the result `n` is a `Double` and ensures that it follows all the rules for `Double`s.

Kotlin's type inference is part of its strategy of doing work for the programmer. If you leave out the type declaration, Kotlin can usually infer it.

Exercises and solutions can be found at www.AtomicKotlin.com.

Functions

> A *function* is like a small program that has its own name, and can be executed (*invoked*) by calling that name from another function.

A function combines a group of activities, and is the most basic way to organize your programs and to re-use code.

You pass information into a function, and the function uses that information to calculate and produce a result. The basic form of a function is:

```
fun functionName(p1: Type1, p2: Type2, ...): ReturnType {
  lines of code
  return result
}
```

`p1` and `p2` are the *parameters*: the information you pass into the function. Each parameter has an identifier name (`p1`, `p2`) followed by a colon and the type of that parameter. The closing parenthesis of the parameter list is followed by a colon and the type of result produced by the function. The lines of code in the *function body* are enclosed in curly braces. The expression following the `return` keyword is the result the function produces when it's finished.

A parameter is how you define what is passed into a function—it's the placeholder. An argument is the actual value that you pass into the function.

The combination of name, parameters and return type is called the *function signature*.

Here's a simple function called `multiplyByTwo()`:

```
// Functions/MultiplyByTwo.kt

fun multiplyByTwo(x: Int): Int {  // [1]
  println("Inside multiplyByTwo") // [2]
  return x * 2
}

fun main() {
  val r = multiplyByTwo(5)        // [3]
  println(r)
}
/* Output:
Inside multiplyByTwo
10
*/
```

- **[1]** Notice the `fun` keyword, the function name, and the parameter list consisting of a single parameter. This function takes an `Int` parameter and returns an `Int`.
- **[2]** These two lines are the body of the function. The final line returns the value of its calculation `x * 2` as the result of the function.
- **[3]** This line *calls* the function with an appropriate argument, and captures the result into `val r`. A function call mimics the form of its declaration: the function name, followed by arguments inside parentheses.

The function code is executed by calling the function, using the function name `multiplyByTwo()` as an abbreviation for that code. This is why functions are the most basic form of simplification and code reuse in programming. You can also think of a function as an expression with substitutable values (the parameters).

`println()` is also a function call—it just happens to be provided by Kotlin. We refer to functions defined by Kotlin as *library functions*.

If the function doesn't provide a meaningful result, its return type is `Unit`. You can specify `Unit` explicitly if you want, but Kotlin lets you omit it:

```
// Functions/SayHello.kt

fun sayHello() {
  println("Hallo!")
}

fun sayGoodbye(): Unit {
  println("Auf Wiedersehen!")
}

fun main() {
  sayHello()
  sayGoodbye()
}
/* Output:
Hallo!
Auf Wiedersehen!
*/
```

Both `sayHello()` and `sayGoodbye()` return `Unit`, but `sayHello()` leaves out the explicit declaration. The `main()` function also returns `Unit`.

If a function is only a single expression, you can use the abbreviated syntax of an equals sign followed by the expression:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = expression
```

A function body surrounded by curly braces is called a *block body*. A function body using the equals syntax is called an *expression body*.

Here, `multiplyByThree()` uses an expression body:

```
// Functions/MultiplyByThree.kt

fun multiplyByThree(x: Int): Int = x * 3

fun main() {
  println(multiplyByThree(5))
}
/* Output:
15
*/
```

This is a short version of saying `return x * 3` inside a block body.

Kotlin infers the return type of a function that has an expression body:

```
// Functions/MultiplyByFour.kt

fun multiplyByFour(x: Int) = x * 4

fun main() {
  val result: Int = multiplyByFour(5)
  println(result)
}
/* Output:
20
*/
```

Kotlin infers that `multiplyByFour()` returns an `Int`.

Kotlin can *only* infer return types for expression bodies. If a function has a block body and you omit its type, that function returns `Unit`.

- -

When writing functions, choose descriptive names. This makes the code easier to read, and can often reduce the need for code comments. We can't always be as descriptive as we would prefer with the function names in this book because we're constrained by line widths.

Exercises and solutions can be found at www.AtomicKotlin.com.

`if` Expressions

> An *`if` expression* makes a choice.

The `if` keyword tests an expression to see whether it's `true` or `false` and performs an action based on the result. A true-or-false expression is called a *Boolean*, after the mathematician George Boole who invented the logic behind these expressions. Here's an example using the > (greater than) and < (less than) symbols:

```
// IfExpressions/If1.kt

fun main() {
  if (1 > 0)
    println("It's true!")
  if (10 < 11) {
    println("10 < 11")
    println("ten is less than eleven")
  }
}
/* Output:
It's true!
10 < 11
ten is less than eleven
*/
```

The expression inside the parentheses after the `if` must evaluate to `true` or `false`. If `true`, the following expression is executed. To execute multiple lines, place them within curly braces.

We can create a Boolean expression in one place, and use it in another:

```
// IfExpressions/If2.kt

fun main() {
  val x: Boolean = 1 >= 1
  if (x)
    println("It's true!")
}
/* Output:
It's true!
*/
```

Because `x` is `Boolean`, the `if` can test it directly by saying `if(x)`.

The Boolean `>=` operator returns `true` if the expression on the left side of the operator is *greater than or equal* to that on the right. Likewise, `<=` returns `true` if the expression on the left side is *less than or equal* to that on the right.

The `else` keyword allows you to handle both `true` and `false` paths:

```
// IfExpressions/If3.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else
    println("It's negative or zero")
}
/* Output:
It's negative or zero
*/
```

The `else` keyword is only used in conjunction with `if`. You are not limited to a single check—you can test multiple combinations by combining `else` and `if`:

```
// IfExpressions/If4.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else if (n == 0)
    println("It's zero")
  else
    println("It's negative")
}
/* Output:
It's negative
*/
```

Here we use == to check two numbers for equality. != tests for inequality.

The typical pattern is to start with `if`, followed by as many `else if` clauses as you need, ending with a final `else` for anything that doesn't match all the previous tests. When an `if` expression reaches a certain size and complexity you'll probably use a `when` expression instead. `when` is described later in the book, in when Expressions.

The "not" operator ! tests for the opposite of a Boolean expression:

```
// IfExpressions/If5.kt

fun main() {
  val y: Boolean = false
  if (!y)
    println("!y is true")
}
/* Output:
!y is true
*/
```

To verbalize `if(!y)`, say "if not y."

The entire `if` is an expression, so it can produce a result:

```
// IfExpressions/If6.kt

fun main() {
  val num = 10
  val result = if (num > 100) 4 else 42
  println(result)
}
/* Output:
42
*/
```

Here, we store the value produced by the entire `if` expression in an intermediate identifier called `result`. If the condition is satisfied, the first branch produces `result`. If not, the `else` value becomes `result`.

Let's practice creating functions. Here's one that takes a Boolean parameter:

```
// IfExpressions/TrueOrFalse.kt

fun trueOrFalse(exp: Boolean): String {
  if (exp)
    return "It's true!"          // [1]
  return "It's false"            // [2]
}

fun main() {
  val b = 1
  println(trueOrFalse(b < 3))
  println(trueOrFalse(b >= 3))
}
/* Output:
It's true!
It's false
*/
```

The Boolean parameter `exp` is passed to the function `trueOrFalse()`. If the argument is passed as an expression, such as `b < 3`, that expression is first evaluated and the result is passed to the function. `trueOrFalse()` tests `exp` and if the result is `true`, line **[1]** is executed, otherwise line **[2]** is executed.

- **[1]** `return` says, "Leave the function and produce this value as the function's result." Notice that `return` can appear anywhere in a function and does not have to be at the end.

Rather than using `return` as in the previous example, you can use the `else` keyword to produce the result as an expression:

```
// IfExpressions/OneOrTheOther.kt

fun oneOrTheOther(exp: Boolean): String =
  if (exp)
    "True!" // No 'return' necessary
  else
    "False"

fun main() {
  val x = 1
  println(oneOrTheOther(x == 1))
  println(oneOrTheOther(x == 2))
}
/* Output:
True!
False
*/
```

Instead of two expressions in `trueOrFalse()`, `oneOrTheOther()` is a single expression. The result of that expression is the result of the function, so the `if` expression becomes the function body.

Exercises and solutions can be found at www.AtomicKotlin.com.

String Templates

> A *`String` template* is a programmatic way to generate a `String`.

If you put a `$` before an identifier name, the `String` template will insert that identifier's contents into the `String`:

```
// StringTemplates/StringTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")     // [1]
  println("printing a $1")      // [2]
}
/* Output:
Found 42!
printing a $1
*/
```

- **[1]** `$answer` substitutes the value of `answer`.
- **[2]** If what follows the `$` isn't recognizable as a program identifier, nothing special happens.

You can also insert values into a `String` using concatenation (+):

```
// StringTemplates/StringConcatenation.kt

fun main() {
  val s = "hi\n" // \n is a newline character
  val n = 11
  val d = 3.14
  println("first: " + s + "second: " +
    n + ", third: " + d)
}
/* Output:
first: hi
second: 11, third: 3.14
*/
```

Placing an expression inside `${}` evaluates it. The return value is converted to a `String` and inserted into the resulting `String`:

```
// StringTemplates/ExpressionInTemplate.kt

fun main() {
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [1]
  val x = 11
  println("$x + 4 = ${x + 4}")
}
/* Output:
a
11 + 4 = 15
*/
```

- **[1]** `if(condition) 'a' else 'b'` is evaluated and the result is substituted for the entire `${}` expression.

When a `String` must include a special character, such as a quote, you can either escape that character with a `\` (*backslash*), or use a `String` literal in triple quotes:

```
// StringTemplates/TripleQuotes.kt

fun main() {
  val s = "value"
  println("s = \"$s\".")
  println("""s = "$s".""")
}
/* Output:
s = "value".
s = "value".
*/
```

With triple quotes, you insert a value of an expression the same way you do it for a single-quoted `String`.

Exercises and solutions can be found at www.AtomicKotlin.com.

Number Types

Different types of numbers are stored in different ways.

If you create an identifier and assign an integer value to it, Kotlin infers the `Int` type:

```
// NumberTypes/InferInt.kt

fun main() {
  val million = 1_000_000  // Infers Int
  println(million)
}
/* Output:
1000000
*/
```

For readability, Kotlin allows underscores within numerical values.

The basic mathematical operators for numbers are the ones available in most programming languages: addition (+), subtraction (-), division (/), multiplication (*) and modulus (%), which produces the remainder from integer division:

```
// NumberTypes/Modulus.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator % denominator)
}
/* Output:
9
*/
```

Integer division truncates its result:

```
// NumberTypes/IntDivisionTruncates.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator / denominator)
}
/* Output:
1
*/
```

If the operation had rounded the result, the output would be `2`.

The order of operations follows basic arithmetic:

```
// NumberTypes/OpOrder.kt

fun main() {
  println(45 + 5 * 6)
}
/* Output:
75
*/
```

The multiplication operation `5 * 6` is performed first, followed by the addition `45 + 30`.

If you want `45 + 5` to happen first, use parentheses:

```
// NumberTypes/OpOrderParens.kt

fun main() {
  println((45 + 5) * 6)
}
/* Output:
300
*/
```

Now let's calculate *body mass index* (BMI), which is weight in kilograms divided by the square of the height in meters. If you have a BMI of less than `18.5`, you are underweight. Between `18.5` and `24.9` is normal weight. BMI of `25` and higher is overweight. This example also shows the preferred formatting style when you can't fit the function's parameters on a single line:

```
// NumberTypes/BMIMetric.kt

fun bmiMetric(
  weight: Double,
  height: Double
): String {
  val bmi = weight / (height * height)  // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}

fun main() {
  val weight = 72.57 // 160 lbs
  val height = 1.727 // 68 inches
  val status = bmiMetric(weight, height)
  println(status)
}
/* Output:
Normal weight
*/
```

- **[1]** If you remove the parentheses, you divide `weight` by `height` then multiply that result by `height`. That's a much larger number, and the wrong answer.

`bmiMetric()` uses `Doubles` for the weight and height. A `Double` holds very large and very small floating-point numbers.

Here's a version using English units, represented by `Int` parameters:

```
// NumberTypes/BMIEnglish.kt

fun bmiEnglish(
  weight: Int,
  height: Int
): String {
  val bmi =
    weight / (height * height) * 703.07 // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}
```

```
fun main() {
  val weight = 160
  val height = 68
  val status = bmiEnglish(weight, height)
  println(status)
}
/* Output:
Underweight
*/
```

Why does the result differ from `bmiMetric()`, which uses `Doubles`? When you divide an integer by another integer, Kotlin produces an integer result. The standard way to deal with the remainder during integer division is *truncation*, meaning "chop it off and throw it away" (there's no rounding). So if you divide `5` by `2` you get `2`, and `7/10` is zero. When Kotlin calculates `bmi` in expression **[1]**, it divides `160` by `68 * 68` and gets zero. It then multiplies zero by `703.07` to get zero.

To avoid this problem, move `703.07` to the front of the calculation. The calculations are then forced to be `Double`:

```
val bmi = 703.07 * weight / (height * height)
```

The `Double` parameters in `bmiMetric()` prevent this problem. Convert computations to the desired type as early as possible to preserve accuracy.

All programming languages have limits to what they can store within an integer. Kotlin's `Int` type can take values between -2^{31} and $+2^{31}-1$, a constraint of the `Int` 32-bit representation. If you sum or multiply two `Int`s that are big enough, you'll overflow the result:

```
// NumberTypes/IntegerOverflow.kt

fun main() {
  val i: Int = Int.MAX_VALUE
  println(i + i)
}
/* Output:
-2
*/
```

`Int.MAX_VALUE` is a predefined value which is the largest number an `Int` can hold.

The overflow produces a result that is clearly incorrect, as it is both negative and much smaller than we expect. Kotlin issues a warning whenever it detects a potential overflow.

Preventing overflow is your responsibility as a developer. Kotlin can't always detect overflow during compilation, and it doesn't prevent overflow because that would produce an unacceptable performance impact.

If your program contains large numbers, you can use `Longs`, which accommodate values from -2^{63} to $+2^{63}-1$. To define a `val` of type `Long`, you can specify the type explicitly or put `L` at the end of a numeric literal, which tells Kotlin to treat that value as a `Long`:

```
// NumberTypes/LongConstants.kt

fun main() {
  val i = 0           // Infers Int
  val l1 = 0L         // L creates Long
  val l2: Long = 0    // Explicit type
  println("$l1 $l2")
}
/* Output:
0 0
*/
```

By changing to `Longs` we prevent the overflow in `IntegerOverflow.kt`:

```
// NumberTypes/UsingLongs.kt

fun main() {
  val i = Int.MAX_VALUE
  println(0L + i + i)               // [1]
  println(1_000_000 * 1_000_000L)  // [2]
}
/* Output:
4294967294
1000000000000
*/
```

Using a numeric literal in both **[1]** and **[2]** forces `Long` calculations, and also produces a result of type `Long`. The location where the `L` appears is unimportant. If one of the values is `Long`, the resulting expression is `Long`.

Although they can hold much larger values than `Int`s, `Long`s still have size limitations:

```
// NumberTypes/BiggestLong.kt

fun main() {
  println(Long.MAX_VALUE)
}
/* Output:
9223372036854775807
*/
```

`Long.MAX_VALUE` is the largest value a `Long` can hold.

Exercises and solutions can be found at www.AtomicKotlin.com.

Booleans

> `if` Expressions demonstrated the "not" operator `!`, which negates a `Boolean` value. This atom introduces more *Boolean Algebra*.

We start with the operators "and" and "or":

- `&&` (and): Produces `true` only if the `Boolean` expression on the left of the operator and the one on the right are both `true`.
- `||` (or): Produces `true` if either the expression on the left or right of the operator is `true`, or if both are `true`.

In this example, we determine whether a business is open or closed, based on the `hour`:

```
// Booleans/Open1.kt

fun isOpen1(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status =
    if (hour >= open && hour < closed) // [1]
      true
    else
      false
  println("Open: $status")
}

fun main() = isOpen1(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

`main()` is a single function call, so we can use an expression body as described in Functions.

The `if` expression in **[1]** Checks whether `hour` is between the opening time and closing time, so we combine the expressions with the `Boolean` `&&` (and).

The `if` expression can be simplified. The result of the expression `if(cond) true else false` is just `cond`:

```
// Booleans/Open2.kt

fun isOpen2(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour >= open && hour < closed
  println("Open: $status")
}

fun main() = isOpen2(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

Let's reverse the logic and check whether the business is currently closed. The "or" operator `||` produces `true` if at least one of the conditions is satisfied:

```
// Booleans/Closed.kt

fun isClosed(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour < open || hour >= closed
  println("Closed: $status")
}

fun main() = isClosed(6)
/* Output:
Operating hours: 9 - 20
Closed: true
*/
```

`Boolean` operators enable complicated logic in compact expressions. However, things can easily become confusing. Strive for readability and specify your intentions explicitly.

Here's an example of a complicated `Boolean` expression where different evaluation order produces different results:

```
// Booleans/EvaluationOrder.kt

fun main() {
  val sunny = true
  val hoursSleep = 6
  val exercise = false
  val temp = 55

  // [1]:
  val happy1 = sunny && temp > 50 ||
    exercise && hoursSleep > 7
  println(happy1)

  // [2]:
  val sameHappy1 = (sunny && temp > 50) ||
    (exercise && hoursSleep > 7)
  println(sameHappy1)

  // [3]:
  val notSame =
    (sunny && temp > 50 || exercise) &&
      hoursSleep > 7
  println(notSame)
}
/* Output:
true
true
false
*/
```

The `Boolean` expressions are `sunny`, `temp > 50`, `exercise`, and `hoursSleep > 7`. We read `happy1` as "It's sunny *and* the temperature is greater than 50 *or* I've exercised *and* had more than 7 hours of sleep." But does `&&` have precedence over `||`, or the opposite?

The expression in **[1]** uses Kotlin's default evaluation order. This produces the same result as the expression in **[2]** because, without parentheses, the "ands" are evaluated first, then the "or". The expression in **[3]** uses parentheses to produce a different result. In **[3]**, we're only happy if we get at least 7 hours of sleep.

Exercises and solutions can be found at www.AtomicKotlin.com.

Repetition with `while`

> Computers are ideal for repetitive tasks.

The most basic form of repetition uses the `while` keyword. This repeats a block as long as the controlling *Boolean expression* is `true`:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

The Boolean expression is evaluated once at the beginning of the loop and again before each further iteration through the block.

```
// RepetitionWithWhile/WhileLoop.kt

fun condition(i: Int) = i < 100  // [1]

fun main() {
  var i = 0
  while (condition(i)) {         // [2]
    print(".")
    i += 10                      // [3]
  }
}
/* Output:
..........
*/
```

- **[1]** The comparison operator `<` produces a `Boolean` result, so Kotlin infers `Boolean` as the result type for `condition()`.
- **[2]** The conditional expression for the `while` says: "repeat the statements in the body as long as `condition()` returns `true`."
- **[3]** The `+=` operator adds `10` to `i` and assigns the result to `i` in a single operation (`i` must be a `var` for this to work). This is equivalent to:

```
i = i + 10
```

There's a second way to use `while`, in conjunction with the `do` keyword:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

Rewriting `WhileLoop.kt` to use a `do-while` produces:

```
// RepetitionWithWhile/DoWhileLoop.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (condition(i))
}
/* Output:
..........
*/
```

The sole difference between `while` and `do-while` is that the body of the `do-while` always executes at least once, even if the Boolean expression initially produces `false`. In a `while`, if the conditional is `false` the first time, then the body never executes. In practice, `do-while` is less common than `while`.

The short versions of assignment operators are available for all the arithmetic operations: `+=`, `-=`, `*=`, `/=`, and `%=`. This uses `-=` and `%=`:

```
// RepetitionWithWhile/AssignmentOperators.kt

fun main() {
  var n = 10
  val d = 3
  print(n)
  while (n > d) {
    n -= d
    print(" - $d")
  }
  println(" = $n")

  var m = 10
  print(m)
  m %= d
  println(" % $d = $m")
}
/* Output:
10 - 3 - 3 - 3 = 1
10 % 3 = 1
*/
```

To calculate the remainder of the integer division of two natural numbers, we start with a `while` loop, then use the remainder operator.

Adding `1` and subtracting `1` from a number are so common that they have their own increment and decrement operators: `++` and `--`. You can replace `i += 1` with `i++`:

```
// RepetitionWithWhile/IncrementOperator.kt

fun main() {
  var i = 0
  while (i < 4) {
    print(".")
    i++
  }
}
/* Output:
....
*/
```

In practice, `while` loops are not used for iterating over a range of numbers. The `for` loop is used instead. This is covered in the next atom.

Exercises and solutions can be found at www.AtomicKotlin.com.

Looping & Ranges

The `for` keyword executes a block of code for each value in a sequence.

The set of values can be a range of integers, a `String`, or, as you'll see later in the book, a collection of items. The `in` keyword indicates that you are stepping through values:

```
for (v in values) {
  // Do something with v
}
```

Each time through the loop, `v` is given the next element in `values`.

Here's a `for` loop repeating an action a fixed number of times:

```
// LoopingAndRanges/RepeatThreeTimes.kt

fun main() {
  for (i in 1..3) {
    println("Hey $i!")
  }
}
/* Output:
Hey 1!
Hey 2!
Hey 3!
*/
```

The output shows the index `i` receiving each value in the range from `1` to `3`.

A *range* is an interval of values defined by a pair of endpoints. There are two basic ways to define ranges:

```
// LoopingAndRanges/DefiningRanges.kt

fun main() {
  val range1 = 1..10          // [1]
  val range2 = 0 until 10     // [2]
  println(range1)
  println(range2)
}
/* Output:
1..10
0..9
*/
```

- **[1]** Using `..` syntax includes both bounds in the resulting range.
- **[2]** `until` excludes the end. The output shows that `10` is not part of the range.

Displaying a range produces a readable format.

This sums the numbers from 10 to 100:

```
// LoopingAndRanges/SumUsingRange.kt

fun main() {
  var sum = 0
  for (n in 10..100) {
    sum += n
  }
  println("sum = $sum")
}
/* Output:
sum = 5005
*/
```

You can iterate over a range in reverse order. You can also use a `step` value to change the interval from the default of `1`:

```
// LoopingAndRanges/ForWithRanges.kt

fun showRange(r: IntProgression) {
  for (i in r) {
    print("$i ")
  }
  print("    // $r")
  println()
}

fun main() {
  showRange(1..5)
  showRange(0 until 5)
  showRange(5 downTo 1)          // [1]
  showRange(0..9 step 2)         // [2]
  showRange(0 until 10 step 3)   // [3]
  showRange(9 downTo 2 step 3)
}
/* Output:
1 2 3 4 5     // 1..5
0 1 2 3 4     // 0..4
5 4 3 2 1     // 5 downTo 1 step 1
0 2 4 6 8     // 0..8 step 2
0 3 6 9     // 0..9 step 3
9 6 3     // 9 downTo 3 step 3
*/
```

- **[1]** `downTo` produces a decreasing range.
- **[2]** `step` changes the interval. Here, the range steps by a value of two instead of one.
- **[3]** `until` can also be used with `step`. Notice how this affects the output.

In each case the sequence of numbers form an arithmetic progression. `showRange()` accepts an `IntProgression` parameter, which is a built-in type that includes `Int` ranges. Notice that the `String` representation of each `IntProgression` as it appears in output comment for each line is often different from the range passed into `showRange()`—the `IntProgression` is translating the input into an equivalent common form.

You can also produce a range of characters. This `for` iterates from `a` to `z`:

```
// LoopingAndRanges/ForWithCharRange.kt

fun main() {
  for (c in 'a'..'z') {
    print(c)
  }
}
/* Output:
abcdefghijklmnopqrstuvwxyz
*/
```

You can iterate over a range of elements that are whole quantities, like integers and characters, but not floating-point values.

Square brackets access characters by index. Because we start counting characters in a `String` at zero, `s[0]` selects the first character of the `String` `s`. Selecting `s.lastIndex` produces the final index number:

```
// LoopingAndRanges/IndexIntoString.kt

fun main() {
  val s = "abc"
  for (i in 0..s.lastIndex) {
    print(s[i] + 1)
  }
}
/* Output:
bcd
*/
```

Sometimes people describe `s[0]` as "the zeroth character."

Characters are stored as numbers corresponding to their Unicode[24] values, so adding an integer to a character produces a new character corresponding to the new code value:

[24] https://en.wikipedia.org/wiki/Unicode

```
// LoopingAndRanges/AddingIntToChar.kt

fun main() {
  val ch: Char = 'a'
  println(ch + 25)
  println(ch < 'z')
}
/* Output:
z
true
*/
```

The second `println()` shows that you can compare character codes.

A `for` loop can iterate over `String`s directly:

```
// LoopingAndRanges/IterateOverString.kt

fun main() {
  for (ch in "Jnskhm ") {
    print(ch + 1)
  }
}
/* Output:
Kotlin!
*/
```

`ch` receives each character in turn.

In the following example, the function `hasChar()` iterates over the `String` `s` and tests whether it contains a given character `ch`. The `return` in the middle of the function stops the function when the answer is found:

```
// LoopingAndRanges/HasChar.kt

fun hasChar(s: String, ch: Char): Boolean {
  for (c in s) {
    if (c == ch) return true
  }
  return false
}

fun main() {
  println(hasChar("kotlin", 't'))
  println(hasChar("kotlin", 'a'))
}
/* Output:
true
false
*/
```

The next atom shows that `hasChar()` is unnecessary—you can use built-in syntax instead.

If you simply want to repeat an action a fixed number of times, you may use `repeat()` instead of a `for` loop:

```
// LoopingAndRanges/RepeatHi.kt

fun main() {
  repeat(2) {
    println("hi!")
  }
}
/* Output:
hi!
hi!
*/
```

`repeat()` is a standard library function, not a keyword. You'll see how it was created much later in the book.

Exercises and solutions can be found at www.AtomicKotlin.com.

The `in` Keyword

The `in` keyword tests whether a value is within a range.

```
// InKeyword/MembershipInRange.kt

fun main() {
  val percent = 35
  println(percent in 1..100)
}
/* Output:
true
*/
```

In Booleans, you learned to check bounds explicitly:

```
// InKeyword/MembershipUsingBounds.kt

fun main() {
  val percent = 35
  println(0 <= percent && percent <= 100)
}
/* Output:
true
*/
```

`0 <= x && x <= 100` is logically equivalent to `x in 0..100`. IntelliJ IDEA suggests automatically replacing the first form with the second, which is easier to read and understand.

The `in` keyword is used for both iteration and membership. An `in` inside the control expression of a `for` loop means iteration, otherwise `in` checks membership:

```
// InKeyword/IterationVsMembership.kt

fun main() {
  val values = 1..3
  for (v in values) {
    println("iteration $v")
  }
  val v = 2
  if (v in values)
    println("$v is a member of $values")
}
/* Output:
iteration 1
iteration 2
iteration 3
2 is a member of 1..3
*/
```

The `in` keyword is not limited to ranges. You can also check whether a character is a part of a `String`. The following example uses `in` instead of `hasChar()` from the previous atom:

```
// InKeyword/InString.kt

fun main() {
  println('t' in "kotlin")
  println('a' in "kotlin")
}
/* Output:
true
false
*/
```

Later in the book you'll see that `in` works with other types, as well.

Here, `in` tests whether a character belongs to a range of characters:

```
// InKeyword/CharRange.kt

fun isDigit(ch: Char) = ch in '0'..'9'

fun notDigit(ch: Char) =
  ch !in '0'..'9'               // [1]

fun main() {
  println(isDigit('a'))
  println(isDigit('5'))
  println(notDigit('z'))
}
/* Output:
false
true
true
*/
```

- **[1]** `!in` checks that a value doesn't belong to a range.

You can create a `Double` range, but you can only use it to check for membership:

```
// InKeyword/FloatingPointRange.kt

fun inFloatRange(n: Double) {
  val r = 1.0..10.0
  println("$n in $r? ${n in r}")
}

fun main() {
  inFloatRange(0.999999)
  inFloatRange(5.0)
  inFloatRange(10.0)
  inFloatRange(10.0000001)
}
/* Output:
0.999999 in 1.0..10.0? false
5.0 in 1.0..10.0? true
10.0 in 1.0..10.0? true
10.0000001 in 1.0..10.0? false
*/
```

You can only use `..` to define a floating-point range in Kotlin.

You can check whether a `String` is a member of a range of `String`s:

```
// InKeyword/StringRange.kt

fun main() {
  println("ab" in "aa".."az")
  println("ba" in "aa".."az")
}
/* Output:
true
false
*/
```

Here Kotlin uses alphabetic comparison.

Exercises and solutions can be found at www.AtomicKotlin.com.

Expressions & Statements

> *Statements* and *expressions* are the smallest useful fragments of code in most programming languages.

There's a basic difference: a statement has an effect, but produces no result. An expression always produces a result.

Because it doesn't produce a result, a statement must change the state of its surroundings to be useful. Another way to say this is "a statement is called for its *side effects*" (that is, what it does *other* than producing a result). As a memory aid:

> *A statement changes state.*

One definition of "express" is "to force or squeeze out," as in "to express the juice from an orange." So

> *An expression expresses.*

That is, it produces a result.

The `for` loop is a statement in Kotlin. You cannot assign it because there's no result:

```
// ExpressionsStatements/ForIsAStatement.kt

fun main() {
  // Can't do this:
  // val f = for(i in 1..10) {}
  // Compiler error message:
  // for is not an expression, and
  // only expressions are allowed here
}
```

A `for` loop is used for its side effects.

An expression produces a value, which can be assigned or used as part of another expression, whereas a statement is always a top-level element.

Every function call is an expression. Even if the function returns `Unit` and is called only for its side effects, the result can still be assigned:

```
// ExpressionsStatements/UnitReturnType.kt

fun unitFun() = Unit

fun main() {
  println(unitFun())
  val u1: Unit = println(42)
  println(u1)
  val u2 = println(0) // Type inference
  println(u2)
}
/* Output:
kotlin.Unit
42
kotlin.Unit
0
kotlin.Unit
*/
```

The `Unit` type contains a single value called `Unit`, which you can return directly, as seen in `unitFun()`. Calling `println()` also returns `Unit`. The `val u1` captures the return value of `println()` and is explicitly declared as `Unit` while `u2` uses type inference.

`if` creates an expression, so you can assign its result:

```
// ExpressionsStatements/AssigningAnIf.kt

fun main() {
  val result1 = if (11 > 42) 9 else 5

  val result2 = if (1 < 2) {
    val a = 11
    a + 42
  } else 42

  val result3 =
    if ('x' < 'y')
      println("x < y")
    else
      println("x > y")

  println(result1)
  println(result2)
  println(result3)
}
/* Output:
x < y
5
53
kotlin.Unit
*/
```

The first output line is `x < y`, even though `result3` isn't displayed until the end of `main()`. This happens because evaluating `result3` calls `println()`, and the evaluation occurs when `result3` is defined.

Notice that `a` is defined inside the block of code for `result2`. The result of the last expression becomes the result of the `if` expression; here, it's the sum of 11 and 42. But what about `a`? Once you leave the code block (move outside the curly braces), you can't access `a`. It is *temporary* and is discarded once you exit the *scope* of that block.

The increment operator `i++` is also an expression, even if it looks like a statement. Kotlin follows the approach used by C-like languages and provides two versions of increment and decrement operators with slightly different semantics. The prefix operator appears before the operand, as in `++i`, and returns the value after the

increment happens. You can read it as "first do the increment, then return the resulting value." The postfix operator is placed after the operand, as in `i++`, and returns the value of `i` before the increment occurs. You can read it as "first produce the result, then do the increment."

```
// ExpressionsStatements/PostfixVsPrefix.kt

fun main() {
  var i = 10
  println(i++)
  println(i)
  var j = 20
  println(++j)
  println(j)
}
/* Output:
10
11
21
21
*/
```

The decrement operator also has two versions: `--i` and `i--`. Using increment and decrement operators within other expressions is discouraged because it can produce confusing code:

```
// ExpressionsStatements/Confusing.kt

fun main() {
  var i = 1
  println(i++ + ++i)
}
```

Try to guess what the output will be, then check it.

Exercises and solutions can be found at www.AtomicKotlin.com.

Summary 1

> This atom summarizes and reviews the atoms in Section I, starting at Hello, World! and ending with Expressions & Statements.

If you're an experienced programmer, this should be your first atom. New programmers should read this atom and perform the exercises as a review of Section I.

If anything isn't clear to you, study the associated atom for that topic (the subheadings correspond to atom titles).

Hello, World

Kotlin supports both `//` single-line comments, and `/*`-to-`*/` multiline comments. A program's entry point is the function `main()`:

```
// Summary1/Hello.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

The first line of each example in this book is a comment containing the name of the atom's subdirectory, followed by a `/` and the name of the file. You can find all the extracted code examples through AtomicKotlin.com[25].

`println()` is a standard library function which takes a single `String` parameter (or a parameter that can be converted to a `String`). `println()` moves the cursor to a new line after displaying its parameter, while `print()` leaves the cursor on the same line.

[25] http://AtomicKotlin.com

Kotlin does not require a semicolon at the end of an expression or statement. Semicolons are only necessary to separate multiple expressions or statements on a single line.

`var` & `val`, Data Types

To create an unchanging identifier, use the `val` keyword followed by the identifier name, a colon, and the type for that value. Then add an equals sign and the value to assign to that `val`:

```
val identifier: Type = initialization
```

Once a `val` is assigned, it cannot be reassigned.

Kotlin's *type inference* can usually determine the type automatically, based on the initialization value. This produces a simpler definition:

```
val identifier = initialization
```

Both of the following are valid:

```
val daysInFebruary = 28
val daysInMarch: Int = 31
```

A `var` (variable) definition looks the same, using `var` instead of `val`:

```
var identifier1 = initialization
var identifier2: Type = initialization
```

Unlike a `val`, you can modify a `var`, so the following is legal:

```
var hoursSpent = 20
hoursSpent = 25
```

However, the *type* can't be changed, so you get an error if you say:

```
hoursSpent = 30.5
```

Kotlin infers the `Int` type when `hoursSpent` is defined, so it won't accept the change to a floating-point value.

Functions

Functions are *named subroutines*:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType {
  // Lines of code ...
  return result
}
```

The `fun` keyword is followed by the function name and the parameter list in parentheses. Each parameter must have an explicit type because Kotlin cannot infer parameter types. The function itself has a type, defined in the same way as for a `var` or `val` (a colon followed by the type). A function's type is the type of the returned result.

The function signature is followed by the function body contained within curly braces. The `return` statement provides the function's return value.

You can use an abbreviated syntax when the function consists of a single expression:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = result
```

This form is called an *expression body*. Instead of an opening curly brace, use an equals sign followed by the expression. You can omit the return type because Kotlin infers it.

Here's a function that produces the cube of its parameter, and another that adds an exclamation point to a `String`:

```
// Summary1/BasicFunctions.kt

fun cube(x: Int): Int {
  return x * x * x
}

fun bang(s: String) = s + "!"

fun main() {
  println(cube(3))
  println(bang("pop"))
}
/* Output:
27
pop!
*/
```

`cube()` has a block body with an explicit `return` statement. `bang()` is an expression body producing the function's return value. Kotlin infers `bang()`'s return type to be `String`.

Booleans

For Boolean algebra, Kotlin provides operators such as:

- `!` (not) logically negates the value (turns `true` to `false` and vice-versa).
- `&&` (and) returns `true` only if *both* conditions are `true`.
- `||` (or) returns `true` if at least one of the conditions is `true`.

```
// Summary1/Booleans.kt

fun main() {
  val opens = 9
  val closes = 20
  println("Operating hours: $opens - $closes")
  val hour = 6
  println("Current time: " + hour)

  val isOpen = hour >= opens && hour < closes
  println("Open: " + isOpen)
  println("Not open: " + !isOpen)

  val isClosed = hour < opens || hour >= closes
  println("Closed: " + isClosed)
}
/* Output:
Operating hours: 9 - 20
Current time: 6
Open: false
Not open: true
Closed: true
*/
```

`isOpen`'s initializer uses `&&` to test whether both conditions are `true`. The first condition `hour >= opens` is `false`, so the result of the entire expression becomes `false`. The initializer for `isClosed` uses `||`, producing `true` if at least one of the conditions is `true`. The expression `hour < opens` is `true`, so the whole expression is `true`.

`if` Expressions

Because `if` is an expression, it produces a result. This result can be assigned to a `var` or `val`. Here, you also see the use of the `else` keyword:

```
// Summary1/IfResult.kt

fun main() {
  val result = if (99 < 100) 4 else 42
  println(result)
}
/* Output:
4
*/
```

Either branch of an `if` expression can be a multiline block of code surrounded by curly braces:

```
// Summary1/IfExpression.kt

fun main() {
  val activity = "swimming"
  val hour = 10

  val isOpen = if (
    activity == "swimming" ||
    activity == "ice skating") {
    val opens = 9
    val closes = 20
    println("Operating hours: " +
      opens + " - " + closes)
    hour >= opens && hour < closes
  } else {
    false
  }
  println(isOpen)
}
/* Output:
Operating hours: 9 - 20
true
*/
```

A value defined inside a block of code, such as `opens`, is not accessible outside the scope of that block. Because they are defined *globally* to the `if` expression, `activity` and `hour` are accessible inside the `if` expression.

The result of an `if` expression is the result of the last expression of the chosen branch. Here, it's `hour >= opens && hour <= closes` which is `true`.

String Templates

You can insert a value within a `String` using `String` templates. Use a `$` before the identifier name:

```
// Summary1/StrTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")            // [1]
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [2]
  println("printing a $1")             // [3]
}
/* Output:
Found 42!
a
printing a $1
*/
```

- **[1]** `$answer` substitutes the value contained in `answer`.
- **[2]** `${if(condition) 'a' else 'b'}` evaluates and substitutes the result of the expression inside `${}`.
- **[3]** If the `$` is followed by anything unrecognizable as a program identifier, nothing special happens.

Use triple-quoted `Strings` to store multiline text or text with special characters:

```
// Summary1/ThreeQuotes.kt

fun json(q: String, a: Int) = """{
  "question" : "$q",
  "answer" : $a
}"""

fun main() {
  println(json("The Ultimate", 42))
}
/* Output:
{
  "question" : "The Ultimate",
  "answer" : 42
}
*/
```

You don't need to escape special characters like " within a triple-quoted `String`. (In a regular `String` you write `\"` to insert a double quote). As with normal `String`s, you can insert an identifier or an expression using $ inside a triple-quoted `String`.

Number Types

Kotlin provides integer types (`Int`, `Long`) and floating point types (`Double`). A whole number constant is `Int` by default and `Long` if you append an `L`. A constant is `Double` if it contains a decimal point:

```
// Summary1/NumberTypes.kt

fun main() {
  val n = 1000    // Int
  val l = 1000L   // Long
  val d = 1000.0  // Double
  println("$n $l $d")
}
/* Output:
1000 1000 1000.0
*/
```

An `Int` holds values between -2^{31} and $+2^{31}-1$. Integral values can overflow; for example, adding anything to `Int.MAX_VALUE` produces an overflow:

```
// Summary1/Overflow.kt

fun main() {
  println(Int.MAX_VALUE + 1)
  println(Int.MAX_VALUE + 1L)
}
/* Output:
-2147483648
2147483648
*/
```

In the second `println()` statement we append `L` to `1`, forcing the whole expression to be of type `Long`, which avoids the overflow. (A `Long` can hold values between -2^{63} and $+2^{63}-1$).

When you divide an `Int` with another `Int`, Kotlin produces an `Int` result, and any remainder is truncated. So `1/2` produces `0`. If a `Double` is involved, the `Int` is *promoted* to `Double` before the operation, so `1.0/2` produces `0.5`.

You might expect `d1` in the following to produce `3.4`:

```
// Summary1/Truncation.kt

fun main() {
  val d1: Double = 3.0 + 2 / 5
  println(d1)
  val d2: Double = 3 + 2.0 / 5
  println(d2)
}
/* Output:
3.0
3.4
*/
```

Because of evaluation order, it doesn't. Kotlin first divides `2` by `5`, and integer math produces `0`, yielding an answer of `3.0`. The same evaluation order *does* produce the expected result for `d2`. Dividing `2.0` by `5` produces `0.4`. The `3` is promoted to a `Double` because we add it to a `Double` (`0.4`), which produces `3.4`.

Understanding evaluation order helps you to decipher what a program does, both with logical operations (Boolean expressions) and with mathematical operations. If you're unsure about evaluation order, use parentheses to force your intention. This also makes it clear to those reading your code.

Repetition with `while`

A `while` loop continues as long as the controlling *Boolean-expression* produces `true`:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

The *Boolean expression* is evaluated once at the beginning of the loop and again before each further iteration.

```
// Summary1/While.kt

fun testCondition(i: Int) = i < 100

fun main() {
  var i = 0
  while (testCondition(i)) {
    print(".")
    i += 10
  }
}
/* Output:
..........
*/
```

Kotlin infers `Boolean` as the result type for `testCondition()`.

The short versions of assignment operators are available for all mathematical operations (`+=`, `-=`, `*=`, `/=`, `%=`). Kotlin also supports the increment and decrement operators `++` and `--`, in both prefix and postfix form.

`while` can be used with the `do` keyword:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

Rewriting `While.kt`:

```
// Summary1/DoWhile.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (testCondition(i))
}
/* Output:
..........
*/
```

The sole difference between `while` and `do-while` is that the body of the `do-while` always executes at least once, even if the Boolean expression produces `false` the first time.

Looping & Ranges

Many programming languages index into an iterable object by stepping through integers. Kotlin's `for` allows you to take elements directly from iterable objects like ranges and `String`s. For example, this `for` selects each character in the `String` `"Kotlin"`:

```
// Summary1/StringIteration.kt

fun main() {
  for (c in "Kotlin") {
    print("$c ")
    // c += 1 // error:
    // val cannot be reassigned
  }
}
/* Output:
K o t l i n
*/
```

`c` can't be explicitly defined as either a `var` or `val`—Kotlin automatically makes it a `val` and infers its type as `Char` (you can provide the type explicitly, but in practice this is rarely done).

You can step through integral values using *ranges*:

```
// Summary1/RangeOfInt.kt

fun main() {
  for (i in 1..10) {
    print("$i ")
  }
}
/* Output:
1 2 3 4 5 6 7 8 9 10
*/
```

Creating a range with `..` includes both bounds, but `until` excludes the top endpoint: `1 until 10` is the same as `1..9`. You can specify an increment value using `step`: `1..21 step 3`.

The `in` Keyword

The same `in` that provides `for` loop iteration also allows you to check membership in a range. `!in` returns `true` if the tested value *isn't* in the range:

```
// Summary1/Membership.kt

fun inNumRange(n: Int) = n in 50..100

fun notLowerCase(ch: Char) = ch !in 'a'..'z'

fun main() {
  val i1 = 11
  val i2 = 100
  val c1 = 'K'
  val c2 = 'k'
  println("$i1 ${inNumRange(i1)}")
  println("$i2 ${inNumRange(i2)}")
  println("$c1 ${notLowerCase(c1)}")
  println("$c2 ${notLowerCase(c2)}")
}
/* Output:
11 false
```

```
100 true
K true
k false
*/
```

`in` can also be used to test membership in floating-point ranges, although such ranges can only be defined using `..` and not `until`.

Expressions & Statements

The smallest useful fragment of code in most programming languages is either a *statement* or an *expression*. These have one basic difference:

- *A statement changes state.*
- *An expression expresses.*

That is, an expression produces a result, while a statement does not. Because it doesn't return anything, a statement must change the state of its surroundings (that is, create a *side effect*) to do anything useful.

Almost everything in Kotlin is an expression:

```
val hours = 10
val minutesPerHour = 60
val minutes = hours * minutesPerHour
```

In each case, everything to the right of the = is an expression, which produces a result that is assigned to the identifier on the left.

Functions like `println()` don't seem to produce a result, but because they are still expressions, they must return *something*. Kotlin has a special `Unit` type for these:

```
// Summary1/UnitReturn.kt

fun main() {
  val result = println("returns Unit")
  println(result)
}
/* Output:
returns Unit
kotlin.Unit
*/
```

Experienced programmers should go to Summary 2 after working the exercises for this atom.

Exercises and solutions can be found at www.AtomicKotlin.com.

Section II: Introduction to Objects

> *Objects* are the foundation for numerous modern languages, including Kotlin.

In an *object-oriented* (OO) programming language, you discover "nouns" in the problem you're solving, and translate those nouns to objects. Objects hold data and perform actions. An object-oriented language creates and uses objects.

Kotlin isn't just object-oriented; it's also *functional*. Functional languages focus on the actions you perform ("verbs"). Kotlin is a hybrid object-functional language.

- This section explains the basics of object-oriented programming.
- Section IV: Functional Programming introduces functional programming.
- Section V: Object-Oriented Programming covers object-oriented programming in detail.

Objects Everywhere

> Objects store data using *properties* (`vals` and `vars`) and perform operations with this data using functions.

Some definitions:

- *Class*: Defines properties and functions for what is essentially a new data type. Classes are also called *user-defined types*.
- *Member*: Either a property or a function of a class.
- *Member function*: A function that works only with a specific class of object.
- *Creating an object*: Making a `val` or `var` of a class. Also called *creating an instance* of that class.

Because classes define *state* and *behavior*, we can even refer to instances of built-in types like `Double` or `Boolean` as objects.

Consider Kotlin's `IntRange` class:

```
// ObjectsEverywhere/IntRanges.kt

fun main() {
  val r1 = IntRange(0, 10)
  val r2 = IntRange(5, 7)
  println(r1)
  println(r2)
}
/* Output:
0..10
5..7
*/
```

We create two objects (instances) of the `IntRange` *class*. Each object has its own piece of storage in memory. `IntRange` is a class, but a particular range `r1` from `0` to `10` is an object that is distinct from range `r2`.

Numerous operations are available for an `IntRange` object. Some are straightforward, like `sum()`, and others require more understanding before you can use them. If you try calling one that needs arguments, the IDE will ask for those arguments.

To learn about a particular member function, look it up in the Kotlin documentation[26]. Notice the magnifying glass icon in the top right area of the page. Click on that and type `IntRange` into the search box. Click on `kotlin.ranges > IntRange` from the resulting search. You'll see the documentation for the `IntRange` class. You can study all the member functions—the *Application Programming Interface* (API)—of the class. Although you won't understand most of it at this time, it's helpful to become comfortable looking things up in the Kotlin documentation.

An `IntRange` is a kind of object, and a defining characteristic of an object is that you perform operations on it. Instead of "performing an operation," we say *calling a member function*. To call a member function for an object, start with the object identifier, then a dot, then the name of the operation:

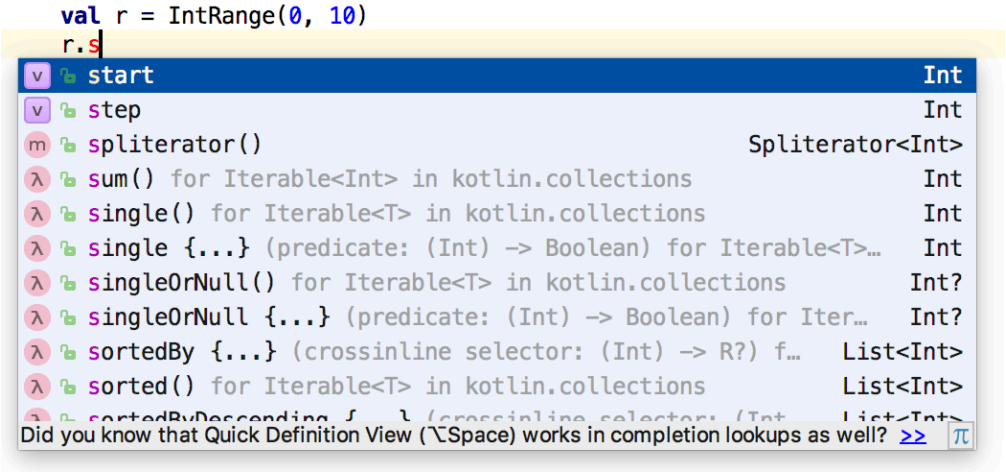
```
// ObjectsEverywhere/RangeSum.kt

fun main() {
  val r = IntRange(0, 10)
  println(r.sum())
}
/* Output:
55
*/
```

Because `sum()` is a member function defined for `IntRange`, you call it by saying `r.sum()`. This adds up all the numbers in that `IntRange`.

Earlier object-oriented languages used the phrase "sending a message" to describe calling a member function for an object. Sometimes you'll still see that terminology.

Classes can have many operations (member functions). It's easy to explore classes using an IDE (integrated development environment) that includes a feature called *code completion*. For example, if you type `.s` after an object identifier within IntelliJ IDEA, it shows all the members of that object that begin with `s`:

[26] https://kotlinlang.org/api/latest/jvm/stdlib/index.html



Code Completion

Try using code completion on other objects. For example, you can reverse a `String` or convert all the characters to lower case:

```
// ObjectsEverywhere/Strings.kt

fun main() {
  val s = "AbcD"
  println(s.reversed())
  println(s.lowercase())
}
/* Output:
DcbA
abcd
*/
```

You can easily convert a `String` to an integer and back:

```
// ObjectsEverywhere/Conversion.kt

fun main() {
  val s = "123"
  println(s.toInt())
  val i = 123
  println(i.toString())
}
/* Output:
123
123
*/
```

Later in the book we discuss strategies to handle situations when the `String` you want to convert doesn't represent a correct integer value.

You can also convert from one numerical type to another. To avoid confusion, conversions between number types are explicit. For example, you convert an `Int` `i` to a `Long` by calling `i.toLong()`, or to a `Double` with `i.toDouble()`:

```
// ObjectsEverywhere/NumberConversions.kt

fun fraction(numerator: Long, denom: Long) =
  numerator.toDouble() / denom

fun main() {
  val num = 1
  val den = 2
  val f = fraction(num.toLong(), den.toLong())
  println(f)
}
/* Output:
0.5
*/
```

Well-defined classes are easy for a programmer to understand, and produce code that's easy to read.

Exercises and solutions can be found at www.AtomicKotlin.com.

Creating Classes

> Not only can you use predefined types like `IntRange` and `String`, you can also create your own types of objects.

Indeed, creating new types comprises much of the activity in object-oriented programming. You create new types by defining *classes*.

An object is a piece of the solution for a problem you're trying to solve. Start by thinking of objects as expressing concepts. As a first approximation, if you discover a "thing" in your problem, represent that thing as an object in your solution.

Suppose you want to create a program to manage animals in a zoo. It makes sense to categorize the different types of animals based on how they behave, their needs, animals they get along with and those they fight with. Everything different about a species of animal is captured in the classification of that animal's object. Kotlin uses the `class` keyword to create a new type of object:

```
// CreatingClasses/Animals.kt

// Create some classes:
class Giraffe
class Bear
class Hippo

fun main() {
  // Create some objects:
  val g1 = Giraffe()
  val g2 = Giraffe()
  val b = Bear()
  val h = Hippo()

  // Each object() is unique:
  println(g1)
  println(g2)
  println(h)
```

```
  println(b)
}
/* Sample output:
Giraffe@28d93b30
Giraffe@1b6d3586
Hippo@4554617c
Bear@74a14482
*/
```

To define a class, start with the `class` keyword, followed by an identifier for your new class. The class name must begin with a letter (A-Z, upper or lower case), but can include things like numbers and underscores. Following convention, we capitalize the first letter of a class name, and lowercase the first letter of all `val`s and `var`s.

`Animals.kt` starts by defining three new classes, then creates four objects (also called *instances*) of those classes.

`Giraffe` is a class, but a particular five-year-old male giraffe that lives in Botswana is an *object*. Each object is different from all others, so we give them names like `g1` and `g2`.

Notice the rather cryptic output of the last four lines. The part before the `@` is the class name, and the number after the `@` is the address where the object is located in your computer's memory. Yes, that's a number even though it includes some letters—it's called "hexadecimal notation"[27]. Every object in your program has its own unique address.

The classes defined here (`Giraffe`, `Bear`, and `Hippo`) are as simple as possible: the entire class definition is a single line. More complex classes use curly braces (`{` and `}`) to create a *class body* containing the characteristics and behaviors for that class.

A function defined within a class belongs to that class. In Kotlin, we call these *member functions* of the class. Some object-oriented languages like Java choose to call them *methods*, a term that came from early object-oriented languages like Smalltalk. To emphasize the functional nature of Kotlin, the designers chose to drop the term *method*, as some beginners found the distinction confusing. Instead, the term *function* is used throughout the language.

If it is unambiguous, we will just say "function." If we must make the distinction:

[27] https://en.wikipedia.org/wiki/Hexadecimal

- *Member* functions belong to a class.
- *Top-level* functions exist by themselves and are not part of a class.

Here, `bark()` belongs to the `Dog` class:

```
// CreatingClasses/Dog.kt

class Dog {
  fun bark() = "yip!"
}

fun main() {
  val dog = Dog()
}
```

In `main()`, we create a `Dog` object and assign it to `val dog`. Kotlin emits a warning because we never use `dog`.

Member functions are called (*invoked*) with the object name, followed by a `.` (dot/period), followed by the function name and parameter list. Here we call the `meow()` function and display the result:

```
// CreatingClasses/Cat.kt

class Cat {
  fun meow() = "mrrrow!"
}

fun main() {
  val cat = Cat()
  // Call 'meow()' for 'cat':
  val m1 = cat.meow()
  println(m1)
}
/* Output:
mrrrow!
*/
```

A member function acts on a particular instance of a class. When you call `meow()`, you must call it with an object. During the call, `meow()` can access other members of that object.

When calling a member function, Kotlin keeps track of the object of interest by silently passing a reference to that object. That reference is available inside the member function by using the keyword `this`.

Member functions have special access to other elements within a class, simply by naming those elements. You can also explicitly *qualify* access to those elements using `this`. Here, `exercise()` calls `speak()` with and without qualification:

```
// CreatingClasses/Hamster.kt

class Hamster {
  fun speak() = "Squeak! "
  fun exercise() =
    this.speak() +   // Qualified with 'this'
      speak() +      // Without 'this'
      "Running on wheel"
}

fun main() {
  val hamster = Hamster()
  println(hamster.exercise())
}
/* Output:
Squeak! Squeak! Running on wheel
*/
```

In `exercise()`, we call `speak()` first with an explicit `this` and then omit the qualification.

Sometimes you'll see code containing an unnecessary explicit `this`. That kind of code often comes from programmers who know a different language where `this` is either required, or part of its style. Using a feature unnecessarily is confusing for the reader, who spends time trying to figure out why you're doing it. We recommend avoiding the unnecessary use of `this`.

Outside the class, you must say `hamster.exercise()` and `hamster.speak()`.

Exercises and solutions can be found at www.AtomicKotlin.com.

Properties

> A *property* is a `var` or `val` that's part of a class.

Defining a property *maintains state* within a class. Maintaining state is the primary motivating reason for creating a class rather than just writing one or more standalone functions.

A `var` property can be reassigned, while a `val` property can't. Each object gets its own storage for properties:

```
// Properties/Cup.kt

class Cup {
  var percentFull = 0
}

fun main() {
  val c1 = Cup()
  c1.percentFull = 50
  val c2 = Cup()
  c2.percentFull = 100

  println(c1.percentFull)
  println(c2.percentFull)
}
/* Output:
50
100
*/
```

Defining a `var` or `val` inside a class looks just like defining it within a function. However, the `var` or `val` becomes *part* of that class, and you must refer to it by specifying its object using *dot notation*, placing a dot between the object and the name of the property. You can see dot notation used for each reference to `percentFull`.

The `percentFull` property represents the state of the corresponding `Cup` object. `c1.percentFull` and `c2.percentFull` contain different values, showing that each object has its own storage.

A member function can refer to a property within its object without using dot notation (that is, without *qualifying* it):

```
// Properties/Cup2.kt

class Cup2 {
  var percentFull = 0
  val max = 100
  fun add(increase: Int): Int {
    percentFull += increase
    if (percentFull > max)
      percentFull = max
    return percentFull
  }
}

fun main() {
  val cup = Cup2()
  cup.add(50)
  println(cup.percentFull)
  cup.add(70)
  println(cup.percentFull)
}
/* Output:
50
100
*/
```

The `add()` member function tries to add `increase` to `percentFull` but ensures that it doesn't go past 100%.

You must qualify both properties and member functions from outside a class.

You can define top-level properties:

```
// Properties/TopLevelProperty.kt

val constant = 42

var counter = 0

fun inc() {
  counter++
}
```

Defining a top-level `val` is safe because it cannot be modified. However, defining a mutable (`var`) top-level property is considered an *anti-pattern*. As your program becomes more complicated, it becomes harder to reason correctly about *shared mutable state*. If everyone in your code base can access the `var counter`, you can't guarantee it will change correctly: while `inc()` increases `counter` by one, some other part of the program might decrease `counter` by ten, producing obscure bugs. It's best to guard mutable state within a class. In Constraining Visibility you'll see how to make it truly hidden.

To say that `var`s can be changed while `val`s cannot is an oversimplification. As an analogy, consider a `house` as a `val`, and a `sofa` inside the `house` as a `var`. You can modify `sofa` because it's a `var`. You can't reassign `house`, though, because it's a `val`:

```
// Properties/ChangingAVal.kt

class House {
  var sofa: String = ""
}

fun main() {
  val house = House()
  house.sofa = "Simple sleeper sofa: $89.00"
  println(house.sofa)
  house.sofa = "New leather sofa: $3,099.00"
  println(house.sofa)
  // Cannot reassign the val to a new House:
  // house = House()
}
/* Output:
Simple sleeper sofa: $89.00
New leather sofa: $3,099.00
*/
```

Although `house` is a `val`, its object can be modified because `sofa` in `class House` is a `var`. Defining `house` as a `val` only prevents it from being reassigned to a new object.

If we make a property a `val`, it cannot be reassigned:

```
// Properties/AnUnchangingVar.kt

class Sofa {
  val cover: String = "Loveseat cover"
}

fun main() {
  var sofa = Sofa()
  // Not allowed:
  // sofa.cover = "New cover"
  // Reassigning a var:
  sofa = Sofa()
}
```

Even though `sofa` is a `var`, its object cannot be modified because `cover` in `class Sofa` is a `val`. However, `sofa` can be reassigned to a new object.

We've talked about identifiers like `house` and `sofa` as if they were objects. They are actually *references* to objects. One way to see this is to observe that two identifiers can refer to the same object:

```
// Properties/References.kt

class Kitchen {
  var table: String = "Round table"
}

fun main() {
  val kitchen1 = Kitchen()
  val kitchen2 = kitchen1
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
  kitchen1.table = "Square table"
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
}
```

```
/* Output:
kitchen1: Round table
kitchen2: Round table
kitchen1: Square table
kitchen2: Square table
*/
```

When `kitchen1` modifies `table`, `kitchen2` sees the modification. `kitchen1.table` and `kitchen2.table` display the same output.

Remember that `var` and `val` control references rather than objects. A `var` allows you to rebind a reference to a different object, and a `val` prevents you from doing so.

Mutability means an object can change its state. In the examples above, `class House` and `class Kitchen` define mutable objects while `class Sofa` defines immutable objects.

Exercises and solutions can be found at www.AtomicKotlin.com.

Constructors

> You initialize a new object by passing information to a *constructor*.

Each object is an isolated world. A program is a collection of objects, so correct initialization of each individual object solves a large part of the initialization problem. Kotlin includes mechanisms to guarantee proper object initialization.

A constructor is like a special member function that initializes a new object. The simplest form of a constructor is a single-line class definition:

```
// Constructors/Wombat.kt

class Wombat

fun main() {
  val wombat = Wombat()
}
```

In `main()`, calling `Wombat()` creates a `Wombat` object. If you are coming from another object-oriented language you might expect to see a `new` keyword used here, but `new` would be redundant in Kotlin so it was omitted.

You pass information to a constructor using a parameter list, just like a function. Here, the `Alien` constructor takes a single argument:

```
// Constructors/Arg.kt

class Alien(name: String) {
  val greeting = "Poor $name!"
}

fun main() {
  val alien = Alien("Mr. Meeseeks")
  println(alien.greeting)
  // alien.name // Error     // [1]
}
/* Output:
Poor Mr. Meeseeks!
*/
```

Creating an `Alien` object requires the argument (try it without one). `name` initializes the `greeting` property within the constructor, but it is not accessible outside the constructor—try uncommenting line **[1]**.

If you want the constructor parameter to be accessible outside the class body, define it as a `var` or `val` in the parameter list:

```
// Constructors/VisibleArgs.kt

class MutableNameAlien(var name: String)

class FixedNameAlien(val name: String)

fun main() {
  val alien1 =
    MutableNameAlien("Reverse Giraffe")
  val alien2 =
    FixedNameAlien("Krombopulos Michael")

  alien1.name = "Parasite"
  // Can't do this:
  // alien2.name = "Parasite"
}
```

These class definitions have no explicit class bodies—the bodies are implied.

When `name` is defined as a `var` or `val`, it becomes a property and is thus accessible outside the constructor. `val` constructor parameters cannot be changed, while `var` constructor parameters are mutable.

Your class can have numerous constructor parameters:

```
// Constructors/MultipleArgs.kt

class AlienSpecies(
  val name: String,
  val eyes: Int,
  val hands: Int,
  val legs: Int
) {
  fun describe() =
    "$name with $eyes eyes, " +
      "$hands hands and $legs legs"
}

fun main() {
  val kevin =
    AlienSpecies("Zigerion", 2, 2, 2)
  val mortyJr =
    AlienSpecies("Gazorpian", 2, 6, 2)
  println(kevin.describe())
  println(mortyJr.describe())
}
/* Output:
Zigerion with 2 eyes, 2 hands and 2 legs
Gazorpian with 2 eyes, 6 hands and 2 legs
*/
```

In Complex Constructors, you'll see that constructors can also contain complex initialization logic.

If an object is used when a `String` is expected, Kotlin calls the object's `toString()` member function. If you don't write one, you still get a default `toString()`:

```
// Constructors/DisplayAlienSpecies.kt

fun main() {
  val krombopulosMichael =
    AlienSpecies("Gromflomite", 2, 2, 2)
  println(krombopulosMichael)
}
/* Sample output:
AlienSpecies@4d7e1886
*/
```

The default `toString()` isn't very useful—it produces the class name and the physical address of the object (this varies from one program execution to the next). You can define your own `toString()`:

```
// Constructors/Scientist.kt

class Scientist(val name: String) {
  override fun toString() =
    "Scientist('$name')"
}

fun main() {
  val zeep = Scientist("Zeep Xanflorp")
  println(zeep)
}
/* Output:
Scientist('Zeep Xanflorp')
*/
```

`override` is a new keyword for us. It is required here because `toString()` already has a definition, the one producing the primitive result. `override` tells Kotlin that yes, we do actually want to replace the default `toString()` with our own definition. The explicitness of `override` clarifies the code and prevents mistakes.

A `toString()` that displays the contents of an object in a convenient form is useful for finding and fixing programming errors. To simplify the process of *debugging*, IDEs provide *debuggers*[28] that allow you to observe each step in the execution of a program and to see inside your objects.

Exercises and solutions can be found at www.AtomicKotlin.com.

[28] https://www.jetbrains.com/help/idea/debugging-code.html

Constraining Visibility

> If you leave a piece of code for a few days or weeks, then come back to it, you might see a much better way to write it.

This is one of the prime motivations for *refactoring*, which rewrites working code to make it more readable, understandable, and thus maintainable.

There is a tension in this desire to change and improve your code. Consumers (*client programmers*) require aspects of your code to be stable. You want to change it, and they want it to stay the same.

This is particularly important for libraries. Consumers of a library don't want to rewrite code for a new version of that library. However, the library creator must be free to make modifications and improvements, with the certainty that the client code won't be affected by those changes.

Therefore, a primary consideration in software design is:

> *Separate things that change from things that stay the same.*

To control visibility, Kotlin and some other languages provide *access modifiers*. Library creators decide what is and is not accessible by the client programmer using the modifiers `public`, `private`, `protected`, and `internal`. This atom covers `public` and `private`, with a brief introduction to `internal`. We explain `protected` later in the book.

An access modifier such as `private` appears before the definition for a class, function, or property. An access modifier only controls access for that particular definition.

A `public` definition is accessible by client programmers, so changes to that definition impact client code directly. If you don't provide a modifier, your definition is automatically `public`, so `public` is technically redundant. You will sometimes still specify `public` for the sake of clarity.

A `private` definition is hidden and only accessible from other members of the same class. Changing, or even removing, a `private` definition doesn't directly impact client programmers.

`private` classes, top-level functions, and top-level properties are accessible only inside that file:

```
// Visibility/RecordAnimals.kt

private var index = 0                   // [1]

private class Animal(val name: String) // [2]

private fun recordAnimal(               // [3]
  animal: Animal
) {
  println("Animal #$index: ${animal.name}")
  index++
}

fun recordAnimals() {
  recordAnimal(Animal("Tiger"))
  recordAnimal(Animal("Antelope"))
}

fun recordAnimalsCount() {
  println("$index animals are here!")
}
```

You can access `private` top-level properties (**[1]**), classes (**[2]**), and functions (**[3]**) from other functions and classes within `RecordAnimals.kt`. Kotlin prevents you from accessing a `private` top-level element from within another file, telling you it's `private` in the file:

```
// Visibility/ObserveAnimals.kt

fun main() {
  // Can't access private members
  // declared in another file.
  // Class is private:
  // val rabbit = Animal("Rabbit")
  // Function is private:
  // recordAnimal(rabbit)
  // Property is private:
  // index++

  recordAnimals()
  recordAnimalsCount()
}
/* Output:
Animal #0: Tiger
Animal #1: Antelope
2 animals are here!
*/
```

Privacy is most commonly used for members of a class:

```
// Visibility/Cookie.kt

class Cookie(
  private var isReady: Boolean  // [1]
) {
  private fun crumble() =       // [2]
    println("crumble")

  public fun bite() =           // [3]
    println("bite")

  fun eat() {                   // [4]
    isReady = true              // [5]
    crumble()
    bite()
  }
}

fun main() {
```

```
  val x = Cookie(false)
  x.bite()
  // Can't access private members:
  // x.isReady
  // x.crumble()
  x.eat()
}
/* Output:
bite
crumble
bite
*/
```

- **[1]** A `private` property, not accessible outside the containing class.
- **[2]** A `private` member function.
- **[3]** A `public` member function, accessible to anyone.
- **[4]** No access modifier means `public`.
- **[5]** Only members of the same class can access `private` members.

The `private` keyword means no one can access that member except other members of that class. Other classes cannot access `private` members, so it's as if you're also insulating the class against yourself and your collaborators. With `private`, you can freely change that member without worrying whether it affects another class in the same package. As a library designer you'll typically keep things as `private` as possible, and expose only functions and classes to client programmers.

Any member function that is a *helper function* for a class can be made `private` to ensure you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing that function.

The same is true for a `private` property inside a class. Unless you must expose the underlying implementation (which is less likely than you might think), make properties `private`. However, just because a reference to an object is `private` inside a class doesn't mean some other object can't have a `public` reference to the same object:

```
// Visibility/MultipleRef.kt

class Counter(var start: Int) {
  fun increment() {
    start += 1
  }
  override fun toString() = start.toString()
}

class CounterHolder(counter: Counter) {
  private val ctr = counter
  override fun toString() =
    "CounterHolder: " + ctr
}

fun main() {
  val c = Counter(11)                  // [1]
  val ch = CounterHolder(c)            // [2]
  println(ch)
  c.increment()                        // [3]
  println(ch)
  val ch2 = CounterHolder(Counter(9)) // [4]
  println(ch2)
}
/* Output:
CounterHolder: 11
CounterHolder: 12
CounterHolder: 9
*/
```

- **[1]** `c` is now defined in the scope *surrounding* the creation of the `CounterHolder` object on the following line.
- **[2]** Passing `c` as the argument to the `CounterHolder` constructor means that the new `CounterHolder` now refers to the same `Counter` object that `c` refers to.
- **[3]** The `Counter` that is supposedly `private` inside `ch` can still be manipulated via `c`.
- **[4]** `Counter(9)` has no other references except within `CounterHolder`, so it cannot be accessed or modified by anything except `ch2`.

Maintaining multiple references to a single object is called *aliasing* and can produce surprising behavior.

Modules

Unlike the small examples in this book, real programs are often large. It can be helpful to divide such programs into one or more *modules*. A module is a logically independent part of a codebase. The way you divide a project into modules depends on the build system (such as Gradle[29] or Maven[30]) and is beyond the scope of this book.

An `internal` definition is accessible only inside the module where it is defined. `internal` lands somewhere between `private` and `public`—use it when `private` is too restrictive but you don't want an element to be a part of the `public` API. We do not use `internal` in the book's examples or exercises.

Modules are a higher-level concept. The following atom introduces *packages*, which enable finer-grained structuring. A library is often a single module consisting of multiple packages, so `internal` elements are available within the library but are not accessible by consumers of that library.

Exercises and solutions can be found at www.AtomicKotlin.com.

[29] https://gradle.org/
[30] https://maven.apache.org/

Packages

> A fundamental principle in programming is the acronym DRY: *Don't Repeat Yourself.*

Multiple identical pieces of code require maintenance whenever you make fixes or improvements. So duplicating code is not just extra work—every duplication creates opportunities for mistakes.

The `import` keyword reuses code from other files. One way to use `import` is to specify a class, function or property name:

```
import packagename.ClassName
import packagename.functionName
import packagename.propertyName
```

A *package* is an associated collection of code. Each package is usually designed to solve a particular problem, and often contains multiple functions and classes. For example, we can import mathematical constants and functions from the `kotlin.math` library:

```
// Packages/ImportClass.kt
import kotlin.math.PI
import kotlin.math.cos  // Cosine

fun main() {
  println(PI)
  println(cos(PI))
  println(cos(2 * PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

Sometimes you want to use multiple third-party libraries containing classes or functions with the same name. The `as` keyword allows you to change names while importing:

```
// Packages/ImportNameChange.kt
import kotlin.math.PI as circleRatio
import kotlin.math.cos as cosine

fun main() {
  println(circleRatio)
  println(cosine(circleRatio))
  println(cosine(2 * circleRatio))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

`as` is useful if a library name is poorly chosen or excessively long.

You can fully qualify an import in the body of your code. In the following example, the code might be less readable due to the explicit package names, but the origin of each element is absolutely clear:

```
// Packages/FullyQualify.kt

fun main() {
  println(kotlin.math.PI)
  println(kotlin.math.cos(kotlin.math.PI))
  println(kotlin.math.cos(2 * kotlin.math.PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

To import everything from a package, use a star:

```
// Packages/ImportEverything.kt
import kotlin.math.*

fun main() {
  println(E)
  println(E.roundToInt())
  println(E.toInt())
}
/* Output:
2.718281828459045
3
2
*/
```

The `kotlin.math` package contains a convenient `roundToInt()` that rounds the `Double` value to the nearest integer, unlike `toInt()` which simply truncates anything after a decimal point.

To reuse your code, create a package using the `package` keyword. The `package` statement must be the first non-comment statement in the file. `package` is followed by the name of your package, which by convention is all lowercase:

```
// Packages/PythagoreanTheorem.kt
package pythagorean
import kotlin.math.sqrt

class RightTriangle(
  val a: Double,
  val b: Double
) {
  fun hypotenuse() = sqrt(a * a + b * b)
  fun area() = a * b / 2
}
```

You can name the source-code file anything you like, unlike Java which requires the file name to be the same as the class name.

Kotlin allows you to choose any name for your package, but it's considered good style for the package name to be identical to the directory name where the package files are located (this will not always be the case for the examples in this book).

The elements in the `pythagorean` package are now available using `import`:

```
// Packages/ImportPythagorean.kt
import pythagorean.RightTriangle

fun main() {
  val rt = RightTriangle(3.0, 4.0)
  println(rt.hypotenuse())
  println(rt.area())
}
/* Output:
5.0
6.0
*/
```

In the remainder of this book we use `package` statements for any file that defines functions, classes, etc., outside of `main()`, to prevent name clashes with other files in the book, but we usually won't put a `package` statement in a file that *only* contains a `main()`.

Exercises and solutions can be found at www.AtomicKotlin.com.

Testing

> Constant testing is essential for rapid program development.

If changing one part of your code breaks other code, your tests reveal the problem right away. If you don't find out immediately, changes accumulate and you can no longer tell which change caused the problem. You'll spend a *lot* longer tracking it down.

Testing is a crucial practice, so we introduce it early and use it throughout the rest of the book. This way, you become accustomed to testing as a standard part of the programming process.

Using `println()` to verify code correctness is a weak approach—you must scrutinize the output every time and consciously ensure that it's correct.

To simplify your experience while using this book, we created our own tiny testing system. The goal is a minimal approach that:

1. Shows the expected result of expressions.
2. Provides output so you know the program is running, even when all tests succeed.
3. Ingrains the concept of testing early in your practice.

Although useful for this book, ours is *not* a testing system for the workplace. Others have toiled long and hard to create such test systems. For example:

- JUnit[31] is one of the most popular Java test frameworks, and is easily used from within Kotlin.
- Kotest[32] is designed specifically for Kotlin, and takes advantage of Kotlin language features.
- The Spek Framework[33] produces a different form of testing, called *Specification Testing*.

To use our testing framework, we must first `import` it. The basic elements of the framework are `eq` (*equals*) and `neq` (*not equals*):

[31] https://junit.org
[32] https://github.com/kotest/kotest
[33] https://spekframework.org/

```
// Testing/TestingExample.kt
import atomictest.*

fun main() {
  val v1 = 11
  val v2 = "Ontology"

  // 'eq' means "equals":
  v1 eq 11
  v2 eq "Ontology"

  // 'neq' means "not equal"
  v2 neq "Epistemology"

  // [Error] Epistemology != Ontology
  // v2 eq "Epistemology"
}
/* Output:
11
Ontology
Ontology
*/
```

The code for the `atomictest` package is in Appendix A: AtomicTest. We don't intend that you understand everything in `AtomicTest.kt` right now, because it uses some features that won't appear until later in the book.

To produce a clean, comfortable appearance, `AtomicTest` uses a Kotlin feature you haven't seen yet: the ability to write a function call `a.function(b)` in the text-like form `a function b`. This is called *infix notation*. Only functions defined using the `infix` keyword can be called this way. `AtomicTest.kt` defines the `infix` `eq` and `neq` used in `TestingExample.kt`:

```
expression eq expected
expression neq expected
```

`eq` and `neq` are flexible—almost anything works as a test expression. If *expected* is a `String`, then *expression* is converted to a `String` and the two `Strings` are compared. Otherwise, *expression* and *expected* are compared directly (without converting them first). In either case, the result of *expression* appears on the console so you see something when the program runs. Even when the tests succeed, you still see

the result on the left of `eq` or `neq`. If *expression* and *expected* are not equivalent, `AtomicTest` shows an error when the program runs.

The last test in `TestingExample.kt` intentionally fails so you see an example of failure output. If the two values are not equal, Kotlin displays the corresponding message starting with `[Error]`. If you uncomment the last line and run the example above, you will see, after all the successful tests:

```
[Error] Epistemology != Ontology
```

The actual value stored in `v2` is not what it is claimed to be in the "expected" expression. `AtomicTest` displays the `String` representations for both expected and actual values.

`eq` and `neq` are the basic (infix) functions defined for `AtomicTest`—it truly is a minimal testing system. When you put `eq` and `neq` expressions in your examples, you'll create both a test and some console output. You verify the correctness of the program by running it.

There's a second tool in `AtomicTest`. The `trace` object captures output for later comparison:

```
// Testing/Trace1.kt
import atomictest.*

fun main() {
  trace("line 1")
  trace(47)
  trace("line 2")
  trace eq """
    line 1
    47
    line 2
  """
}
```

Adding results to `trace` looks like a function call, so you can effectively replace `println()` with `trace()`.

In previous atoms, we displayed output and relied on human visual inspection to catch any discrepancies. That's unreliable; even in a book where we scrutinize the

code over and over, we've learned that visual inspection can't be trusted to find errors. From now on we rarely use commented output blocks because `AtomicTest` will do everything for us. However, sometimes we still include commented output blocks when that produces a more useful effect.

Seeing the benefits of using testing throughout the rest of the book should help you incorporate testing into your programming process. You'll probably start feeling uncomfortable when you see code that doesn't have tests. You might even decide that code without tests is broken by definition.

Testing as Part of Programming

Testing is most effective when it's built into your software development process. Writing tests ensures you get the results you expect. Many people advocate writing tests *before* writing the implementation code—you first make the test fail before you write the code to make it pass. This technique, called *Test Driven Development* (TDD), is a way to ensure that you're really testing what you think you are. You'll find a more complete description of TDD on Wikipedia (search for "Test Driven Development").

There's another benefit to writing testably—it changes the way you craft your code. You could just display the results on the console. But in the test mindset you wonder, "How will I test this?" When you create a function, you decide you should return something from the function, if for no other reason than to test that result. Functions that do nothing but take input and produce output tend to generate better designs, as well.

Here's a simplified example using TDD to implement the BMI calculation from Number Types. First, we write the tests, along with an initial implementation that fails (because we haven't yet implemented the functionality):

```
// Testing/TDDFail.kt
package testing1
import atomictest.eq

fun main() {
  calculateBMI(160, 68) eq "Normal weight"
//  calculateBMI(100, 68) eq "Underweight"
//  calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(lbs: Int, height: Int) =
  "Normal weight"
```

Only the first test passes. The other tests fail and are commented. Next, we add code to determine which weights are in which categories. Now *all* the tests fail:

```
// Testing/TDDStillFails.kt
package testing2
import atomictest.eq

fun main() {
  // Everything fails:
  // calculateBMI(160, 68) eq "Normal weight"
  // calculateBMI(100, 68) eq "Underweight"
  // calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(
  lbs: Int,
  height: Int
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

We're using `Int`s instead of `Double`s, producing a zero result. The tests guide us to the fix:

```
// Testing/TDDWorks.kt
package testing3
import atomictest.eq

fun main() {
  calculateBMI(160.0, 68.0) eq "Normal weight"
  calculateBMI(100.0, 68.0) eq "Underweight"
  calculateBMI(200.0, 68.0) eq "Overweight"
}

fun calculateBMI(
  lbs: Double,
  height: Double
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

You may choose to add additional tests for the boundary conditions.

In the exercises for this book, we include tests that your code must pass.

Exercises and solutions can be found at www.AtomicKotlin.com.

Exceptions

> The word "exception" is used in the same sense as the phrase "I take exception to that."

An exceptional condition prevents the continuation of the current function or scope. At the point the problem occurs, you might not know what to do with it, but you cannot continue within the current context. You don't have enough information to fix the problem. So you must stop and hand the problem to another context that's able to take appropriate action.

This atom covers the basics of *exceptions* as an error-reporting mechanism. In Section VI: Preventing Failure, we look at other ways to deal with problems.

It's important to distinguish an exceptional condition from a normal problem. A normal problem has enough information in the current context to cope with the issue. With an exceptional condition, you cannot continue processing. All you can do is leave, relegating the problem to an external context. This is what happens when you *throw an exception*. The exception is the object that is "thrown" from the site of the error.

Consider `toInt()`, which converts a `String` to an `Int`. What happens if you call this function for a `String` that doesn't contain an integer value?

```
// Exceptions/ToIntException.kt
package exceptions

fun erroneousCode() {
  // Uncomment this line to get an exception:
  // val i = "1$".toInt()        // [1]
}

fun main() {
  erroneousCode()
}
```

Uncommenting line **[1]** produces an exception. Here, the failing line is commented so we don't stop the book's build, which checks whether each example compiles and runs as expected.

When an exception is thrown, the path of execution—the one that can't be continued—stops, and the exception object ejects from the current context. Here, it exits the context of `erroneousCode()` and goes out to the context of `main()`. In this case, Kotlin only reports the error; the programmer has presumably made a mistake and must fix the code.

When an exception isn't caught, the program aborts and displays a *stack trace* containing detailed information. Uncommenting line **[1]** in `ToIntException.kt`, produces the following output:

```
Exception in thread "main" java.lang.NumberFormatException: For input s\
tring: "1$"
  at java.lang.NumberFormatException.forInputString(NumberFormatExcepti\
on.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at ToIntExceptionKt.erroneousCode(at ToIntException.kt:6)
  at ToIntExceptionKt.main(at ToIntException.kt:10)
```

The stack trace gives details such as the file and line where the exception occurred, so you can quickly discover the issue. The last two lines show the problem: in line 10 of `main()` we call `erroneousCode()`. Then, more precisely, in line 6 of `erroneousCode()` we call `toInt()`.

To avoid commenting and uncommenting code to display exceptions, we use the `capture()` function from the `AtomicTest` package:

```
// Exceptions/IntroducingCapture.kt
import atomictest.*

fun main() {
  capture {
    "1$".toInt()
  } eq "NumberFormatException: " +
    """For input string: "1$""""
}
```

Using `capture()`, we compare the generated exception to the expected error message. `capture()` isn't very helpful for normal programming—it's designed specifically for this book, so you can see the exception and know that the output has been checked by the book's build system.

Another strategy when you can't successfully produce the expected result is to return `null`, which is a special constant denoting "no value." You can return `null` instead of a value of any type. Later in Nullable Types we discuss the way `null` affects the type of the resulting expression.

The Kotlin standard library contains `String.toIntOrNull()` which performs the conversion if the `String` contains an integer number, or produces `null` if the conversion is impossible—`null` is a simple way to indicate failure:

```
// Exceptions/IntroducingNull.kt
import atomictest.eq

fun main() {
  "1$".toIntOrNull() eq null
}
```

Suppose we calculate average income over a period of months:

```
// Exceptions/AverageIncome.kt
package firstversion
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "ArithmeticException: / by zero"
}
```

If `months` is zero, the division in `averageIncome()` throws an `ArithmeticException`. Unfortunately, this doesn't tell us anything about why the error occurred, what the denominator means and whether it can legally be zero in the first place. This is clearly a bug in the code—`averageIncome()` should cope with a `months` of `0` in a way that prevents a divide-by-zero error.

Let's modify `averageIncome()` to produce more information about the source of the problem. If `months` is zero, we can't return a regular integer value as a result. One strategy is to return `null`:

```
// Exceptions/AverageIncomeWithNull.kt
package withnull
import atomictest.eq

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    null
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  averageIncome(5000, 0) eq null
}
```

If a function can return `null`, Kotlin requires that you check the result before using it (this is covered in Nullable Types). Even if you only want to display output to the

user, it's better to say "No full month periods have passed," rather than "Your average income for the period is: null."

Instead of executing `averageIncome()` with the wrong arguments, you can throw an exception—escape and force some other part of the program to manage the issue. You *could* just allow the default `ArithmeticException`, but it's often more useful to throw a specific exception with a detailed error message. When, after a couple of years in production, your application suddenly throws an exception because a new feature calls `averageIncome()` without properly checking the arguments, you'll be grateful for that message:

```
// Exceptions/AverageIncomeWithException.kt
package properexception
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    throw IllegalArgumentException(    // [1]
      "Months can't be zero")
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "IllegalArgumentException: " +
    "Months can't be zero"
}
```

- **[1]** When throwing an exception, the `throw` keyword is followed by the exception to be thrown, along with any arguments it might need. Here we use the standard exception class `IllegalArgumentException`.

Your goal is to generate the most useful messages possible to simplify the support of your application in the future. Later you'll learn to define your own exception types and make them specific to your circumstances.

Exercises and solutions can be found at www.AtomicKotlin.com.

Lists

> A `List` is a *container*, which is an object that holds other objects.

Containers are also called *collections*. When we need a basic container for the examples in this book, we normally use a `List`.

`Lists` are part of the standard Kotlin package so they don't require an `import`.

The following example creates a `List` populated with `Int`s by calling the standard library function `listOf()` with initialization values:

```
// Lists/Lists.kt
import atomictest.eq

fun main() {
  val ints = listOf(99, 3, 5, 7, 11, 13)
  ints eq "[99, 3, 5, 7, 11, 13]"   // [1]

  // Select each element in the List:
  var result = ""
  for (i in ints) {                 // [2]
    result += "$i "
  }
  result eq "99 3 5 7 11 13"

  // "Indexing" into the List:
  ints[4] eq 11                     // [3]
}
```

- **[1]** A `List` uses square brackets when displaying itself.
- **[2]** `for` loops work well with `Lists`: `for(i in ints)` means `i` receives each value in `ints`. You don't declare `val i` or give its type; Kotlin knows from the context that `i` is a `for` loop identifier.

- **[3]** Square brackets *index* into a `List`. A `List` keeps its elements in initialization order, and you select them individually by number. Like most programming languages, Kotlin starts indexing at element zero, which in this case produces the value `99`. Thus an index of `4` produces the value `11`.

Forgetting that indexing starts at zero produces the so-called *off-by-one* error. In a language like Kotlin we often don't select elements one at a time, but instead *iterate* through an entire container using `in`. This eliminates off-by-one errors.

If you use an index beyond the last element in a `List`, Kotlin throws an `ArrayIndexOutOfBoundsException`:

```
// Lists/OutOfBounds.kt
import atomictest.*

fun main() {
  val ints = listOf(1, 2, 3)
  capture {
    ints[3]
  } contains
    listOf("ArrayIndexOutOfBoundsException")
}
```

A `List` can hold all different types. Here's a `List` of `Doubles` and a `List` of `Strings`:

```
// Lists/ListUsefulFunction.kt
import atomictest.eq

fun main() {
  val doubles =
    listOf(1.1, 2.2, 3.3, 4.4)
  doubles.sum() eq 11.0

  val strings = listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings eq listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings.sorted() eq listOf("And",
    "Brillig", "Slithy", "Toves", "Twas")
  strings.reversed() eq listOf("Toves",
    "Slithy", "And", "Brillig", "Twas")
  strings.first() eq "Twas"
```

```
  strings.takeLast(2) eq
    listOf("Slithy", "Toves")
}
```

This shows some of `List`'s operations. Note the name "sorted" instead of "sort." When you call `sorted()` it *produces* a new `List` containing the same elements as the old, in sorted order—but it leaves the original `List` alone. Calling it "sort" implies that the original `List` is changed directly (a.k.a. *sorted in place*). Throughout Kotlin, you see this tendency of "leaving the original object alone and producing a new object." `reversed()` also produces a new `List`.

Parameterized Types

We consider it good practice to use type inference—it tends to make the code cleaner and easier to read. Sometimes, however, Kotlin complains that it can't figure out what type to use, and in other cases explicitness makes the code more understandable. Here's how we tell Kotlin the type contained by a `List`:

```
// Lists/ParameterizedTypes.kt
import atomictest.eq

fun main() {
  // Type is inferred:
  val numbers = listOf(1, 2, 3)
  val strings =
    listOf("one", "two", "three")
  // Exactly the same, but explicitly typed:
  val numbers2: List<Int> = listOf(1, 2, 3)
  val strings2: List<String> =
    listOf("one", "two", "three")
  numbers eq numbers2
  strings eq strings2
}
```

Kotlin uses the initialization values to infer that `numbers` contains a `List` of `Int`s, while `strings` contains a `List` of `String`s.

`numbers2` and `strings2` are explicitly-typed versions of `numbers` and `strings`, created by adding the type declarations `List<Int>` and `List<String>`. You haven't

seen angle brackets before—they denote a *type parameter*, allowing you to say, "this container holds 'parameter' objects." We pronounce `List<Int>` as "`List` of `Int`."

Type parameters are useful for components other than containers, but you often see them with container-like objects.

Return values can also have type parameters:

```
// Lists/ParameterizedReturn.kt
package lists
import atomictest.eq

// Return type is inferred:
fun inferred(p: Char, q: Char) =
  listOf(p, q)

// Explicit return type:
fun explicit(p: Char, q: Char): List<Char> =
  listOf(p, q)

fun main() {
  inferred('a', 'b') eq "[a, b]"
  explicit('y', 'z') eq "[y, z]"
}
```

Kotlin infers the return type for `inferred()`, while `explicit()` specifies the function return type. You can't just say it returns a `List`; Kotlin will complain, so you must give the type parameter as well. When you specify the return type of a function, Kotlin enforces your intention.

Read-Only and Mutable Lists

If you don't explicitly say you want a mutable `List`, you won't get one. `listOf()` produces a read-only `List` that has no mutating functions.

If you're creating a `List` gradually (that is, you don't have all the elements at creation time), use `mutableListOf()`. This produces a `MutableList` that can be modified:

```
// Lists/MutableList.kt
import atomictest.eq

fun main() {
  val list = mutableListOf<Int>()

  list.add(1)
  list.addAll(listOf(2, 3))

  list += 4
  list += listOf(5, 6)

  list eq listOf(1, 2, 3, 4, 5, 6)
}
```

Because `list` has no initial elements, we must tell Kotlin what type it is by providing the `<Int>` specification in the call to `mutableListOf()`. You can add elements to a `MutableList` using `add()` and `addAll()`, or the operator `+=` which adds either a single element or another collection.

A `MutableList` can be treated as a `List`, in which case it cannot be changed. You can't, however, treat a read-only `List` as a `MutableList`:

```
// Lists/MutListIsList.kt
package lists
import atomictest.eq

fun makeList(): List<Int> =
  mutableListOf(1, 2, 3)

fun main() {
  // makeList() produces a read-only List:
  val list = makeList()
  // list.add(3) // Unresolved reference: add
  list eq listOf(1, 2, 3)
}
```

`list` lacks mutation functions despite being originally created using `mutableListOf()` inside `makeList()`. Notice that the result type of `makeList()` is `List<Int>`. The original object is still a `MutableList`, but it is viewed through the lens of a `List`.

A `List` is *read-only*—you can read its contents but not write to it. If the underlying implementation is a `MutableList` and you retain a mutable reference to that implementation, you can still modify it via that mutable reference, and any read-only references will see those changes. This is another example of *aliasing*, introduced in Constraining Visibility:

```
// Lists/MultipleListRefs.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first.add(2)
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first` is an immutable reference (`val`) to the mutable object produced by `mutableListOf(1)`. When `second` is aliased to `first` it becomes a view of that same object. `second` is read-only because `List<Int>` does not include modification functions. Without the explicit `List<Int>` type declaration, Kotlin would infer that `second` was also a reference to a mutable object.

We're able to add an element (2) to the object because `first` is a reference to a mutable `List`. Note that `second` observes these changes—it cannot change the `List` although the `List` changes via `first`.

The += Puzzle

The `+=` operator can give the appearance that an immutable `List` is actually mutable:

```
// Lists/ApparentlyMutableList.kt
import atomictest.eq

fun main() {
  var list = listOf('X') // Immutable
  list += 'Y' // Appears to be mutable
  list eq "[X, Y]"
}
```

`listOf()` produces an immutable `List`, but `list += 'Y'` seems to be modifying that `List`. Does `+=` somehow violate immutability?

This only happens because `list` is a `var`. Here's a more detailed example that shows the different combinations of mutable/immutable `Lists` with `val`/`var`:

```
// Lists/PlusAssignPuzzle.kt
import atomictest.eq

fun main() {
    // Mutable List assigned to a 'val'/'var':
    val list1 = mutableListOf('A') // or 'var'
    list1 += 'A' // Is the same as:
    list1.plusAssign('A')              // [1]

    // Immutable List assigned to a 'val':
    val list2 = listOf('B')
    // list2 += 'B' // Is the same as:
    // list2 = list2 + 'B'             // [2]

    // Immutable List assigned to a 'var':
    var list3 = listOf('C')
    list3 += 'C' // Is the same as:
    val newList = list3 + 'C'          // [3]
    list3 = newList                    // [4]

    list1 eq "[A, A, A]"
    list2 eq "[B]"
    list3 eq "[C, C, C]"
}
```

- **[1]** `list1` refers to a mutable object, which can therefore be modified in place. The compiler translates `+=` to the `plusAssign()` call. It doesn't matter if `list1`

is a `val` or a `var` because nothing is ever *reassigned* to `list1` after creation—it always refers to the same mutable list. Make it a `var` and IntelliJ points out that it never changes and suggests that it be a `val`.

- **[2]** This tries to create a new `List` by combining `list2` and `'B'`, but it can't reassign that new `List` to `list2` because `list2` is a `val`. Without the ability to perform that reassignment, the `+=` cannot compile.
- **[3]** Creates `newList` without modifying the existing immutable `List` referred to by `list3`.
- **[4]** Because `list3` is a `var`, the compiler assigns `newList` back into `list3`. The previous contents of `list3` are then forgotten, and it appears that `list3` has been mutated. Actually, the old `list3` has been discarded and replaced by the newly-created `newList`, giving the illusion that `list3` is mutable.

This behavior of `+=` happens with other collections, as well. The resulting confusion is another reason to choose `val` over `var` for your identifiers.

Exercises and solutions can be found at www.AtomicKotlin.com.

Variable Argument Lists

> The `vararg` keyword produces a flexibly-sized argument list.

In Lists we introduced `listOf()`, which takes any number of parameters and produces a `List`:

```
// Varargs/ListOf.kt
import atomictest.eq

fun main() {
  listOf(1) eq "[1]"
  listOf("a", "b") eq "[a, b]"
}
```

Using the `vararg` keyword, you can define a function that takes any number of arguments, just like `listOf()` does. `vararg` is short for *variable argument list*:

```
// Varargs/VariableArgList.kt
package varargs

fun v(s: String, vararg d: Double) {}

fun main() {
  v("abc", 1.0, 2.0)
  v("def", 1.0, 2.0, 3.0, 4.0)
  v("ghi", 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
}
```

A function definition may specify only one parameter as `vararg`. Although it's possible to specify any item in the parameter list as `vararg`, it's usually simplest to do it for the last one.

`vararg` allows you to pass any number (including zero) of arguments. All arguments must be of the specified type. `vararg` arguments are accessed using the parameter name, which becomes an `Array`:

```
// Varargs/VarargSum.kt
package varargs
import atomictest.eq

fun sum(vararg numbers: Int): Int {
  var total = 0
  for (n in numbers) {
    total += n
  }
  return total
}

fun main() {
  sum(13, 27, 44) eq 84
  sum(1, 3, 5, 7, 9, 11) eq 36
  sum() eq 0
}
```

Although `Arrays` and `Lists` look similar, they are implemented differently—`List` is a regular library class while `Array` has special low-level support. `Array` comes from Kotlin's requirement for compatibility with other languages, especially Java.

In day-to-day programming, use a `List` when you need a simple sequence. Use `Arrays` only when a third-party API requires an `Array`, or when you're dealing with `varargs`.

In most cases you can just ignore the fact that `vararg` produces an `Array` and treat it as if it were a `List`:

```
// Varargs/VarargLikeList.kt
package varargs
import atomictest.eq

fun evaluate(vararg ints: Int) =
  "Size: ${ints.size}\n" +
  "Sum: ${ints.sum()}\n" +
  "Average: ${ints.average()}"

fun main() {
  evaluate(10, -3, 8, 1, 9) eq """
    Size: 5
    Sum: 25
```

```
    Average: 5.0
  """
}
```

You can pass an `Array` of elements wherever a `vararg` is accepted. To create an `Array`, use `arrayOf()` in the same way you use `listOf()`. An `Array` is always mutable. To convert an `Array` into a sequence of arguments (not just a single element of type `Array`), use the *spread operator*, `*`:

```
// Varargs/SpreadOperator.kt
import varargs.sum
import atomictest.eq

fun main() {
  val array = intArrayOf(4, 5)
  sum(1, 2, 3, *array, 6) eq 21  // [1]
  // Doesn't compile:
  // sum(1, 2, 3, array, 6)

  val list = listOf(9, 10, 11)
  sum(*list.toIntArray()) eq 30  // [2]
}
```

If you pass an `Array` of primitive types (like `Int`, `Double` or `Boolean`) as in the example above, the `Array` creation function must be specifically typed. If you use `arrayOf(4, 5)` instead of `intArrayOf(4, 5)`, line `[1]` will produce an error complaining that *inferred type is Array<Int> but IntArray was expected.*

The spread operator only works with arrays. If you have a `List` that you want to pass as a sequence of arguments, first convert it to an `Array` and then apply the spread operator, as in **[2]**. Because the result is an `Array` of a primitive type, we must again use the specific conversion function `toIntArray()`.

The spread operator is especially helpful when you must pass `vararg` arguments to another function that also expects `varargs`:

```
// Varargs/TwoFunctionsWithVarargs.kt
package varargs
import atomictest.eq

fun first(vararg numbers: Int): String {
  var result = ""
  for (i in numbers) {
    result += "[$i]"
  }
  return result
}

fun second(vararg numbers: Int) =
  first(*numbers)

fun main() {
  second(7, 9, 32) eq "[7][9][32]"
}
```

Command-Line Arguments

When invoking a program on the command line, you can pass it a variable number of arguments. To capture command-line arguments, you must provide a particular parameter to `main()`:

```
// Varargs/MainArgs.kt

fun main(args: Array<String>) {
  for (a in args) {
    println(a)
  }
}
```

The parameter is traditionally called `args` (although you can call it anything), and the type for `args` can only be `Array<String>` (`Array` of `String`).

If you are using IntelliJ IDEA, you can pass program arguments by editing the corresponding "Run configuration," as shown in the last exercise for this atom.

You can also use the `kotlinc` compiler to produce a command-line program. If `kotlinc` isn't on your computer, follow the instructions on the Kotlin main site[34]. Once you've entered and saved the code for `MainArgs.kt`, type the following at a command prompt:

```
kotlinc MainArgs.kt
```

You provide the command-line arguments following the program invocation, like this:

```
kotlin MainArgsKt hamster 42 3.14159
```

You'll see this output:

```
hamster
42
3.14159
```

If you want to turn a `String` parameter into a specific type, Kotlin provides conversion functions, such as a `toInt()` for converting to an `Int`, and `toFloat()` for converting to a `Float`. Using these assumes that the command-line arguments appear in a particular order. Here, the program expects a `String`, followed by something convertible to an `Int`, followed by something convertible to a `Float`:

```
// Varargs/MainArgConversion.kt

fun main(args: Array<String>) {
  if (args.size < 3) return
  val first = args[0]
  val second = args[1].toInt()
  val third = args[2].toFloat()
  println("$first  $second  $third")
}
```

The first line in `main()` quits the program if there aren't enough arguments. If you don't provide something convertible to an `Int` and a `Float` as the second and third command-line arguments, you will see runtime errors (try it to see the errors).

Compile and run `MainArgConversion.kt` with the same command-line arguments we used before, and you'll see:

```
hamster 42 3.14159
```

Exercises and solutions can be found at www.AtomicKotlin.com.

[34] https://kotlinlang.org/

Sets

A `Set` is a collection that allows only one element of each value.

The most common `Set` activity is to test for membership using `in` or `contains()`:

```
// Sets/Sets.kt
import atomictest.eq

fun main() {
  val intSet = setOf(1, 1, 2, 3, 9, 9, 4)
  // No duplicates:
  intSet eq setOf(1, 2, 3, 4, 9)

  // Element order is unimportant:
  setOf(1, 2) eq setOf(2, 1)

  // Set membership:
  (9 in intSet) eq true
  (99 in intSet) eq false

  intSet.contains(9) eq true
  intSet.contains(99) eq false

  // Does this set contain another set?
  intSet.containsAll(setOf(1, 9, 2)) eq true

  // Set union:
  intSet.union(setOf(3, 4, 5, 6)) eq
    setOf(1, 2, 3, 4, 5, 6, 9)

  // Set intersection:
  intSet intersect setOf(0, 1, 2, 7, 8) eq
    setOf(1, 2)

  // Set difference:
  intSet subtract setOf(0, 1, 9, 10) eq
```

```
    setOf(2, 3, 4)
  intSet - setOf(0, 1, 9, 10) eq
    setOf(2, 3, 4)
}
```

This example shows:

1. Placing duplicate items into a `Set` automatically removes those duplicates.
2. Element order is not important for sets. Two sets are equal if they contain the same elements.
3. Both `in` and `contains()` test for membership.
4. You can perform the usual Venn-diagram operations like checking for subset, union, intersection and difference, using either dot notation (`set.union(other)`) or infix notation (`set intersect other`). The functions `union`, `intersect` and `subtract` can be used with infix notation.
5. Set difference can be expressed with either `subtract()` or the minus operator.

To remove duplicates from a `List`, convert it to a `Set`:

```
// Sets/RemoveDuplicates.kt
import atomictest.eq

fun main() {
  val list = listOf(3, 3, 2, 1, 2)
  list.toSet() eq setOf(1, 2, 3)
  list.distinct() eq listOf(3, 2, 1)
  "abbcc".toSet() eq setOf('a', 'b', 'c')
}
```

You can also use `distinct()`, which returns a `List`. You may call `toSet()` on a `String` to convert it into a set of unique characters.

As with `List`, Kotlin provides two creation functions for `Set`. The result of `setOf()` is read-only. To create a mutable `Set`, use `mutableSetOf()`:

```
// Sets/MutableSet.kt
import atomictest.eq

fun main() {
  val mutableSet = mutableSetOf<Int>()
  mutableSet += 42
  mutableSet += 42
  mutableSet eq setOf(42)
  mutableSet -= 42
  mutableSet eq setOf<Int>()
}
```

The operators `+=` and `-=` add and remove elements to `Set`s, just as with `List`s.

Exercises and solutions can be found at www.AtomicKotlin.com.

Maps

A `Map` connects *keys* to *values* and looks up a value when given a key.

You create a `Map` by providing key-value pairs to `mapOf()`. Using `to`, we separate each key from its associated value:

```
// Maps/Maps.kt
import atomictest.eq

fun main() {
  val constants = mapOf(
    "Pi" to 3.141,
    "e" to 2.718,
    "phi" to 1.618
  )
  constants eq
    "{Pi=3.141, e=2.718, phi=1.618}"

  // Look up a value from a key:
  constants["e"] eq 2.718              // [1]
  constants.keys eq setOf("Pi", "e", "phi")
  constants.values eq "[3.141, 2.718, 1.618]"

  var s = ""
  // Iterate through key-value pairs:
  for (entry in constants) {           // [2]
    s += "${entry.key}=${entry.value}, "
  }
  s eq "Pi=3.141, e=2.718, phi=1.618,"

  s = ""
  // Unpack during iteration:
  for ((key, value) in constants)      // [3]
    s += "$key=$value, "
  s eq "Pi=3.141, e=2.718, phi=1.618,"
}
```

- **[1]** The `[]` operator looks up a value using a key. You can produce all the keys using `keys` and all the values using `values`. Calling `keys` produces a `Set` because all keys in a `Map` must be unique, otherwise you'd have ambiguity during a lookup.
- **[2]** Iterating through a `Map` produces key-value pairs as map entries.
- **[3]** You can unpack keys and values as you iterate.

A plain `Map` is read-only. Here's a `MutableMap`:

```
// Maps/MutableMaps.kt
import atomictest.eq

fun main() {
  val m =
    mutableMapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  m[5] = "5ive"
  m[5] eq "5ive"
  m += 4 to "four"
  m eq mapOf(5 to "5ive",
    4 to "four", 6 to "six")
}
```

`map[key] = value` adds or changes the `value` associated with `key`. You can also explicitly add a pair by saying `map += key to value`.

`mapOf()` and `mutableMapOf()` preserve the order in which the elements are put into the `Map`. This is not guaranteed for other types of `Map`.

A read-only `Map` doesn't allow mutations:

```
// Maps/ReadOnlyMaps.kt
import atomictest.eq

fun main() {
  val m = mapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  // m[5] = "5ive" // Fails
  // m += (4 to "four") // Fails
  m + (4 to "four") // Doesn't change m
  m eq mapOf(5 to "five", 6 to "six")
  val m2 = m + (4 to "four")
  m2 eq mapOf(
    5 to "five", 6 to "six", 4 to "four")
}
```

The definition of `m` creates a `Map` associating `Int`s with `String`s. If we try to replace a `String`, Kotlin emits an error.

An expression with `+` creates a new `Map` that includes both the old elements and the new one, but doesn't affect the original `Map`. The only way to "add" an element to a read-only `Map` is by creating a new `Map`.

A `Map` returns `null` if it doesn't contain an entry for a given key. If you need a result that can't be `null`, use `getValue()` and catch `NoSuchElementException` if the key is missing:

```
// Maps/GetValue.kt
import atomictest.*

fun main() {
  val map = mapOf('a' to "attempt")
  map['b'] eq null
  capture {
    map.getValue('b')
  } eq "NoSuchElementException: " +
    "Key b is missing in the map."
  map.getOrDefault('a', "??") eq "attempt"
  map.getOrDefault('b', "??") eq "??"
}
```

`getOrDefault()` is usually a nicer alternative to `null` or an exception.

You can store class instances as values in a `Map`. Here's a map that retrieves a `Contact` using a number `String`:

```
// Maps/ContactMap.kt
package maps
import atomictest.eq

class Contact(
  val name: String,
  val phone: String
) {
  override fun toString() =
    "Contact('$name', '$phone')"
}

fun main() {
  val miffy = Contact("Miffy", "1-234-567890")
  val cleo = Contact("Cleo", "098-765-4321")
  val contacts = mapOf(
    miffy.phone to miffy,
    cleo.phone to cleo)
  contacts["1-234-567890"] eq miffy
  contacts["1-111-111111"] eq null
}
```

It's possible to use class instances as keys in a `Map`, but that's trickier so we discuss it later in the book.

- -

`Maps` look like simple little databases. They are sometimes called *associative arrays*, because they associate keys with values. Although they are quite limited compared to a full-featured database, they are nonetheless remarkably useful (and far more efficient than a database).

Exercises and solutions can be found at www.AtomicKotlin.com.

Property Accessors

> To read a property, use its name. To assign a value to a mutable property, use the assignment operator =.

This reads and writes the property `i`:

```
// PropertyAccessors/Data.kt
package propertyaccessors
import atomictest.eq

class Data(var i: Int)

fun main() {
  val data = Data(10)
  data.i eq 10 // Read the 'i' property
  data.i = 20  // Write to the 'i' property
}
```

This appears to be straightforward access to the piece of storage named `i`. However, Kotlin calls functions to perform the read and write operations. As you expect, the default behavior of those functions reads and writes the data stored in `i`. In this atom you'll learn to write your own *property accessors* to customize the reading and writing actions.

The accessor used to get the value of a property is called a *getter*. You create a getter by defining `get()` immediately after the property definition. The accessor used to modify a mutable property is called a *setter*. You create a setter by defining `set()` immediately after the property definition.

The property accessors defined in the following example imitate the default implementations generated by Kotlin. We display additional information so you can see that the property accessors are indeed called during reads and writes. We indent `get()` and `set()` to visually associate them with the property, but the actual association happens because `get()` and `set()` are defined immediately after that property (Kotlin doesn't care about the indentation):

```
// PropertyAccessors/Default.kt
package propertyaccessors
import atomictest.*

class Default {
  var i: Int = 0
    get() {
      trace("get()")
      return field       // [1]
    }
    set(value) {
      trace("set($value)")
      field = value      // [2]
    }
}

fun main() {
  val d = Default()
  d.i = 2
  trace(d.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

The definition order for `get()` and `set()` is unimportant. You can define `get()` without defining `set()`, and vice-versa.

The default behavior for a property returns its stored value from a getter and modifies it with a setter—the actions of **[1]** and **[2]**. Inside the getter and setter, the stored value is manipulated indirectly using the `field` keyword, which is only accessible within these two functions.

This next example uses the default implementation of the getter and adds a setter to trace changes to the property `n`:

```
// PropertyAccessors/LogChanges.kt
package propertyaccessors
import atomictest.*

class LogChanges {
  var n: Int = 0
    set(value) {
      trace("$field becomes $value")
      field = value
    }
}

fun main() {
  val lc = LogChanges()
  lc.n eq 0
  lc.n = 2
  lc.n eq 2
  trace eq "0 becomes 2"
}
```

If you define a property as `private`, both accessors become `private`. You can also make the setter `private` and the getter `public`. Then you can read the property outside the class, but only change its value inside the class:

```
// PropertyAccessors/Counter.kt
package propertyaccessors
import atomictest.eq

class Counter {
  var value: Int = 0
    private set
  fun inc() = value++
}

fun main() {
  val counter = Counter()
  repeat(10) {
    counter.inc()
  }
  counter.value eq 10
}
```

Using `private set`, we control the `value` property so it can only be incremented by one.

Normal properties store their data in a field. You can also create a property that doesn't have a field:

```
// PropertyAccessors/Hamsters.kt
package propertyaccessors
import atomictest.eq

class Hamster(val name: String)

class Cage(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  val capacity: Int
    get() = maxCapacity - hamsters.size
  val full: Boolean
    get() = hamsters.size == maxCapacity
  fun put(hamster: Hamster): Boolean =
    if (full)
      false
    else {
      hamsters += hamster
      true
    }
  fun take(): Hamster =
    hamsters.removeAt(0)
}

fun main() {
  val cage = Cage(2)
  cage.full eq false
  cage.capacity eq 2
  cage.put(Hamster("Alice")) eq true
  cage.put(Hamster("Bob")) eq true
  cage.full eq true
  cage.capacity eq 0
  cage.put(Hamster("Charlie")) eq false
  cage.take()
  cage.capacity eq 1
}
```

The properties `capacity` and `full` contain no underlying state—they are computed at the time of each access. Both `capacity` and `full` are similar to functions, and you can define them as such:

```
// PropertyAccessors/Hamsters2.kt
package propertyaccessors

class Cage2(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  fun capacity(): Int =
    maxCapacity - hamsters.size
  fun isFull(): Boolean =
    hamsters.size == maxCapacity
}
```

In this case, using properties improves readability because capacity and fullness are properties of the cage. However, don't just convert all your functions to properties—first, see how they read.

- -

The Kotlin style guide prefers properties over functions when the value is cheap to calculate and the property returns the same result for each invocation as long as the object state hasn't changed.

Property accessors provide a kind of protection for properties. Many object-oriented languages rely on making a physical field `private` to control access to that property. With property accessors you can add code to control or modify that access, while allowing anyone to use a property.

Exercises and solutions can be found at www.AtomicKotlin.com.

Summary 2

> This atom summarizes and reviews the atoms in Section II, from Objects Everywhere through Property Accessors.

If you're an experienced programmer, this is your next atom after Summary 1, and you will go through the atoms sequentially after this.

New programmers should read this atom and perform the exercises as review. If any information here isn't clear to you, go back and study the atom for that topic.

The topics appear in appropriate order for experienced programmers, which is not the same as the order of the atoms in the book. For example, we start by introducing packages and imports so we can use our minimal test framework for the rest of the atom.

Packages & Testing

Any number of reusable library components can be bundled under a single library name using the `package` keyword:

```
// Summary2/ALibrary.kt
package com.yoururl.libraryname

// Components to reuse ...
fun f() = "result"
```

You can put multiple components in a single file, or spread components out among multiple files under the same package name. Here we've defined `f()` as the sole component.

To make it unique, the package name conventionally begins with your reversed domain name. In this example, the domain name is `yoururl.com`.

In Kotlin, the package name can be independent from the directory where its contents are located. Java requires that the directory structure correspond to the fully-qualified package name, so the package `com.yoururl.libraryname` should be located under the `com/yoururl/libraryname` directory. For mixed Kotlin and Java projects, Kotlin's style guide recommends the same practice. For pure Kotlin projects, put the directory `libraryname` at the top level of your project's directory structure.

An `import` statement brings one or more names into the current namespace:

```
// Summary2/UseALibrary.kt
import com.yoururl.libraryname.*

fun main() {
  val x = f()
}
```

The star after `libraryname` tells Kotlin to import all the components of a library. You can also select components individually; details are in Packages.

In the remainder of this book we use `package` statements for any file that defines functions, classes, etc., outside of `main()`. This prevents name clashes with other files in the book. We usually won't put a `package` statement in a file that *only* contains a `main()`.

An important library for this book is `atomictest`, our simple testing framework. `atomictest` is defined in Appendix A: AtomicTest, although it uses language features you will not understand at this point in the book.

After importing `atomictest`, you use `eq` (equals) and `neq` (not equals) almost as if they were language keywords:

```
// Summary2/UsingAtomicTest.kt
import atomictest.*

fun main() {
  val pi = 3.14
  val pie = "A round dessert"
  pi eq 3.14
  pie eq "A round dessert"
  pi neq pie
}
/* Output:
3.14
A round dessert
3.14
*/
```

The ability to use eq/neq without any dots or parentheses is called *infix notation*. You can call `infix` functions either in the regular way: `pi.eq(3.14)`, or using infix notation: `pi eq 3.14`. Both `eq` and `neq` are assertions of truth that also display the result from the left side of the eq/neq statement, and an error message if the expression on the right of the `eq` isn't equivalent to the left (or *is* equivalent, in the case of `neq`). This way you see verified results in the source code.

`atomictest.trace` uses function-call syntax for adding results, which can then be validated using `eq`:

```
// Testing/UsingTrace.kt
import atomictest.*

fun main() {
  trace("Hello,")
  trace(47)
  trace("World!")
  trace eq """
    Hello,
    47
    World!
  """
}
```

You can effectively replace `println()` with `trace()`.

Objects Everywhere

Kotlin is a *hybrid object-functional* language: it supports both object-oriented and functional programming paradigms.

Objects contain `val`s and `var`s to store data (these are called *properties*) and perform operations using functions defined within a class, called *member functions* (when it's unambiguous, we just say "functions"). A *class* defines properties and member functions for what is essentially a new, user-defined data type. When you create a `val` or `var` of a class, it's called *creating an object* or *creating an instance*.

An especially useful type of object is the *container*, also called *collection*. A container is an object that holds other objects. In this book, we often use the `List` because it's the most general-purpose sequence. Here we perform several operations on a `List` that holds `Doubles`. `listOf()` creates a new `List` from its arguments:

```
// Summary2/ListCollection.kt
import atomictest.eq

fun main() {
  val lst = listOf(19.2, 88.3, 22.1)
  lst[1] eq 88.3  // Indexing
  lst.reversed() eq listOf(22.1, 88.3, 19.2)
  lst.sorted() eq listOf(19.2, 22.1, 88.3)
  lst.sum() eq 129.6
}
```

No `import` statement is required to use a `List`.

Kotlin uses square brackets for indexing into sequences. Indexing is zero-based.

This example also shows a few of the many standard library functions available for `Lists`: `sorted()`, `reversed()`, and `sum()`. To understand these functions, consult the online Kotlin documentation[35].

When you call `sorted()` or `reversed()`, `lst` is not modified. Instead, a new `List` is created and returned, containing the desired result. This approach of never modifying the original object is consistent throughout Kotlin libraries and you should endeavor to follow this pattern when writing your own code.

[35] https://kotlinlang.org/docs/reference/

Creating Classes

A class definition consists of the `class` keyword, a name for the class, and an optional body. The body contains property definitions (`val`s and `var`s) and function definitions.

This example defines a `NoBody` class without a body, and classes with `val` properties:

```
// Summary2/ClassBodies.kt
package summary2

class NoBody

class SomeBody {
  val name = "Janet Doe"
}

class EveryBody {
  val all = listOf(SomeBody(),
    SomeBody(), SomeBody())
}

fun main() {
  val nb = NoBody()
  val sb = SomeBody()
  val eb = EveryBody()
}
```

To create an instance of a class, put parentheses after its name, along with arguments if those are required.

Properties within class bodies can be any type. `SomeBody` contains a property of type `String`, and `EveryBody`'s property is a `List` holding `SomeBody` objects.

Here's a class with member functions:

```
// Summary2/Temperature.kt
package summary2
import atomictest.eq

class Temperature {
  var current = 0.0
  var scale = "f"
  fun setFahrenheit(now: Double) {
    current = now
    scale = "f"
  }
  fun setCelsius(now: Double) {
    current = now
    scale = "c"
  }
  fun getFahrenheit(): Double =
    if (scale == "f")
      current
    else
      current * 9.0 / 5.0 + 32.0
  fun getCelsius(): Double =
    if (scale == "c")
      current
    else
      (current - 32.0) * 5.0 / 9.0
}

fun main() {
  val temp = Temperature()   // [1]
  temp.setFahrenheit(98.6)
  temp.getFahrenheit() eq 98.6
  temp.getCelsius() eq 37.0
  temp.setCelsius(100.0)
  temp.getFahrenheit() eq 212.0
}
```

These member functions are just like the top-level functions we've defined *outside* of classes, except they belong to the class and have unqualified access to the other members of the class, such as `current` and `scale`. Member functions can also call other member functions in the same class without qualification.

- **[1]** Although `temp` is a `val`, we later modify the `Temperature` object. The `val`

definition prevents the reference `temp` from being reassigned to a new object, but it does not restrict the behavior of the object itself.

The following two classes are the foundation of a tic-tac-toe game:

```
// Summary2/TicTacToe.kt
package summary2
import atomictest.eq

class Cell {
  var entry = ' '                    // [1]
  fun setValue(e: Char): String =    // [2]
    if (entry == ' ' &&
      (e == 'X' || e == 'O')) {
      entry = e
      "Successful move"
    } else
      "Invalid move"
}

class Grid {
  val cells = listOf(
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell())
  )
  fun play(e: Char, x: Int, y: Int): String =
    if (x !in 0..2 || y !in 0..2)
      "Invalid move"
    else
      cells[x][y].setValue(e)        // [3]
}

fun main() {
  val grid = Grid()
  grid.play('X', 1, 1) eq "Successful move"
  grid.play('X', 1, 1) eq "Invalid move"
  grid.play('O', 1, 3) eq "Invalid move"
}
```

The `Grid` class holds a `List` containing three `Lists`, each containing three `Cells`—a matrix.

- **[1]** The `entry` property in `Cell` is a `var` so it can be modified. The single quotes in the initialization produce a `Char` type, so all assignments to `entry` must also be `Char`s.
- **[2]** `setValue()` tests that the `Cell` is available and that you've passed the correct character. It returns a `String` result to indicate success or failure.
- **[3]** `play()` checks to see if the `x` and `y` arguments are within range, then indexes into the matrix, relying on the tests performed by `setValue()`.

Constructors

Constructors create new objects. You pass information to a constructor using its parameter list, placed in parentheses directly after the class name. A constructor call thus looks like a function call, except that the initial letter of the name is capitalized (following the Kotlin style guide). The constructor returns an object of the class:

```
// Summary2/WildAnimals.kt
package summary2
import atomictest.eq

class Badger(id: String, years: Int) {
  val name = id
  val age = years
  override fun toString() =
    "Badger: $name, age: $age"
}

class Snake(
  var type: String,
  var length: Double
) {
  override fun toString() =
    "Snake: $type, length: $length"
}

class Moose(
  val age: Int,
  val height: Double
) {
```

```
  override fun toString() =
    "Moose, age: $age, height: $height"
}

fun main() {
  Badger("Bob", 11) eq "Badger: Bob, age: 11"
  Snake("Garden", 2.4) eq
    "Snake: Garden, length: 2.4"
  Moose(16, 7.2) eq
    "Moose, age: 16, height: 7.2"
}
```

The parameters `id` and `years` in `Badger` are only available in the *constructor body*. The constructor body consists of the lines of code other than function definitions; in this case, the definitions for `name` and `age`.

Often you want the constructor parameters to be available in parts of the class other than the constructor body, but without the trouble of explicitly defining new identifiers as we did with `name` and `age`. If you define your parameters as `var`s or `val`s, they becomes properties and are accessible everywhere in the class. Both `Snake` and `Moose` use this approach, and you can see that the constructor parameters are now available inside their respective `toString()` functions.

Constructor parameters declared with `val` cannot be changed, but those declared with `var` can.

Whenever you use an object in a situation that expects a `String`, Kotlin produces a `String` representation of that object by calling its `toString()` member function. To define a `toString()`, you must understand a new keyword: `override`. This is necessary (Kotlin insists on it) because `toString()` is already defined. `override` tells Kotlin that we do actually want to replace the default `toString()` with our own definition. The explicitness of `override` makes this clear to the reader and helps prevent mistakes.

Notice the formatting of the multiline parameter list for `Snake` and `Moose`—this is the recommended standard when you have too many parameters to fit on one line, for both constructors and functions.

Constraining Visibility

Kotlin provides *access modifiers* similar to those available in other languages like C++ or Java. These allow component creators to decide what is available to the client programmer. Kotlin's access modifiers include the `public`, `private`, `protected`, and `internal` keywords. `protected` is explained later.

An access modifier like `public` or `private` appears before the definition for a class, function or property. Each access modifier only controls the access for that particular definition.

A `public` definition is available to everyone, in particular to the client programmer who uses that component. Thus, any changes to a `public` definition will impact client code.

If you don't provide a modifier, your definition is automatically `public`. For clarity in certain cases, programmers still sometimes redundantly specify `public`.

If you define a class, top-level function, or property as `private`, it is available only within that file:

```
// Summary2/Boxes.kt
package summary2
import atomictest.*

private var count = 0                    // [1]

private class Box(val dimension: Int) { // [2]
  fun volume() =
    dimension * dimension * dimension
  override fun toString() =
    "Box volume: ${volume()}"
}

private fun countBox(box: Box) {         // [3]
  trace("$box")
  count++
}

fun countBoxes() {
  countBox(Box(4))
```

```
  countBox(Box(5))
}

fun main() {
  countBoxes()
  trace("$count boxes")
  trace eq """
    Box volume: 64
    Box volume: 125
    2 boxes
  """
}
```

You can access `private` properties (**[1]**), classes (**[2]**), and functions (**[3]**) only from other functions and classes in the `Boxes.kt` file. Kotlin prevents you from accessing `private` top-level elements from another file.

Class members can be `private`:

```
// Summary2/JetPack.kt
package summary2
import atomictest.eq

class JetPack(
  private var fuel: Double   // [1]
) {
  private var warning = false
  private fun burn() =       // [2]
    if (fuel - 1 <= 0) {
      fuel = 0.0
      warning = true
    } else
      fuel -= 1
  public fun fly() = burn()  // [3]
  fun check() =              // [4]
    if (warning)             // [5]
      "Warning"
    else
      "OK"
}

fun main() {
```

```
  val jetPack = JetPack(3.0)
  while (jetPack.check() != "Warning") {
    jetPack.check() eq "OK"
    jetPack.fly()
  }
  jetPack.check() eq "Warning"
}
```

- **[1]** `fuel` and `warning` are both `private` properties and can't be used by non-members of `JetPack`.
- **[2]** `burn()` is `private`, and thus only accessible inside `JetPack`.
- **[3]** `fly()` and `check()` are `public` and can be used everywhere.
- **[4]** No access modifier means `public` visibility.
- **[5]** Only members of the same class can access `private` members.

Because a `private` definition is *not* available to everyone, you can generally change it without concern for the client programmer. As a library designer, you'll typically keep everything as `private` as possible, and expose only functions and classes you want client programmers to use. To limit the size and complexity of example listings in this book, we only use `private` in special cases.

Any function you're certain is only a *helper function* can be made `private`, to ensure you don't accidentally use it elsewhere and thus prohibit yourself from changing or removing the function.

It can be useful to divide large programs into *modules*. A module is a logically independent part of a codebase. An `internal` definition is accessible only inside the module where it is defined. The way you divide a project into modules depends on the build system (such as Gradle[36] or Maven[37]) and is beyond the scope of this book.

Modules are a higher-level concept, while *packages* enable finer-grained structuring.

Exceptions

Consider `toDouble()`, which converts a `String` to a `Double`. What happens if you call it for a `String` that doesn't translate into a `Double`?

[36] https://gradle.org/
[37] https://maven.apache.org/

```
// Summary2/ToDoubleException.kt

fun main() {
  // val i = "$1.9".toDouble()
}
```

Uncommenting the line in `main()` produces an exception. Here, the failing line is commented so we don't stop the book's build (which checks whether each example compiles and runs as expected).

When an exception is thrown, the current path of execution stops, and the exception object ejects from the current context. When an exception isn't caught, the program aborts and displays a *stack trace* containing detailed information.

To avoid displaying exceptions by commenting and uncommenting code, `atomictest.capture()` stores the exception and compares it to what we expect:

```
// Summary2/AtomicTestCapture.kt
import atomictest.*

fun main() {
  capture {
    "$1.9".toDouble()
  } eq "NumberFormatException: " +
    """For input string: "$1.9""""
}
```

`capture()` is designed specifically for this book, so you can see the exception and know that the output has been checked by the book's build system.

Another strategy when your function can't successfully produce the expected result is to return `null`. Later in Nullable Types we discuss how `null` affects the type of the resulting expression.

To throw an exception, use the `throw` keyword followed by the exception you want to throw, along with any arguments it might need. `quadraticZeroes()` in the following example solves the quadratic equation[38] that defines a parabola:

$$ax^2 + bx + c = 0$$

The solution is the *quadratic formula*:

[38] https://en.wikipedia.org/wiki/Quadratic_formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The Quadratic Formula

The example finds the *zeroes* of the parabola, where the lines cross the x-axis. We throw exceptions for two limitations:

1. `a` cannot be zero.
2. For zeroes to exist, $\mathbf{b^2 - 4ac}$ cannot be negative.

If zeroes exist, there are two of them, so we create the `Roots` class to hold the return values:

```
// Summary2/Quadratic.kt
package summary2
import kotlin.math.sqrt
import atomictest.*

class Roots(
  val root1: Double,
  val root2: Double
)

fun quadraticZeroes(
  a: Double,
  b: Double,
  c: Double
): Roots {
  if (a == 0.0)
    throw IllegalArgumentException(
      "a is zero")
  val underRadical = b * b - 4 * a * c
  if (underRadical < 0)
    throw IllegalArgumentException(
      "Negative underRadical: $underRadical")
  val squareRoot = sqrt(underRadical)
  val root1 = (-b - squareRoot) / (2 * a)
  val root2 = (-b + squareRoot) / (2 * a)
  return Roots(root1, root2)
}
```

```
fun main() {
  capture {
    quadraticZeroes(0.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
    "a is zero"
  capture {
    quadraticZeroes(3.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
    "Negative underRadical: -44.0"
  val roots = quadraticZeroes(1.0, 2.0, -8.0)
  roots.root1 eq -4.0
  roots.root2 eq 2.0
}
```

Here we use the standard exception class `IllegalArgumentException`. Later you'll learn to define your own exception types and to make them specific to your circumstances. Your goal is to generate the most useful messages possible, to simplify the support of your application in the future.

Lists

`Lists` are Kotlin's basic sequential container type. You create a read-only list using `listOf()` and a mutable list using `mutableListOf()`:

```
// Summary2/ReadonlyVsMutableList.kt
import atomictest.*

fun main() {
  val ints = listOf(5, 13, 9)
  // ints.add(11) // 'add()' not available
  for (i in ints) {
    if (i > 10) {
      trace(i)
    }
  }
  val chars = mutableListOf('a', 'b', 'c')
  chars.add('d') // 'add()' available
  chars += 'e'
```

```
  trace(chars)
  trace eq """
    13
    [a, b, c, d, e]
  """
}
```

A basic `List` is read-only, and does not include modification functions. Thus, the modification function `add()` doesn't work with `ints`.

`for` loops work well with `Lists`: `for(i in ints)` means `i` gets each value in `ints`.

`chars` is created as a `MutableList`; it can be modified using functions like `add()` or `remove()`. You can also use `+=` and `-=` to add or remove elements.

A read-only `List` is not the same as an *immutable* `List`, which can't be modified at all. Here, we assign `first`, a mutable `List`, to `second`, a read-only `List` reference. The read-only characteristic of `second` doesn't prevent the `List` from changing via `first`:

```
// Summary2/MultipleListReferences.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first += 2
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first` and `second` refer to the same object in memory. We mutate the `List` via the `first` reference, and then observe this change in the `second` reference.

Here's a `List` of `Strings` created by breaking up a triple-quoted paragraph. This shows the power of some of the standard library functions. Notice how those functions can be chained:

```
// Summary2/ListOfStrings.kt
import atomictest.*

fun main() {
  val wocky = """
    Twas brillig, and the slithy toves
      Did gyre and gimble in the wabe:
    All mimsy were the borogoves,
      And the mome raths outgrabe.
  """.trim().split(Regex("\\W+"))
  trace(wocky.take(5))
  trace(wocky.slice(6..12))
  trace(wocky.slice(6..18 step 2))
  trace(wocky.sorted().takeLast(5))
  trace(wocky.sorted().distinct().takeLast(5))
  trace eq """
    [Twas, brillig, and, the, slithy]
    [Did, gyre, and, gimble, in, the, wabe]
    [Did, and, in, wabe, mimsy, the, And]
    [the, the, toves, wabe, were]
    [slithy, the, toves, wabe, were]
  """
}
```

`trim()` produces a new `String` with the leading and trailing whitespace characters (including newlines) removed. `split()` divides the `String` according to its argument. In this case we use a `Regex` object, which creates a *regular expression*—a pattern that matches the parts to split. `\W` is a special pattern that means "not a word character," and `+` means "one or more of the preceding." Thus `split()` will break at one or more non-word characters, and so divides the block of text into its component words.

In a `String` literal, `\` precedes a special character and produces, for example, a newline character (`\n`), or a tab (`\t`). To produce an actual `\` in the resulting String you need two backslashes: `"\\"`. Thus all regular expressions require an extra `\` to insert a backslash, unless you use a triple-quoted `String`: `"""\W+"""`.

`take(n)` produces a new `List` containing the first `n` elements. `slice()` produces a new `List` containing the elements selected by its `Range` argument, and this `Range` can include a `step`.

Note the name `sorted()` instead of `sort()`. When you call `sorted()` it *produces* a sorted `List`, leaving the original `List` alone. `sort()` only works with a `MutableList`, and that list is *sorted in place*—the original `List` is modified.

As the name implies, `takeLast(n)` produces a new `List` of the last `n` elements. You can see from the output that "the" is duplicated. This is eliminated by adding the `distinct()` function to the call chain.

Parameterized Types

Type parameters allow us to describe compound types, most commonly containers. In particular, type parameters specify what a container holds. Here, we tell Kotlin that `numbers` contain a `List` of `Int`, while `strings` contain a `List` of `String`:

```
// Summary2/ExplicitTyping.kt
package summary2
import atomictest.eq

fun main() {
  val numbers: List<Int> = listOf(1, 2, 3)
  val strings: List<String> =
    listOf("one", "two", "three")
  numbers eq "[1, 2, 3]"
  strings eq "[one, two, three]"
  toCharList("seven") eq "[s, e, v, e, n]"
}

fun toCharList(s: String): List<Char> =
  s.toList()
```

For both the `numbers` and `strings` definitions, we add colons and the type declarations `List<Int>` and `List<String>`. The angle brackets denote a *type parameter*, allowing us to say, "the container holds 'parameter' objects." You typically pronounce `List<Int>` as "`List` of `Int`."

A return value can also have a type parameter, as seen in `toCharList()`. You can't just say it returns a `List`—Kotlin complains, so you must give the type parameter as well.

Variable Argument Lists

The `vararg` keyword is short for *variable argument list*, and allows a function to accept any number of arguments (including zero) of the specified type. The `vararg` becomes an `Array`, which is similar to a `List`:

```
// Summary2/VarArgs.kt
package summary2
import atomictest.*

fun varargs(s: String, vararg ints: Int) {
  for (i in ints) {
    trace("$i")
  }
  trace(s)
}

fun main() {
  varargs("primes", 5, 7, 11, 13, 17, 19, 23)
  trace eq "5 7 11 13 17 19 23 primes"
}
```

A function definition may specify only one parameter as `vararg`. Any parameter in the list can be the `vararg`, but the final one is generally simplest.

You can pass an `Array` of elements wherever a `vararg` is accepted. To create an `Array`, use `arrayOf()` in the same way you use `listOf()`. An `Array` is always mutable. To convert an `Array` into a sequence of arguments (not just a single element of type `Array`), use the *spread operator* `*`:

```
// Summary2/ArraySpread.kt
import summary2.varargs
import atomictest.trace

fun main() {
  val array = intArrayOf(4, 5)      // [1]
  varargs("x", 1, 2, 3, *array, 6)  // [2]
  val list = listOf(9, 10, 11)
  varargs(
    "y", 7, 8, *list.toIntArray())  // [3]
  trace eq "1 2 3 4 5 6 x 7 8 9 10 11 y"
}
```

If you pass an `Array` of primitive types as in the example above, the `Array` creation function must be specifically typed. If **[1]** uses `arrayOf(4, 5)` instead of `intArrayOf(4, 5)`, **[2]** produces an error: *inferred type is Array<Int> but IntArray was expected.*

The spread operator only works with arrays. If you have a `List` to pass as a sequence of arguments, first convert it to an `Array` and then apply the spread operator, as in **[3]**. Because the result is an `Array` of a primitive type, we must use the specific conversion function `toIntArray()`.

Sets

`Sets` are collections that allow only one element of each value. A `Set` automatically prevents duplicates.

```
// Summary2/ColorSet.kt
package summary2
import atomictest.eq

val colors =
  "Yellow Green Green Blue"
    .split(Regex("""\W+""")).sorted()  // [1]

fun main() {
  colors eq
    listOf("Blue", "Green", "Green", "Yellow")
```

```
  val colorSet = colors.toSet()        // [2]
  colorSet eq
    setOf("Yellow", "Green", "Blue")
  (colorSet + colorSet) eq colorSet    // [3]
  val mSet = colorSet.toMutableSet()   // [4]
  mSet -= "Blue"
  mSet += "Red"                        // [5]
  mSet eq
    setOf("Yellow", "Green", "Red")
  // Set membership:
  ("Green" in colorSet) eq true        // [6]
  colorSet.contains("Red") eq false
}
```

- **[1]** The `String` is `split()` using a regular expression as described earlier for `ListOfStrings.kt`.
- **[2]** When `colors` is copied into the read-only `Set colorSet`, one of the two `"Green" Strings` is removed, because it is a duplicate.
- **[3]** Here we create and display a new `Set` using the `+` operator. Placing duplicate items into a `Set` automatically removes those duplicates.
- **[4]** `toMutableSet()` produces a new `MutableSet` from a read-only `Set`.
- **[5]** For a `MutableSet`, the operators `+=` and `-=` add and remove elements, as they do with `MutableLists`.
- **[6]** Test for `Set` membership using `in` or `contains()`

The normal mathematical set operations such as union, intersection, difference, etc., are all available.

Maps

A `Map` connects *keys* to *values* and looks up a value when given a key. You create a `Map` by providing key-value pairs to `mapOf()`. Using `to`, we separate each key from its associated value:

```
// Summary2/ASCIIMap.kt
import atomictest.eq

fun main() {
  val ascii = mapOf(
    "A" to 65,
    "B" to 66,
    "C" to 67,
    "I" to 73,
    "J" to 74,
    "K" to 75
  )
  ascii eq
    "{A=65, B=66, C=67, I=73, J=74, K=75}"
  ascii["B"] eq 66                    // [1]
  ascii.keys eq "[A, B, C, I, J, K]"
  ascii.values eq
    "[65, 66, 67, 73, 74, 75]"
  var kv = ""
  for (entry in ascii) {              // [2]
    kv += "${entry.key}:${entry.value},"
  }
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  kv = ""
  for ((key, value) in ascii)         // [3]
    kv += "$key:$value,"
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  val mutable = ascii.toMutableMap() // [4]
  mutable.remove("I")
  mutable eq
    "{A=65, B=66, C=67, J=74, K=75}"
  mutable.put("Z", 90)
  mutable eq
    "{A=65, B=66, C=67, J=74, K=75, Z=90}"
  mutable.clear()
  mutable["A"] = 100
  mutable eq "{A=100}"
}
```

- **[1]** A key (`"B"`) is used to look up a value with the `[]` operator. You can produce all the keys using `keys` and all the values using `values`. Accessing `keys`

produces a `Set` because all keys in a `Map` must already be unique (otherwise you'd have ambiguity during a lookup).
- **[2]** Iterating through a `Map` produces key-value pairs as map entries.
- **[3]** You can unpack key-value pairs as you iterate.
- **[4]** You can create a `MutableMap` from a read-only `Map` using `toMutableMap()`. Now we can perform operations that modify `mutable`, such as `remove()`, `put()`, and `clear()`. Square brackets can assign a new key-value pair into `mutable`. You can also add a pair by saying `map += key to value`.

Property Accessors

Accessing the property `i` appears straightforward:

```
// Summary2/PropertyReadWrite.kt
package summary2
import atomictest.eq

class Holder(var i: Int)

fun main() {
  val holder = Holder(10)
  holder.i eq 10 // Read the 'i' property
  holder.i = 20  // Write to the 'i' property
}
```

However, Kotlin calls functions to perform the read and write operations. The default behavior of those functions is to read and write the data stored in `i`. By creating *property accessors*, you change the actions that occur during reading and writing.

The accessor used to fetch the value of a property is called a *getter*. To create your own getter, define `get()` immediately after the property declaration. The accessor used to modify a mutable property is called a *setter*. To create your own setter, define `set()` immediately after the property declaration. The order of definition of getters and setters is unimportant, and you can define one without the other.

The property accessors in the following example imitate the default implementations while displaying additional information so you can see that the property accessors are indeed called during reads and writes. We indent the `get()` and `set()` functions to

visually associate them with the property, but the actual association happens because they are defined immediately after that property:

```
// Summary2/GetterAndSetter.kt
package summary2
import atomictest.*

class GetterAndSetter {
  var i: Int = 0
    get() {
      trace("get()")
      return field
    }
    set(value) {
      trace("set($value)")
      field = value
    }
}

fun main() {
  val gs = GetterAndSetter()
  gs.i = 2
  trace(gs.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

Inside the getter and setter, the stored value is manipulated indirectly using the `field` keyword, which is only accessible within these two functions. You can also create a property that doesn't have a `field`, but simply calls the getter to produce a result.

If you declare a `private` property, both accessors become `private`. You can make the setter `private` and the getter `public`. This means you can read the property outside the class, but only change its value inside the class.

Exercises and solutions can be found at www.AtomicKotlin.com.

Section III: Usability

Computer languages differ not so much in what they make possible, but in what they make easy—**Larry Wall**, inventor of the Perl language

Extension Functions

> Suppose you discover a library that does everything you need … almost. If it only had one or two additional member functions, it would solve your problem perfectly.

But it's not your code—either you don't have access to the source code or you don't control it. You'd have to repeat your modifications every time a new version came out.

Kotlin's *extension functions* effectively add member functions to existing classes. The type you extend is called the *receiver*. To define an extension function, you precede the function name with the receiver type:

```
fun ReceiverType.extensionFunction() { ... }
```

This adds two extension functions to the `String` class:

```
// ExtensionFunctions/Quoting.kt
package extensionfunctions
import atomictest.eq

fun String.singleQuote() = "'$this'"
fun String.doubleQuote() = "\"$this\""

fun main() {
  "Hi".singleQuote() eq "'Hi'"
  "Hi".doubleQuote() eq "\"Hi\""
}
```

You call extension functions as if they were members of the class.

To use extensions from another package, you must import them:

```
// ExtensionFunctions/Quote.kt
package other
import atomictest.eq
import extensionfunctions.doubleQuote
import extensionfunctions.singleQuote

fun main() {
  "Single".singleQuote() eq "'Single'"
  "Double".doubleQuote() eq "\"Double\""
}
```

You can access member functions or other extensions using the `this` keyword. `this` can also be omitted in the same way it can be omitted inside a class, so you don't need explicit qualification:

```
// ExtensionFunctions/StrangeQuote.kt
package extensionfunctions
import atomictest.eq

// Apply two sets of single quotes:
fun String.strangeQuote() =
  this.singleQuote().singleQuote()  // [1]

fun String.tooManyQuotes() =
  doubleQuote().doubleQuote()       // [2]

fun main() {
  "Hi".strangeQuote() eq "''Hi''"
  "Hi".tooManyQuotes() eq "\"\"Hi\"\""
}
```

- **[1]** `this` refers to the `String` receiver.
- **[2]** We omit the receiver object (`this`) of the first `doubleQuote()` function call.

Creating extensions to your own classes can sometimes produce simpler code:

```
// ExtensionFunctions/BookExtensions.kt
package extensionfunctions
import atomictest.eq

class Book(val title: String)

fun Book.categorize(category: String) =
  """title: "$title", category: $category"""

fun main() {
  Book("Dracula").categorize("Vampire") eq
    """title: "Dracula", category: Vampire"""
}
```

Inside `categorize()`, we access the `title` property without explicit qualification.

- -

Extension functions can only access `public` elements of the type being extended. Thus, extensions can only perform the same actions as regular functions. You can rewrite `Book.categorize(String)` as `categorize(Book, String)`. The only reason for using an extension function is the syntax, but this syntax sugar is powerful. To the calling code, extensions look the same as member functions, and IDEs show extensions when listing the functions that you can call for an object.

Exercises and solutions can be found at www.AtomicKotlin.com.

Named & Default Arguments

> You can provide argument names during a function call.

Named arguments improve code readability. This is especially true for long and complex argument lists—named arguments can be clear enough that the reader can understand a function call without looking at the documentation.

In this example, all parameters are `Int`. Named arguments clarify their meaning:

```
// NamedAndDefaultArgs/NamedArguments.kt
package color1
import atomictest.eq

fun color(red: Int, green: Int, blue: Int) =
  "($red, $green, $blue)"

fun main() {
  color(1, 2, 3) eq "(1, 2, 3)"   // [1]
  color(
    red = 76,                     // [2]
    green = 89,
    blue = 0
  ) eq "(76, 89, 0)"
  color(52, 34, blue = 0) eq      // [3]
    "(52, 34, 0)"
}
```

- **[1]** This doesn't tell you much. You'll have to look at the documentation to know what the arguments mean.
- **[2]** The meaning of every argument is clear.
- **[3]** You aren't required to name all arguments.

Named arguments allow you to change the order of the colors. Here, we specify `blue` first:

```
// NamedAndDefaultArgs/ArgumentOrder.kt
import color1.color
import atomictest.eq

fun main() {
  color(blue = 0, red = 99, green = 52) eq
    "(99, 52, 0)"
  color(red = 255, 255, 0) eq
    "(255, 255, 0)"
}
```

You can mix named and regular (positional) arguments. If you change argument order, you should use named arguments throughout the call—not only for readability, but the compiler often needs to be told where the arguments are.

Named arguments are even more useful when combined with *default arguments*, which are default values for arguments, specified in the function definition:

```
// NamedAndDefaultArgs/Color2.kt
package color2
import atomictest.eq

fun color(
  red: Int = 0,
  green: Int = 0,
  blue: Int = 0,
) = "($red, $green, $blue)"

fun main() {
  color(139) eq "(139, 0, 0)"
  color(blue = 139) eq "(0, 0, 139)"
  color(255, 165) eq "(255, 165, 0)"
  color(red = 128, blue = 128) eq
    "(128, 0, 128)"
}
```

Any argument you don't provide gets its default value, so you only need to provide arguments that differ from the defaults. If you have a long argument list, this simplifies the resulting code, making it easier to write and—more importantly—to read.

This example also uses a *trailing comma* in the definition for `color()`. The trailing comma is the extra comma after the last parameter (`blue`). This is useful when your parameters or values are written on multiple lines. With a trailing comma, you can add new items and change their order without adding or removing commas.

Named and default arguments (as well as trailing commas) also work for constructors:

```
// NamedAndDefaultArgs/Color3.kt
package color3
import atomictest.eq

class Color(
  val red: Int = 0,
  val green: Int = 0,
  val blue: Int = 0,
) {
  override fun toString() =
    "($red, $green, $blue)"
}

fun main() {
  Color(red = 77).toString() eq "(77, 0, 0)"
}
```

`joinToString()` is a standard library function that uses default arguments. It combines the contents of an iterable (a list, set or range) into a `String`. You can specify a separator, a prefix element and a postfix element:

```
// NamedAndDefaultArgs/CreateString.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3,)
  list.toString() eq "[1, 2, 3]"
  list.joinToString() eq "1, 2, 3"
  list.joinToString(prefix = "(",
    postfix = ")") eq "(1, 2, 3)"
  list.joinToString(separator = ":") eq
    "1:2:3"
}
```

The default `toString()` for a `List` returns the contents in square brackets, which might not be what you want. The default values for `joinToString()`s parameters are a comma for `separator` and empty `Strings` for `prefix` and `postfix`. In the above example, we use named and default arguments to specify only the arguments we want to change.

The initializer for `list` includes a trailing comma. Normally you'll only use a trailing comma when each element is on its own line.

If you use an object as a default argument, a new instance of that object is created for each invocation:

If you pass an object instance as a default argument (`da` within `g()` in the following example), that same instance is used for each call to `g()`. If you pass the syntax for a constructor call (`DefaultArg()` within `h()`), that constructor is called every time you call `h()`:

```
// NamedAndDefaultArgs/Evaluation.kt
package namedanddefault

class DefaultArg
val da = DefaultArg()

fun g(d: DefaultArg = da) = println(d)

fun h(d: DefaultArg = DefaultArg()) =
  println(d)

fun main() {
  g()
  g()
  h()
  h()
}
/* Sample output:
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@49476842
namedanddefault.DefaultArg@78308db1
*/
```

The output of the two `g()` calls shows identical object addresses. For the two calls to

`h()`, the addresses of the `DefaultArg` objects are different, showing that there are two distinct objects.

Specify argument names when they improve readability. Compare the following two calls to `joinToString()`:

```
// NamedAndDefaultArgs/CreateString2.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3)
  list.joinToString(". ", "", "!") eq
    "1. 2. 3!"
  list.joinToString(separator = ". ",
    postfix = "!") eq "1. 2. 3!"
}
```

It's hard to guess whether `". "` or `""` is a separator unless you memorize the parameter order, which is impractical.

As another example of default arguments, `trimMargin()` is a standard library function that formats multi-line `String`s. It uses a margin prefix `String` to establish the beginning of each line. `trimMargin()` trims leading whitespace characters followed by the margin prefix from every line of the source `String`. It removes the first and last lines if they are blank:

```
// NamedAndDefaultArgs/TrimMargin.kt
import atomictest.eq

fun main() {
  val poem = """
    |->Last night I saw upon the stair
        |->A little man who wasn't there
          |->He wasn't there again today
|->Oh, how I wish he'd go away."""
  poem.trimMargin() eq
"""->Last night I saw upon the stair
->A little man who wasn't there
->He wasn't there again today
->Oh, how I wish he'd go away."""
  poem.trimMargin(marginPrefix = "|->") eq
"""Last night I saw upon the stair
```

```
A little man who wasn't there
He wasn't there again today
Oh, how I wish he'd go away."""
}
```

The | ("pipe") is the default argument for the margin prefix, and you can replace it with a `String` of your choosing.

Exercises and solutions can be found at www.AtomicKotlin.com.

Overloading

> Languages without support for default arguments often use overloading to imitate that feature.

The term *overload* refers to the name of a function: You use the same name ("overload" that name) for different functions as long as the parameter lists differ. Here, we overload the member function `f()`:

```
// Overloading/Overloading.kt
package overloading
import atomictest.eq

class Overloading {
  fun f() = 0
  fun f(n: Int) = n + 2
}

fun main() {
  val o = Overloading()
  o.f() eq 0
  o.f(11) eq 13
}
```

In `Overloading`, you see two functions with the same name, `f()`. The function's *signature* consists of the name, parameter list and return type. Kotlin distinguishes one function from another by comparing signatures. When overloading functions, the parameter lists must be unique—you cannot overload on return types.

The calls show that they are indeed different functions. A function signature also includes information about the enclosing class (or the receiver type, if it's an extension function).

If a class already has a member function with the same signature as an extension function, Kotlin prefers the member function. However, you can overload the member function with an extension function:

```
// Overloading/MemberVsExtension.kt
package overloading
import atomictest.eq

class My {
  fun foo() = 0
}

fun My.foo() = 1              // [1]

fun My.foo(i: Int) = i + 2    // [2]

fun main() {
  My().foo() eq 0
  My().foo(1) eq 3
}
```

- **[1]** It's senseless to declare an extension that duplicates a member, because it can never be called.
- **[2]** You can overload a member function using an extension function by providing a different parameter list.

Don't use overloading to imitate default arguments. That is, don't do this:

```
// Overloading/WithoutDefaultArguments.kt
package withoutdefaultarguments
import atomictest.eq

fun f(n: Int) = n + 373
fun f() = f(0)

fun main() {
  f() eq 373
}
```

The function without parameters just calls the first function. The two functions can be replaced with a single function by using a default argument:

```
// Overloading/WithDefaultArguments.kt
package withdefaultarguments
import atomictest.eq

fun f(n: Int = 0) = n + 373

fun main() {
  f() eq 373
}
```

In both examples you can call the function either without an argument or by passing an integer value. Prefer the form in `WithDefaultArguments.kt`.

When using overloaded functions together with default arguments, calling the overloaded function searches for the "closest" match. In the following example, the `foo()` call in `main()` does *not* call the first version of the function using its default argument of 99, but instead calls the second version, the one without parameters:

```
// Overloading/OverloadedVsDefaultArg.kt
package overloadingvsdefaultargs
import atomictest.*

fun foo(n: Int = 99) = trace("foo-1-$n")

fun foo() {
  trace("foo-2")
  foo(14)
}

fun main() {
  foo()
  trace eq """
    foo-2
    foo-1-14
  """
}
```

You can never utilize the default argument of 99, because `foo()` always calls the second version of `f()`.

Why is overloading useful? It allows you to express "variations on a theme" more clearly than if you were forced to use different function names. Suppose you want addition functions:

```
// Overloading/OverloadingAdd.kt
package overloading
import atomictest.eq

fun addInt(i: Int, j: Int) = i + j
fun addDouble(i: Double, j: Double) = i + j

fun add(i: Int, j: Int) = i + j
fun add(i: Double, j: Double) = i + j

fun main() {
  addInt(5, 6) eq add(5, 6)
  addDouble(56.23, 44.77) eq
    add(56.23, 44.77)
}
```

`addInt()` takes two `Int`s and returns an `Int`, while `addDouble()` takes two `Double`s and returns a `Double`. Without overloading, you can't just name the operation `add()`, so programmers typically conflate *what* with *how* to produce unique names (you can also create unique names using random characters but the typical pattern is to use meaningful information like parameter types). In contrast, the overloaded `add()` is much clearer.

- -

The lack of overloading in a language is not a terrible hardship, but the feature provides valuable simplification, producing more readable code. With overloading, you just say *what*, which raises the level of abstraction and puts less mental load on the reader. If you want to know *how*, look at the parameters. Notice also that overloading reduces redundancy: If we must say `addInt()` and `addDouble()`, then we essentially repeat the parameter information in the function name.

Exercises and solutions can be found at www.AtomicKotlin.com.