PROGRAMMING LANGUAGE EXPLORATIONS

Second Edition

Ray Toal, Sage Strieker,
and Marco Berardini

A **Chapman & Hall** Book

CRC Press
Taylor & Francis Group

Programming Language Explorations

Programming Language Explorations helps its readers gain proficiency in programming language practice and theory by presenting both example-focused, chapter-length explorations of fourteen important programming languages and detailed discussions of the major concepts transcending multiple languages. A language-by-language approach is sandwiched between an introductory chapter that motivates and lays out the major concepts of the field and a final chapter that brings together all that was learned in the middle chapters into a coherent and organized view of the field.

Each of the featured languages in the middle chapters is introduced with a common trio of example programs and followed by a tour of its basic language features and coverage of interesting aspects from its type system, functional forms, scoping rules, concurrency patterns, and metaprogramming facilities. These chapters are followed by a brief tour of over 40 additional languages designed to enhance the reader's appreciation of the breadth of the programming language landscape and to motivate further study.

Targeted to both professionals and advanced college undergraduates looking to expand the range of languages and programming patterns they can apply in their work and studies, the book pays attention to modern programming practices, keeps a focus on cutting-edge programming patterns, and provides many runnable examples, all of which are available in the book's companion GitHub repository. The combination of conceptual overviews with exploratory example-focused coverage of individual programming languages provides its readers with the foundation for more effectively authoring programs, prompting AI programming assistants, and, perhaps most importantly, learning—and creating—new languages.

Ray Toal is Professor of Computer Science at LMU in Los Angeles where he's been teaching since 1986. He received his Ph.D. from UCLA in 1993 in Semantics, with minors in theoretical computer science and database systems. His current research interests are in programming language design, language translation, APIs, and computer security. He has consulted for several companies in the Los Angeles area, including Citysearch, Medaxis, Friendbuy, Handmade Mobile, M-GO, Criteo, Zoodiker, and (currently) Undivided, a startup with the mission of assisting parents of special needs children in helping their children lead their best lives. Ray has authored three books on programming languages and has been involved with projects at the Human Advancement Research Community (HARC). At LMU he enjoys interdisciplinary research and working on creating new cross-college programs such as the Interactive Gaming and Immersive Media minor, a collaboration between College of Science and Engineering and the School of Film and Television. Since 2017, he has been the lead instructor for the LMU cohort of Google's Computer Science Summer Institute.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Programming Language Explorations

Second Edition

Ray Toal, Sage Strieker and Marco Berardini

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Second edition published 2024
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2024 Ray Toal, Sage Strieker and Marco Berardini

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data
Names: Toal, Ray, author. | Strieker, Sage Angelica, author. | Berardini, Marco, author.
Title: Programming language explorations / Ray Toal, Sage Angelica Strieker and Marco Berardini.
Description: Second edition. | Boca Raton, FL : CRC Press, 2024. | Includes bibliographical references and index. | Summary: "Programming Language Explorations helps its readers gain proficiency in programming language practice and theory by presenting both example-focused, chapter-length explorations of fourteen important programming languages and detailed discussions of the major concepts transcending multiple languages. A language-by-language approach is sandwiched between an introductory chapter that motivates and lays out the major concepts of the field and a final chapter that brings together all that was learned in the middle chapters into a coherent and organized view of the field. Each of the featured languages in the middle chapters is introduced with a common trio of example programs and followed by a tour of its basic language features and coverage of interesting aspects from its type system, functional forms, scoping rules, concurrency patterns, and metaprogramming facilities. These chapters are followed by a brief tour of over 40 additional languages designed to enhance the reader's appreciation of the breadth of the programming language landscape and to motivate further study. Targeted to both professionals and advanced college undergraduates looking to expand the range of languages and programming patterns they can apply in their work and studies, the book pays attention to modern programming practices, keeps a focus on cutting-edge programming patterns, and provides many runnable examples, all of which are available in the book's companion GitHub repository. The combination of conceptual overviews with exploratory example-focused coverage of individual programming languages provides its readers with the foundation for more effectively authoring programs, prompting AI programming assistants, and, perhaps most importantly, learning-and creating-new languages"-- Provided by publisher.
Identifiers: LCCN 2024002989 (print) | LCCN 2024002990 (ebook) | ISBN 9781032119632 (hbk) | ISBN 9781032110844 (pbk) | ISBN 9781003222354 (ebk)
Subjects: LCSH: Programming languages (Electronic computers)
Classification: LCC QA76.7 .T62 2024 (print) | LCC QA76.7 (ebook) | DDC 005.13--dc23/eng/20240220
LC record available at https://lccn.loc.gov/2024002989
LC ebook record available at https://lccn.loc.gov/2024002990

ISBN: 978-1-032-11963-2 (hbk)
ISBN: 978-1-032-11084-4 (pbk)
ISBN: 978-1-003-22235-4 (ebk)

DOI: 10.1201/9781003222354

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

To Noëlle and the girls

—RT

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Preface xiii

CHAPTER 1 ▪ Welcome 1

1.1 THE STUDY OF PROGRAMMING LANGUAGES 1
1.2 PROGRAMMING LANGUAGE CONCEPTS 3
1.3 PROGRAMMING LANGUAGE ELEMENTS 4
1.3.1 Literals 4
1.3.2 Variables 5
1.3.3 References 5
1.3.4 Types 7
1.3.5 Expressions 9
1.3.6 Statements 9
1.3.7 Routines 10
1.3.8 Errors 10
1.3.9 Modules 10
1.4 PROGRAMMING LANGUAGE EVALUATION 11
1.4.1 Classification 12
1.4.2 Criteria 13
1.4.3 Tradeoffs 14

CHAPTER 2 ▪ Lua 15

2.1 HELLO LUA 16
2.2 THE BASICS 19
2.3 FUNCTIONS 24
2.4 SCOPE 26
2.5 CLOSURES 27
2.6 TABLES 28
2.7 METATABLES 30
2.8 COROUTINES 33

CHAPTER 3 ▪ JavaScript 39

3.1 HELLO JAVASCRIPT 40
3.2 THE BASICS 41
3.3 DESTRUCTURING 47
3.4 GENERATORS 48
3.5 METHODS 49
3.6 PROTOTYPES 49
3.7 SCOPE 55
3.8 PROMISES 55

CHAPTER 4 ▪ TypeScript 63

4.1 HELLO TYPESCRIPT 64
4.2 THE BASICS 65
4.3 GENERICS 72
4.4 OPTIONALITY 74
4.5 NARROWING 76
4.6 TYPE OPERATORS 78
4.7 TYPE ASSERTIONS 79

CHAPTER 5 ▪ Python 83

5.1 HELLO PYTHON 84
5.2 THE BASICS 85
5.3 TYPE HINTS 91
5.4 SCOPE 91
5.5 PARAMETER ASSOCIATION 92
5.6 SPECIAL METHODS 94
5.7 ITERATORS AND GENERATORS 95
5.8 DECORATORS 97

CHAPTER 6 ▪ Ruby 103

6.1 HELLO RUBY 104
6.2 THE BASICS 105
6.3 OBJECT ORIENTATION 107
6.4 SINGLETON CLASSES 110
6.5 BLOCKS 111
6.6 MIXINS 113
6.7 ACCESS CONTROL 116
6.8 METAPROGRAMMING 117

CHAPTER 7 ▪ Java 123

7.1 HELLO JAVA 124
7.2 THE BASICS 126
7.3 CLASSES 127
7.4 INTERFACES 131
7.5 GENERICS 133
7.6 TYPE VARIANCE 137
7.7 STREAMS 138
7.8 THREADS 139
7.9 METAPROGRAMMING 146

CHAPTER 8 ▪ Kotlin 151

8.1 HELLO KOTLIN 152
8.2 THE BASICS 153
8.3 CLASSES 157
8.4 EXTENSIONS 161
8.5 COMPANION OBJECTS 162
8.6 NULL SAFETY 162
8.7 COROUTINES 164

CHAPTER 9 ▪ Julia 169

9.1 HELLO JULIA 170
9.2 THE BASICS 171
9.3 MULTIDIMENSIONAL ARRAYS 177
9.4 MULTIPLE DISPATCH 179
9.5 METAPROGRAMMING 180
9.6 PARALLEL COMPUTING 183

CHAPTER 10 ▪ Clojure 189

10.1 HELLO CLOJURE 190
10.2 THE BASICS 193
10.3 NAMESPACES 196
10.4 PERSISTENT DATA STRUCTURES 197
10.5 TRANSIENTS 198
10.6 SOFTWARE TRANSACTIONAL MEMORY 199
10.7 AGENTS 201
10.8 THE EXPRESSION PROBLEM 201
10.9 MACROS 204

CHAPTER 11 ▪ Haskell 211

11.1 HELLO HASKELL 212
11.2 THE BASICS 214
11.3 LAZY EVALUATION 220
11.4 TYPE INFERENCE 220
11.5 TYPECLASSES 221
11.6 ALGEBRAIC DATA TYPES 224
11.7 EFFECTS 228
11.8 MONADS 229
11.9 SPARKS 232

CHAPTER 12 ▪ Go 237

12.1 HELLO GO 238
12.2 THE BASICS 240
12.3 POINTERS 246
12.4 SLICES 250
12.5 INTERFACES 251
12.6 PANICS 255
12.7 GOROUTINES 257

CHAPTER 13 ▪ C 267

13.1 HELLO C 268
13.2 THE BASICS 271
13.3 POINTERS 275
13.4 STORAGE DURATIONS 278
13.5 STRINGS 282
13.6 BUFFER OVERRUNS 283
13.7 THE PREPROCESSOR 284

CHAPTER 14 ▪ Swift 289

14.1 HELLO SWIFT 290
14.2 THE BASICS 291
14.3 OPERATORS 299
14.4 PROTOCOLS 301
14.5 EXTENSIONS 303
14.6 SAFETY FEATURES 305
14.7 AUTOMATIC REFERENCE COUNTING 307
14.8 STRUCTURED CONCURRENCY 310

CHAPTER 15 ▪ Rust 315

15.1 HELLO RUST 316
15.2 THE BASICS 318
15.3 OWNERSHIP 321
15.4 LIFETIMES 323
15.5 STRUCTS AND ENUMS 324
15.6 TRAITS 329
15.7 FEARLESS CONCURRENCY 331
15.8 MACROS 332

CHAPTER 16 ▪ Additional Languages 335

16.1 THE CLASSICS 335
16.1.1 Fortran 335
16.1.2 COBOL 336
16.1.3 Lisp 336
16.1.4 Algol 337
16.1.5 Simula 337
16.1.6 Smalltalk 337
16.2 CRYSTALLIZATIONS OF STYLE 338
16.2.1 APL 338
16.2.2 Prolog 339
16.2.3 K 340
16.2.4 Io 341
16.2.5 Factor 342
16.3 SCRIPTING 342
16.3.1 Bash 343
16.3.2 Perl 343
16.3.3 R 343
16.4 THE ML FAMILY 344
16.4.1 Standard ML 344
16.4.2 OCaml 344
16.4.3 F# 344
16.4.4 PureScript 345
16.4.5 Idris 345
16.5 SYSTEMS LANGUAGES 345
16.5.1 C++ 346
16.5.2 D 346
16.5.3 Nim 346
16.5.4 Zig 347

16.6 CONCURRENCY 347
16.6.1 Erlang 347
16.6.2 Elixir 347
16.6.3 ParaSail 348
16.6.4 Chapel 348
16.7 MULTIPARADIGM LANGUAGES 348
16.7.1 CLU 348
16.7.2 Ada 349
16.7.3 C# 349
16.7.4 Scala 350
16.7.5 Mojo 350
16.8 THE WEB 351
16.8.1 PHP 351
16.8.2 Dart 351
16.8.3 Elm 352
16.8.4 Grain 352
16.9 QUANTUM COMPUTING LANGUAGES 352
16.10 ESOTERIC LANGUAGES 352
16.10.1 Brainf**k 352
16.10.2 Befunge 354
16.10.3 LOLCODE 354
16.10.4 CJam 355
16.10.5 Pyth 355
16.11 ASSEMBLY LANGUAGES 356

CHAPTER 17 ▪ Concepts 359

17.1 CLASSIFICATION 360
17.2 STRUCTURE 361
17.3 EXECUTION 364

Glossary 369

Bibliography 377

Index 383

Preface

Much has happened since the question "Why would you want more than machine language?" was answered.

- 1950s–60s: Several computing professionals, most notably Grace Hopper, John McCarthy, and John Backus, create high-level, machine-independent programming languages. We soon have Fortran for scientific computing, COBOL for business, Algol for computing research, and Lisp for artificial intelligence. In the mid-1960s, PL/I and Algol 68 emerge for multi-purpose computing.
- 1960s–1970s: With the realization that tangled control flow negatively impacts project success, the *structured programming* revolution begins. Languages providing *information hiding*, such as Modula, CLU, Simula, Mesa, and Euclid address the "software crisis." Structured and modular features are bolted on to earlier languages. We also see a golden area of alternative programming paradigms and groundbreaking research yielding goal-driven computation, pattern matching, local and wide area networking, interactive graphics, and spatial representations of code.
- 1980s: *Object-oriented programming* (OOP) takes over the world. Though begun in the 1960s with Simula and refined in the 1970s at Xerox PARC with Smalltalk, OOP—or approximations to it—explodes in the 1980s with C with Classes (since renamed C++), Objective-C, Eiffel, and Self. Earlier languages such as Lisp and Pascal gain OO features, becoming Common Lisp and Object Pascal, respectively.
- 1990s: The World Wide Web appears and Perl becomes popular. Java, JavaScript, and PHP are created with web applications in mind.
- 2000s: With machine speeds increasing, interest in established dynamic languages such as Ruby takes off. Scala shows that static languages can feel dynamic. Clojure arrives as a dynamic, modern Lisp, leveraging Java's virtual machine.
- 2010s: Old things become new again. Multicore processors and "Big Data" revive interest in functional programming. Older languages such as Python and R, and the newer Julia language, find use in data science. Net-centric computing and performance concerns make static typing popular again as Go, Rust, Kotlin, Swift, and Zig challenge C and C++ for native applications.
- 2020s: The trend of marrying performance-intensive systems-oriented forms with expressive dynamic constructs continues and finds applications in programming AI-powered systems, as evidenced by languages like Mojo.

Each of the thirty-five languages we've mentioned, and the tens of thousands we did not, is created for some purpose. New creations may address a particular problem domain, improve upon old languages, or help us express computations in exciting new ways. There's no single best language for all possible tasks, so we learn many of them.

This book aims to acquaint you with a number of programming languages *and* with fundamental concepts transcending multiple languages. Language tours will present these concepts augmented with plenty of runnable scripts. We hope to provide you with a foundation for more effectively authoring programs, prompting your AI programming assistants, and, perhaps most importantly, learning—and creating—new languages.

ORGANIZATION

In order to become proficient in multiple languages, you must learn fundamental language *concepts*. Conversely, mastering concepts requires that you learn multiple languages. Both directions of influence are important! We've chosen to sandwich a language-by-language approach between an introductory chapter that lays out the major concepts of the field and a final chapter that brings together all that was learned in the middle chapters into a coherent and organized view of the field, with the goal of facilitating future study.

As we present each language in the middle chapters, we'll introduce new concepts as they appear, and revisit familiar ones, comparing their implementation with those from languages we've seen in prior chapters. We introduce each with a common trio of example programs, take a brief tour of the basic language features, and highlight interesting aspects from its type system, functional forms, scoping rules, concurrency patterns, and metaprogramming facilities.

We've carefully chosen the order of languages to make a story out of the book. We start with Lua for its elegant design and relatively small but powerful feature set. JavaScript follows with similar features to Lua (such as first class functions and prototypal inheritance), while adding a host of modern features and a powerful event-driven, asynchronous programming model. As perhaps the world's most popular language, JavaScript has influenced hundreds of successors, the most popular of which, TypeScript, comes next, along with an introduction to static types.

The next two languages, Python and Ruby, are general-purpose scripting languages, in which we focus on first-class types and object-oriented programming, respectively. We then encounter Java, an extraordinarily popular language in the enterprise computing space. Java was introduced along with the Java platform, which includes a virtual machine and an extensive set of powerful libraries for just about every task imaginable. Hundreds of languages are targeted to this platform, including our next case study, Kotlin. This pair of languages provides us with an opportunity to study how expressive forms in one language are improved upon in a successor language.

Next comes Julia, a modern language for scientific computing featuring multidimensional arrays and mechanisms for concurrent and distributed programming. It also introduces an approach to object orientation—multimethods—shared by the next stop on our tour, Clojure. Clojure is a modern dialect of the venerable Lisp, famous for its hygienic macros, support for functional programming, and concurrency. Next up is another functional language, Haskell, which while less popular, is essential to visit: its powerful type inference mechanism and support for pure functions and lazy computing are not found in quite the same way anywhere else on our tour. Our next language, Go, introduces explicit pointers, an interesting convention for package export, and its signature feature, the goroutine, which enables large-scale concurrent and distributed applications rather unlike those of our previous three languages.

Our last three languages place the burden of memory management knowledge on the programmer, forgoing the tracing garbage collector taken for granted in our first 11 languages.

First is the venerable C, an intentionally small and unsafe system-level language, but worth studying to see low-level details every other language hides from application programmers. Next up is Swift, exemplifying many of the modern trends in programming language design—it is static, safe, and expressive—and achieves memory safety not with a garbage collector but via reference counting. The much-loved Rust closes the tour with its elegant approach to compile-time guarantees of memory and thread safety that surprisingly maintain zero-cost abstractions.

Each language chapter ends with a summary, a link to the companion website where further readings and references to open source projects can be found, and a collection of exercises, designed as *further explorations*. The exercises motivate you to find the language's online playgrounds, REPLs, or downloadable installation, and to try out various interesting snippets of code and muse on what they say about the language. Occasionally we'll suggest that you write more substantial scripts, research various aspects of the language not covered in the chapter proper, or ask you to do some comparisons between languages.

Following our featured language chapters, we provide a brief tour of a couple dozen or so additional languages, grouped into twelve thematic sections including classic languages, stylistic exemplars, web-oriented languages, concurrency-focused languages, and even esoteric languages. We close with a chapter covering the major concepts of the field in a language-independent fashion. The concepts are organized around three major themes:

- **Classification**. How are the information units manipulated by our programs represented? How are low-level notions of information chunked into units that *humans* can process? How are these units of information classified? How do they behave? Answering these questions motivates the creation and development of type systems.
- **Structure**. How are programs—expressions of computations—linguistically structured? What are the major architectural elements, or entities from which programs are formed? Answering these questions leads us to notions of naming and reference, expressions, statements, objects, routines, modules, and more.
- **Execution**. How do computations unfold in time? How are processes and threads implemented and how do they communicate? We'll consider sequential, conditional, iterative, synchronous and asynchronous flows, disruption, nondeterminism, concurrency, distribution, and parallelism.

By bringing the concepts together *after* introducing them in the context of whole-language overviews with runnable programs, we can provide a technical overview of the entire landscape of the programming languages field to readers with a fair amount of existing, practical knowledge. Our goal is to provide the strongest possible foundation to undertake deeper study into programming language theory and implementation.

A NOTE ON THE SELECTED LANGUAGES

No choice of languages can possibly satisfy everyone; you will undoubtably see your favorite languages missing, or wonder why the great influencers—Algol, Lisp, Smalltalk, Prolog, ML, etc.—lack their own special chapter. Generally, the influencers have modern descendants: Clojure is a modern Lisp; Ruby borrows much from Smalltalk; Haskell owes much to the venerable ML family. Python, Ruby, and Java, each over 25 years old, continue to evolve in significant ways. Recent languages, such as TypeScript, Julia, Go, Kotlin, and Swift, collect many of the best ideas of the last fifty years. In the systems programming landscape, we would be remiss to not cover C, but we've found Rust to cover most, and in some cases more, of the interesting concepts found in the more popular C++.

We will generally emphasize *ideas* over *features*. Alan Kay [60] has written:

> Programming languages can be categorized in a number of ways: imperative, applicative, logic-based, problem-oriented, etc. But they all seem to be either an "agglutination of features" or a "crystalization of style." COBOL, PL/I, Ada, etc., belong to the first kind; LISP, APL—and Smalltalk—are the second kind.

We haven't selected *only* languages of the second kind, but we will try to emphasize big ideas, insights, and idiomatic expression in addition to features. While we focus on today's languages, many of the great classics, along with a handful of newcomers and even a few esoterics, are highlighted briefly in our "Additional Languages" chapter.

AUDIENCE

This book is targeted to both professionals and advanced college undergraduates looking to expand the range of languages and programming patterns they can apply in their work and studies. We've paid attention to modern programming practices, covered some cutting-edge languages and patterns, and provided many runnable examples so readers will discover new skills to apply to their craft. In addition, we've slipped in a bit of academic terminology because knowledge of foundational principles is required to take the leap from hobbyist to professional builder of scalable, efficient systems. However, the book remains unapologetically an *exploration* of high-level language practice, placing it squarely between a tutorial and a reference: we make no attempt to teach any particular language from scratch, nor to provide complete descriptions of any language. Indeed, we've kept the focus *on the concepts and practices underlying programming language design and usage*, many which are covered within multiple whole-language overviews.

If you are an instructor looking for material for a programming languages or compiler construction course, you may at first find our emphasis on a language tour unusual, but, we hope, also quite useful (as we expect most students will). The traditional topics—binding, expressions, control flow, types, modularity, concurrency, and metaprogramming—appear several times to reinforce student learning: they are introduced in the opening chapter, covered repeatedly within the context of whole-language overviews, then summarized in the final chapter. Students have ample scenarios to explore how the various features *within* a given language work together as well as how a given feature works *across* languages. You can, however, open a traditional concept-oriented course with the introductory chapter, then "drive" the course from the final chapter, selecting content from the middle chapters as desired. You will find many, though intentionally not quite all, of the squares in a grid similar to Figure 1 targeted.

PREREQUISITES

This book is not for beginning programmers. We're assuming you know at least two languages pretty well, and hopefully a couple more. You should know the basic concepts surrounding variables, expressions, operators, functions, and basic data structures, and have experience writing nontrivial applications—this book is a tour, not a tutorial. We will be covering many interesting (and often powerful) features, and purposely do so with code that is **often very dense**, **and sometimes cryptic**, even when presenting unfamiliar paradigms for the first time. Be on the lookout for new concepts that are introduced within the code examples themselves, rather than in the surrounding text. Don't be surprised if bits of unexplained syntax appear in the examples—welcome this chance to do some research or converse with a colleague, mentor, or well-trained chatbot about the code.

	Binding/Scope	Funcs/Closures	Frames/ArgPassing	Prototypes/Classes	Extension	Dispatch	Operators	Control Flow	Matching	Type Systems	Type Inference	Immutability	Persistent Structs	Processes/Threads	Messaging	Synchronization	Pointers/Garbage	Metaprogramming
Lua																		
JavaScript																		
TypeScript																		
Python																		
Ruby																		
Java																		
Kotlin																		
Julia																		
Clojure																		
Haskell																		
Go																		
C																		
Swift																		
Rust																		

Figure 1 Languages and concepts

We don't enforce any particular style for running the book's examples. Feel free to use your favorite cloud platforms for authoring and running code, or find and install the software needed to run the examples locally on your own machine. You may find the latter approach better allows you to keep up with the latest versions of each language and configure your personal IDE or text editor with your favorite plugins for the most enjoyable coding experience, but doing so is not required.

We've chosen to favor runnable scripts with assertions throughout the text, as opposed to transcripts of sessions with a REPL. The former was chosen as it allowed us to run each of the code examples through an automated test suite. However, when practicing with a new language, we would very much encourage exploration and experimentation with REPLs and similar playgrounds. Notes on REPL usage for various languages can be found on the book's companion website.

SUPPLEMENTARY MATERIAL

All of the code from the book, together with test scripts that show how to run the examples locally, can be found in the online repository at `https://github.com/rtoal/ple`. We've also included notes in the repository for installing and working with the languages featured in the text, since this information changes too frequently to be mentioned in print. The repository also contains additional code of interest, and will grow and be updated as the programming languages evolve. Errata, additional exercises, and bonus material can be found at the companion website at `http://rtoal.github.io/ple/`.

ACKNOWLEDGEMENTS

We'd like to thank Rachel Rivera, Eileen Choe, and Alex Schneider for their contributions to the first edition of the text, many of which have made it through to this second edition. We're also grateful to a number of friends who read portions of the text and provided constructive comments, including Loren Abrams, Meta; David Pedowitz, Netflix; Andy Won, Amazon; Saturnino Garcia, University of San Diego; Caskey Dickson, Microsoft; Jasmine Dahilig, Undivided; B.J. Johnson, Loyola Marymount University; Zach Dodds, Harvey Mudd College, Craig Reinhardt, California Lutheran University, and Kira Toal, Booz Allen Hamilton. Errors in the text are, of course, the responsibility of the authors, not the reviewers. Gabrielle Barnes, Coby Schumitzky, Zoey Ho, Juan Carrillo, Stephen Smith, Andrew Akers, Trixie Roque, Matt Flickner, Andrés Buriticá, Zane Kansil, Ed Bramanti, Amanda Marques, Mike Megally, Aaron Jacobs, and Carl Downing were also available for several discussions, and at times wrote some sample code.

We are also grateful to the staff at Taylor & Francis, including Randi Slack, Publisher, Solomon Pace-McCarrick , Editorial Assistant, and Robin Lloyd-Starkes, Project Editor, without whose hard work this book would not have been possible.

CHAPTER 1

Welcome

Hello! It appears that you might be interested in languages of the programming kind. Whether you are certain that you are, or are not so sure that you are, we'll try, in this brief but very dense chapter, to pique your interest and make the study of programming languages look not only fun, but worthwhile.

1.1 THE STUDY OF PROGRAMMING LANGUAGES

Learning new programming languages enables you to think about, and express, computations in new and sometimes surprising ways.

How so? Try writing a function to sum the squares of the even numbers in an array. If all you know is C, you might think through this exercise as follows:

```
int sum_of_even_squares(int* a, unsigned int length) {
    int total = 0;
    for (unsigned int i = 0; i < length; i++) {
        if (a[i] % 2 == 0) {
            total += a[i] * a[i];
        }
    }
    return total;
}
```

But this is unsatisfying: *why is the variable* `i` *there?* Why do we care about the *indexes* of the array? Why does the function have to be passed the number of items in the array? Certainly, there must be more concise and expressive ways to perform this task. In Swift, we can iterate over the array *elements* directly:

```
func sumOfEvenSquares(of a: [Int]) -> Int {
    var sum = 0
    for x in a where x.isMultiple(of: 2) {
        sum += x * x
    }
    return sum
}
```

Slightly better, perhaps, but we can express a solution in a completely different way. Rather than the `for`-loop putting each individual element in charge of the overall computation, we can let the array do the work, by:

1. *Selecting*, or *filtering by*, its even elements,
2. *Mapping* the square operation over the selected elements, then finally
3. *Reducing* the squares to a single value by summing them together.

Many popular languages have these operations. Ruby does:

```
def sum_of_even_squares(a)
  a.select{|x| x % 2 == 0}.map{|x| x * x}.reduce(0, :+)
end
```

And so does Clojure:

```
(defn sum-of-even-squares [a]
  (->> a (filter even?) (map #(* % %)) (reduce +)))
```

We should be fair and show that Swift is just as capable:

```
func sumOfEvenSquares(of a: [Int]) -> Int {
    return a.filter{$0 % 2 == 0}.map{$0 * $0}.reduce(0, +)
}
```

Java, too, can filter, map, and reduce:

```
public static int sumOfEvenSquares(int[] a) {
    return IntStream.of(a).filter(x -> x%2==0).map(x -> x*x).sum();
}
```

Python expresses mapped and filtered sequences with a **generator expression**:

```
def sum_of_even_squares(a):
    return sum(x*x for x in a if x % 2 == 0)
```

K uses powerful array operators to yield a remarkably terse one-liner:

```
sumofevensquares: {+/x[&~x!2]^2}
```

There's more. Ask yourself: What happens after the assignment `x = y + z`? The variable x is updated and the old value is lost forever, right? Not so fast! You may encounter languages in which the "value" of x is its entire *history* of values, or in which x automatically updates whenever y or z are subsequently changed. There are even languages that prohibit assignment altogether!

The fun doesn't stop there. In learning new languages, you'll find lists that aren't physically stored but produce their elements on demand, and models of the world where every piece of data, even small integers, can act as "little computers" that can send and receive messages. You may run into languages that can perform computation during type checking and languages whose processes restart after they die. Language models and ideas such as these are not only fascinating, but worthy of study. In [112], Bret Victor shows how work in the 1960s and 1970s created programming models that manipulate data rather than code,

are parallel rather than sequential, use goals (*what* is needed) instead of procedures (*how* to compute what is needed), and are represented spatially rather than with lines of text. It would be a tragedy, he says, if people were to miss these and other ideas, or worse, to simply master one way of programming and forget that they could ever have new ideas about programming models.

Exposure to new languages can help you write better code in the languages you *do* use every day as you discover ways to simulate features that your language lacks. Learning the vocabulary of the concepts common to multiple languages may help you better understand obscure error messages presented to you by a compiler. You'll also acquire the foundation to learn new languages more easily, and even design your own language. Even if you don't create the next Python or C++, remember that language design happens at small scales too. You can create little languages to control robots or painting tools, structure commands in a conversation-based game, describe formulas to input into a calculator or spreadsheet, or specify questions to ask of a search engine or database.

1.2 PROGRAMMING LANGUAGE CONCEPTS

What are some of the concepts that will help guide our study of programming languages? We can look to the various theories of computation have developed over the last century: **language theory**, concerned with how computations are expressed, **automata theory**, concerned with how computations are carried out, **computability theory**, concerned with what can and cannot be computed, and **complexity theory**, concerned with quantifying the resources required for computation.

Programming languages enable the communication of computational ideas between humans and among humans and machines. While the question "what is computation?" is a little outside the scope of this book, we'll assume that it involves the processing of information over time. We can then ask questions to encourage various concepts to emerge in our attempts to find answers:

- What kinds of information units do we wish to store and process, and what are their behavioral capabilities? (Typing)
- How do we denote, or encode, these values? (Data Representation)
- From what kinds of entities can we build our programs, and how do we name these entities? (Binding)
- How do we "chunk" information and computational procedures into larger units to better manage large systems? (Abstraction)
- How can we build entities that are customizable in some way? (Parameterization)
- How do entities emit and respond to events? (Communication)
- How do we synchronize various sub-computations? When do we require sequential execution and when do we allow true parallel execution? (Coordination)
- How can computations know about themselves? (Metaprogramming)
- How can we know our programs always do what they should? (Correctness)
- How do we know they never do what they should not? (Security)

This list of questions is not intended to be complete. In fact, as our understanding of computation evolves over time, some of the questions and the areas of concern may even change. What is important, though, is that these questions necessarily have multiple answers, and it is precisely the variations in these answers that give rise to different programming languages.

1.3 PROGRAMMING LANGUAGE ELEMENTS

To understand the principles and concepts that facilitate the expression of computation, we must go beyond general questions surrounding the nature of computation, and perform the actual work of inventing and discovering the basic linguistic elements that underlie these concepts. These elements provide us with the working vocabulary needed to describe programming principles and compare and contrast languages in meaningful ways.

Computations process information units we call **values**. Values can be numeric, textual, logical, computational, groupings of values, indirect references to other values, or indications of missing or unknown information. A program manipulates values through **entities**—structures that describe, store, or transform values in desired ways. Let's explore a few of the more popular kinds of values and entities.

1.3.1 Literals

A **literal** is a (direct) representation of a value. A few examples follow:

- `95` is a literal for the value *ninety-five*. Alternate representations for this value include `0x5F` (`0x` is a common prefix for hexadecimal numerals), `0b1011111` (`0b` for binary), and `0.0095E4` (`E` means "times-ten-to-the").
- `true` (sometimes `True` or `T`) is the literal representing truth; `false` (or `False` or `F`) is the literal representing falsity.
- `'*'` is a character literal, denoting the character named ASTERISK.
- The quoted sequence of characters `"Hello, how are you?"` is a **string** literal representing a common English greeting.
- `:alice` (rendered in some languages as just `alice`) is a kind of literal known as an **atom**, or **symbol**. Atoms stand only for themselves and nothing else. Unlike strings, they are not decomposed into character sequences.
- `[0, true, 98.6]` is a literal for the **sequence** of three values: *zero*, *truth*, and *ninety-eight point six*.
- `{latitude: 29.9792, longitude: 31.1344}` is a literal representing the location of the Great Pyramid of Giza.
- `x => x / 2` is a literal representing a function that halves its argument.
- `null` (sometimes `nil` or `None`) is a literal indicating the intentional absence of a value.
- `undefined` is a literal used to indicate some desired value is unknown or purposely not being divulged. Think of this as "I don't know," "I don't care," or "None of your business."

1.3.2 Variables

A **variable** provides access to a value through a name. Do not confuse variables with values: variables may *stand for*, *own*, or *hold* values, they are *not themselves values*. Nor should you confuse a variable with its name. Instead, a variable is a container into which a value can be stored. If a variable is allowed to store different values throughout a program's execution, it may be more properly called an **assignable** [42]. We often think of variables as boxes; for example, the sequence of JavaScript assignments `x = 1; y = true; z = "so " + y.toString()` produces the useful rendering:

Compound values, generally known as **objects**, store values in "internal" variables, called **properties**, **fields**, **slots**, or **attributes**. For example, we may have a variable named `airport` holding an object with two internal variables, one of which is itself an object:

Many languages provide the ability to mark a variable **immutable**, prohibiting the variable from ever containing a value other than its initial value. Variables that *are* allowed to store new, updated, values are **mutable**. We use the term **constant** for an immutable variable whose value is known before the program is run.[1] If values in at least one of the internal variables (fields) of an object may be changed, we say the object itself is mutable, otherwise immutable.

1.3.3 References

Interestingly, some languages have prohibitions on the kinds of values that variables can store! For example, Lua and JavaScript allow variables to hold numbers, booleans, and strings, but do *not* permit variables to hold object values directly—they must instead hold a special kind of value, known as a **reference**, which provides an additional level of access to a value. So, although the syntax of these languages make it *appear* that an object is being stored, as in the following Lua script:

```
a = {x=3, y=5}    -- creates an object, stores reference in a
b = a.y           -- directly stores 5 into b
c = true          -- directly stores true into c
d = {x=3, y=5}    -- creates an object, stores reference in d
e = d             -- does not create an object, induces sharing
```

[1] While this definition may be technically precise, do note that vocabulary sometimes, unfortunately, varies from language to language. You may hear the term "constant" used to denote any immutable variable, even one initialized during program execution. Sometimes the term will even be used to describe a literal.

the variables a, d, and e hold reference values, not object values, like so:

References can be very helpful! For example, assignments, such as `e = d` above, avoid the cost of copying (potentially large amounts of) internal object data, which can translate into significant performance gains.

However, references come with security implications. Assigning references creates **aliasing**: `d.x` and `e.x` are now two names for the same variable. An assignment to one updates the other! This sharing of objects through references can lead to accidental (or malicious) modification through these aliases. Programmers, therefore, have the responsibility for knowing exactly when references are used and when object internals are copied—rules that, for better or worse, vary from language to language. They must also ensure references cannot be manipulated by attackers into exposing sensitive data.

The target of a reference—the **referent**—requires storage space known as **memory**. If variable holding a reference is reassigned, there is a possibility that the previously referenced value is no longer accessible. Here's another example in Lua:

```
p = {x=3, y=5}
p = {x=2, y=8}
```

The second assignment renders the first object inaccessible:

Inaccessible storage is officially known as **garbage** as it wastes system resources. Different languages deal with garbage in different ways. Many languages (including those in Chapters 2 through 12 in this text) provide a **garbage collector** that runs behind the scenes, picking up after the programmer. Some track, or even count, references to data, reclaiming inaccessible storage when the last reference goes away. Some give the programmer *no* support at all, requiring the programmer to both request storage when needed and give back, or **free**, storage that is no longer needed. Freed storage can be reallocated by the runtime for future requests. Attempting to use freed storage or freeing the same storage area twice can lead to serious problems; we'll cover these in our chapter on C.

In some, but not all, languages, you will encounter this:

This is a reference to nothing, known as a **null reference**. It is *not* a reference to the value `null`, nor even to `undefined`. There's nothing on the receiving end—no referent. Nothing to dereference. What could go wrong? Apparently, quite a bit. The inventor of the null reference, Sir Charles Anthony Richard Hoare, back in 2009, stated:

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965.... My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Despite the number of potential disasters surrounding references, they are essential to modern programming. Language designers have taken note and, perhaps with the exception of C, have gone to great lengths to make their use relatively safe, without sacrificing too much convenience. But even in safe languages, programmer vigilance is required, especially when dealing with shared references to mutable entities.

1.3.4 Types

A **type**, roughly, determines a collection of values with some prescribed behavior. That a value v has type T means that only certain operations may be applied to v. The notion of "type" in programming language theory is extremely deep and has filled entire books, for example [89]. Typing is central to the study of languages, as the point of a language's type system is *to provide (meaningful) constraints on what we can and cannot say.*

We have an intuitive sense that values belong to types—`true` to the boolean type, `0.3` to some kind of numeric type, and `"Hello"` to the string type. Exactly how these types are defined, though, can vary significantly by language.

Some have bounded integer types (see Table 1.1) that map directly to efficient hardware arithmetic units, but must deal with overflow. Consider the 8-bit signed integers, bounded to the range $-128 \ldots 127$. What is $123 + 8$? 131 is not in the range! Should the operation cause an error, **saturate** to 127 ("as high as it can go"), or "wrap around" to -125 (with **modular arithmetic**)? Some languages provide an unbounded integer type to sidestep this problem.

For floating-point numbers, many languages use the IEEE-754 [51] standards **binary32** and **binary64**. The former represents some 4 billion values between -3.4×10^{-38} and -3.4×10^{38}; the latter represents some 18 quintillion values between -1.7977×10^{-308} and -1.7977×10^{308}. Each also includes $-\infty$, $+\infty$ and a special value `NaN`, meaning *not-a-number*—a useful result for operations such as $0/0$ and $\infty - \infty$. Since floating-point number schemes must choose which numbers to represent and which not to, floating-point arithmetic is not exact. Even the high-precision binary64 standard is subject to round-off error (e.g., $0.1 + 0.2 = 0.30000000000000004$). Some languages allow the programmer to bypass these errors with arbitrary-precision real number types (often called **decimal types**) and with **ratio** (a.k.a. rational) types that avoid round-off errors in integer division.

Bits	Signed	Unsigned
8	−128 . . . 127	0 . . . 255
16	−32768 . . . 32767	0 . . . 65535
32	−2147483648 . . . 2147483647	0 . . . 4294967295
64	−9223372036854775808 . . . 9223372036854775807	0 . . . 18446744073709551615

Table 1.1 Fixed-size integers

For text, units of *writing*, called **graphemes**, are made from units of textual *information*, called **characters**. For example, the grapheme for (the emoji) *Woman scientist* is composed of three characters: WOMAN, ZERO-WIDTH JOINER, and MICROSCOPE. Languages often provide a **string** type for character sequences, and allow extraction of code points and the underlying bytes that encode characters. Grapheme extraction usually requires external libraries.

You may often see hierarchies of types, where a value of a subtype (e.g., `Dog`) can be used wherever a value of a supertype (e.g., `Animal`) is expected. You may find languages with only a single `list` type, and others that distinguish, say, lists of integers from lists of strings. In the latter, a **generic**, or **parametric**, type includes a **type parameter** (e.g., `List<T>`), and concrete types instantiate the parameter (e.g., `List<String>`). Function types are similar: some languages place all functions into a single type, while others feature a generic function type parameterized on the types of arguments and permitted return values.

While every *value* has a type (or even multiple types, when supertypes and subtypes are present), many languages allow the imposition of **type constraints** on *variables*. For example, the Java declaration

```
var message = "Please wait, this may take a minute";
```

constrains the variable `message` to values of type `String`. When a language enforces constraints at compile-time (before the program is run), the constraint defines the variable's **static type**. If it is possible to compute the type of a variable's value at run-time (during program execution), the computed type is a **dynamic type**. Some languages have one type system for the static and dynamic types. Some, including TypeScript, Python, and Java have a rich set of static types and a weaker set of dynamic types.

Closely related to the notion of type is that of **class**, which, roughly, is a kind of factory for instantiating objects that have a particular internal structure. Some languages have no classes, some conflate the notion of class and type, and some take great care to distinguish classes from types. There's more: **interfaces** and **typeclasses** are types that other types can conform to. For example, a type whose instances can be tested for equality might belong to the typeclass *Equatable*; if its instances can be compared via `<`, it might belong to the typeclass *Comparable*. Types with least and greatest values may be members of the *Bounded* typeclass; collection types supporting iteration might belong to *Iterable*. Going meta, in some languages the type of type is just a type; in others, it's called a **kind**.

When values are combined in a way that is inconsistent with their types, such as dividing a number by a string, a language definition must specify whether to (1) declare the program erroneous, or (2) find replacement values to make things work. JavaScript, for example, evaluates `15 / "3"` as 5, as the string `"3"` looks like a number. Languages that favor replacements, known as **coercions**, are **weakly typed**; those that are stricter and favor reporting type errors are **strongly typed**.

1.3.5 Expressions

An **expression** is a combination of literals, variables, and operators that is **evaluated** to produce a value. Some operators require all of their operands to be evaluated; others, like the popular `&&` ("and then") and `||` ("or else"), do not. An example of expression evaluation (using JavaScript for concreteness), where s holds the value `"car"`, y holds the value 100, and *found* holds the value `true`, follows:

```
7 * s.indexOf('r') + Math.sqrt(y) / 2 <= 0 || !found
⇒ 7 * 2 + Math.sqrt(y) / 2 <= 0 || !found
⇒ 14 + Math.sqrt(y) / 2 <= 0 || !found
⇒ 14 + Math.sqrt(100) / 2 <= 0 || !found
⇒ 14 + 10 / 2 <= 0 || !found
⇒ 14 + 5 <= 0 || !found
⇒ 19 <= 0 || !found
⇒ false || !found
⇒ !found
⇒ !true
⇒ false
```

Operators with a single operand are **unary** operators; those with two operands, **binary**, and three, **ternary**. Unary operators generally appear in the **prefix** position, as in the expressions `-x` or `~x`, though **postfix** is sometimes seen. Binary operators often appear as **infix** operators (between the operands). In general, the number of operands is known as the operator's **arity** and the position of the operator is known as its **fixity**.

The **precedence** of an operator determines whether its application to its operands precedes applications of operators appearing at the same level in the source code. To say operator $*$ has higher precedence than operator $+$ means that $a + b * c$ parses as $a + (b * c)$, that is, multiplication *precedes* addition in the evaluation. **Associativity** describes the structure of expressions among operators of the same precedence: given the expression $a \bullet b \bullet c$, a left-associative $\bullet$ would cause the expression to parse as $(a \bullet b) \bullet c$ (leftmost operation applied first), right-associativity would induce the parse $a \bullet (b \bullet c)$, and if the operator were nonassociative, the expression would be illegal.[2]

1.3.6 Statements

A **statement** performs an action. Common statement types include (1) declaration statements to bring named entities (e.g., variables, functions, types) into existence, (2) expression statements to evaluate expressions, (3) assignment statements, (4) invocation statements to execute chunks of code, (5) conditional statements (`if`, `unless`, `switch`, `match`, `when`) to execute code only under certain conditions, (6) iteration statements (`loop`, `while`, `for`, `repeat`) to execute code repeatedly, and (7) disruptive statements (`break`, `continue`, `retry`, `throw`, `return`, `yield`, `resume`) for changing the "normal" control flow.

Unlike expressions which produce values, statements are executed for their **effects** (sometimes called **side-effects**), such as updating the values held in variables, or writing to files or databases. Statements are often combined to run sequentially (run s_1 then s_2), nonde-

[2] For example, many languages make relational operators such as `<` nonassociative, since the expression `x<y<z` does not have an obvious interpretation: is it `(x<y)<z` or perhaps `(x<y)and(y<z)`?

terministically (run either s_1 or s_2), or in parallel (run s_1 and s_2 "at the same time," either on separate processor cores or with interleaved execution on a single core).

1.3.7 Routines

A **routine** is a (possibly parameterized) unit of code. Routines that can only start from the beginning when called are known as **subroutines**; those able to resume from where they last left off are known as **coroutines**. Some languages use the term **function** in place of subroutine. Others use the term only for subroutines that return values, while using **procedure** for subroutines returning no values. Routines that are properties of—or directly associated with—an object, whose execution reads or manipulates the object's state, are often known as **methods**.

A running program is known as a **process**. The operating system will allocate one or more **threads** (lines of execution) to the process, onto which subroutines and coroutines are mapped. In some cases, the programmer may access these **platform threads** directly; in others, the runtime system chooses its own strategy to map routines onto threads. The terms *thread* and *process* may also be used for concurrently executing units *within* a program: threads communicate via shared memory, while processes maintain their own local memory and communicate exclusively via **messages**. While coroutines *must* interleave their execution to form a single sequence of instructions and explicitly yield control to each other, threads and processes run independently and may be **preempted** by the operating system when their number exceeds the number of available hardware execution units.

Routines may be invoked directly in code or be triggered by **events**. Events may originate outside of a program, via a user action (mouse click, form text entry, button press, device rotation, sensory input from a vision or haptic system), or within the program via **event emitters**.

1.3.8 Errors

When (not if) things go awry, it is often best to *fail fast* and terminate the current thread of execution, so that inconsistencies do not spread throughout the computation and cause significant damage down the line. But if a problem *can* be dealt with within a program (and possibly recovered from), we may benefit from errors being instances of a special error type.

Many languages provide a means for **throwing**, or **raising**, error objects to signal a problem, and a means to **catch**, or **handle**, thrown errors. Throwing and catching, while sometimes convenient, disrupt the normal control flow and are not universally adopted. As an alternative, some languages permit operations to return two values, one indicating success or failure, and the other for the expected value on success. You may also find **result values** which contain the expected value *or* an error. Some languages offer direct support for the "error" condition of an entity not being found in a search through the use of **option values**—containers that either hold the found value or are empty, indicating the desired value is not present.

1.3.9 Modules

Modern languages support the notion of programming in the large via **modules** or **packages**, which encapsulate strongly related entities of a library or application. Modules may

bundle types, objects, functions, coroutines, or even other modules. The programmer may **export** some of the entities to the rest of the program, leaving the others private to the module. Modules may also **import** entities from other modules. Import and export are usually explicit but can be implicit as well. Often, but not always, modules correspond to files. Modules are somewhat related to **namespaces**, which generally refer to ad-hoc groupings of names that need not be tied to any particular module.

1.4 PROGRAMMING LANGUAGE EVALUATION

Why do people sometimes *argue* over their favorite languages, often claiming "their" language is better than "your" language? All languages have their flaws, which can be a source of great humor: Gary Bernhardt's *WAT* [11], Melissa Elliott's *PHP Manual Masterpieces* [25], and Eric Wastl's *PHP Sadness* [114]. But most languages actually *do* some things well and other things not so well. You might be able to rate a language's suitability for certain tasks, but an overall language rating is not worth looking for. To judge a language, you must know *why* it was designed. Let's take a look at a few of these reasons:

Fortran	Numeric computation
Lisp	Symbolic computation; Artificial intelligence
COBOL	Business computation
Algol 60	Algorithmic description
APL	Array processing with a concise mathematical notation
Algol 68	General-purpose computation
Pascal	Teaching structured programming
Modula	Modular programming
C	Systems programming
CLU	Adding modular type extensions to Algol
SQL	Database applications
Scheme	A simpler alternative to Lisp
Prolog	Expert systems; Natural language processing
Smalltalk	Personal computing
Ada	Megaprogramming; Embedded systems
C++	Simulation
ML	Theorem proving
Eiffel	High reliability
Perl	Scripting with a focus on text processing
Erlang	Massively scalable, available, soft real-time systems
J	APL without the fancy characters
Python	Scripting with an extensible language
Ruby	The language Matz[3] wanted, and to be better than Perl
Lua	General-purpose scripting
K	Ultra high-performance numerical (typically financial) analysis
R	Statistical and graphical computing
Java	Compact downloadable executable components
C#	Enterprise computing on the .NET platform
JavaScript	Client-side scripting for web applications
OCaml	Adding object orientation to ML

[3] Yukihiro Matsumoto, the designer of Ruby

PHP	Server-side scripting for web applications
PostScript	Formatting of printed documents
Haskell	Unifying the best of existing functional languages
Io	Dynamic language experimentation
Scratch	Creating and sharing stories, games, and animations
Scala	Supporting both functional and object-oriented programming
F#	Solving complex problems in a simple way
Clojure	Providing a modern Lisp dialect for the JVM
Nim	Expressivity in a systems programming language
CoffeeScript	Improving JavaScript
Dart	Modern web applications
Snap!	Exploratory programming
Go	Large distributed, interconnected systems
Parasail	Safe, secure, highly parallel applications
Chapel	Parallel programming at scale
Rust	Large, safe, network clients and servers
TypeScript	Large-scale JavaScript applications
Quipper	Functional programming for Quantum computers
Elm	Browser-based apps in a functional, reactive style
Elixir	Modernizing and extending Erlang
Kotlin	Fixing and improving Java
Julia	High-performance scientific computing in a dynamic language
Crystal	High expressiveness together with speed and safety
Swift	iOS development with modern language features
Reason	Combining the best of OCaml and JavaScript
Futhark	High-performance, data-parallel array programming
Q#	Integration of quantum and classical computation
Odin	Simplicity and high performance
Zig	Safety, efficiency, and reusability
Ballerina	Network interaction
Grain	Support for web applications in a modern language
Verse	Programming metaverse applications
Mojo	Programming machine learning applications

It's a good bet that a language designed expressly for the purpose of formatting printed documents might not be good for writing web servers. So why argue whether PostScript is "better than" Go? The correct retort to "A is better than B" is "for what, exactly?" Languages more often complement each other rather than compete with other.

1.4.1 Classification

In addition to understanding the kinds of problems a language was designed to solve, evaluation demands also that we understand *how* a language solves these problems. After all, programming languages exist so we humans can express computational ideas in ways that make sense to us. But there is a great diversity in the ways we can organize units of information and units of computation. Many of these forms can be characterized in a manner that is useful for comparing and contrasting languages. Unlike book outlines and company organization charts, we can't fit programming languages into a neat, single, hierarchical ontology. This isn't too surprising: in many knowledge disciplines, hierarchies simply don't exist. Clay Shirky makes this point beautifully in his essay *Ontology is Overrated* [102].

When there is no hierarchy, we employ *tags*. Examples of common tags used to label languages follow:

Declarative | Imperative | Applicative | Functional | Logic | Von Neumann | Object-Oriented | Expression-Oriented | Persistent | Concurrent | Reactive | Dataflow | Stack | Free-Form | Curly Brace | Visual | Compiled | Interpreted | Very High-Level | System | Scripting | Glue | Intermediate | Esoteric | Toy | Educational | Domain-specific

A given language will generally have quite a few tags. Java, for example, was introduced to the world as simple, object-oriented, familiar, robust, secure, architecture neutral, portable, high-performance, interpreted, threaded, and dynamic [33]. And several languages do their best to blend the functional and object-oriented styles.

1.4.2 Criteria

Now what might the criteria be upon which we evaluate languages? We may look at *technical* reasons:

- **Readability**: A significant portion of a programmer's time is spent trying to figure out existing code. Code must be understandable enough to be validated, fixed, or adapted to new requirements.
- **Writability**: Sometimes a language makes programming a chore, requiring one to fit a solution into the language's "opinionated" constraints. James Hague calls these *puzzle languages*, writing: "if it takes focused thought to phrase that solution into working code, you go down one path then back up, then give up, then try something completely different—then you're almost certainly using a puzzle language." [40]
- **Expressiveness**: It's quite lovely when we can express ourselves without a lot of code, such as writing `A=B+C` when A, B, and C are matrices, or when we can employ high-level constructs to make concurrent programming safe without us having to remember to acquire and release locks.
- **Guidance**: It can be very frustrating when a programmer uses a value of the wrong type, tries to use an uninitialized variable, forgets a case in a switch statement, or mis-indents, and the code happily executes—with surprising results. Some languages try to make common mistakes impossible to write, guiding you in the direction of constructs more likely to be correct.
- **Security**: The field of secure software development is concerned with ensuring external inputs cannot cause a program to divulge confidential information, corrupt data, slow to a crawl, or crash. "Safe" languages that make it easy to set and check bounds, control failures, create immutable values, and set privilege levels go a long way to minimizing **attack surfaces** that lead to costly exploits.
- **Efficiency**: Many languages have been expressly designed to allow for efficient compilation, execution, or both. Who wants to sit around waiting for code to compile,[4] or run?

[4] Obligatory xkcd link: http://xkcd.com/303/

These technical criteria comprise the core focus of this book. But let's not forget that language preference, for us human beings, is often driven *more* by "softer," criteria. Perhaps, for language X:

- It might be the only language suitable for a specific problem.
- You just like it. We all have our own preferences.
- You need to hire enough people that already know it. Some languages are very popular, some are not. The more popular a language is, the larger, too, the training set for AI code assistants is likely to be, making your organization more productive.
- There is a wealth of development tools (IDEs, fast compilers), or resources (books, blog posts, and activity on programming Q&A sites) for it.
- A government or notable company created or backed it (PL/I—IBM, Java—Sun, Go—Google, TypeScript—Microsoft, Rust—Mozilla, Swift—Apple, Verse—Epic Games).
- Everyone else is using it. There was once an old saying: "No manager ever got fired for choosing Java."
- Your company already invested too much in it, and it's too expensive to change.
- You're sticking with it because you're too tired, or don't have enough time, to learn something new.

1.4.3 Tradeoffs

When you finally go out on a limb and proclaim your love of language A, or that your team should use language B for task C, you should be ready with an evidence-based rationale, perhaps informed from the criteria above. If your decision is less than popular, you may be challenged with questions such as *why* your preferred language X does not have feature Y. The answer may be "because it has feature Z instead." For instance:

- The expressive power of dynamic typing, persistent data structures, functions as first-class values, higher-order functions, and closures (all topics we'll introduce in the book) saves development time and makes it easier to reason about program behavior, but may negatively impact runtime performance.
- Automatic garbage collection, where unreachable memory is automatically freed up without direct programmer intervention, may save thousands of hours of programmer time, but should not be used in embedded, life-critical, real-time systems. We do not want a collector to pause your program to free up space while your code is performing a time-sensitive complicated maneuver in a passenger aircraft or delivering radiation to a patient.

These tradeoffs, and others like them, tell us why there cannot be a best language for all situations. But this makes life better, because we get so many choices, as we will see throughout this book. The next fourteen chapters will take you through fourteen modern languages and the choices made by their designers. We'll then present a chapter containing brief highlights of a few dozen more. We'll see design tradeoffs time and time again, and summarize and expound on them in the book's final chapter. We hope that the book will not only dramatically increase your programming proficiency, but also provide you the tools necessary to design languages of your own. Have fun!

CHAPTER 2

Lua

Lua is a "powerful, fast, lightweight, embeddable scripting language." [67]

First appeared	1993
Creators	Roberto Ierusalimschy, Waldemar Celes, Luiz Henrique de Figueiredo
Recognized for	Tables, Interoperability with C
Notable uses	World of Warcraft, Angry Birds, Scripting
Tags	Imperative, Prototypal, Scripting
Six words or less	Lightweight, fast, powerful scripting language

Lua was born in 1993 at PUC-Rio, Pontifícia Universidade Católica do Rio de Janeiro. It has been evolving steadily but is still quite small in terms of the number of concepts and basic features. It is therefore a natural place to start our tour.

The language comes with a small number of types and operators, the usual statement repertoire (assignments, conditionals, loops, calls), flexible functions and coroutines, and a *single* data structure—the **table**—that does double duty representing both traditional arrays and key-value dictionaries. Lua's small size should not be confused with a lack of power; on the contrary, Lua is *extensible*. Lua tables can have metatables, providing the ability to customize lookup and override operators to build expressive user-defined datatypes. Lua's functions can be assigned to variables and passed to and returned from other functions. It can interoperate with other languages, including C, C++, Java, Fortran, Smalltalk, and Erlang.

As the first language on our tour, we'll use Lua to introduce many of the basic elements of programming, such as values, types, variables, expressions, operators, statements, and the all-important references. We'll cover various aspects of functions: argument passing, scope, the handling of free variables, closures, and anonymous and higher-order functions. We will then go into some depth on its signature type, the table. We'll explain how metatables provide a mechanism for creating new datatypes. We'll see how Lua allows its operators to take on new meanings under programmer control. We'll close with Lua's implementation of a popular mechanism for concurrency: coroutines.

Figure 2.1 Integer right-triangle lengths

2.1 HELLO LUA

Let's get a feel for Lua by generating some lucky numbers:

```
for c = 1, 40 do
  for b = 1, c-1 do
    for a = 1, b-1 do
      if a * a + b * b == c * c then
        print(string.format("%d, %d, %d", a, b, c))
      end
    end
  end
end
```

We're terribly sorry for the flashback to high school math, but this code serves as a more interesting opener than "Hello, world." It outputs all possible integer-valued right triangles with side lengths through size 40 (as seen in Figure 2.1).

You can run the script from the command line:

```
$ lua triple.lua
3, 4, 5
6, 8, 10
5, 12, 13
9, 12, 15
8, 15, 17
12, 16, 20
15, 20, 25
7, 24, 25
10, 24, 26
20, 21, 29
18, 24, 30
16, 30, 34
21, 28, 35
12, 35, 37
15, 36, 39
24, 32, 40
```

As mentioned in the preface of this book, we're expecting you to have programming experience, so you can probably figure out what the for-loops and the if-statement do, and surmise that `print` writes to standard output. Unlike many popular languages, though, Lua's for-loop range bounds are inclusive on both ends, so side lengths of length 40 *are* covered. String formatting interpolates the second through final arguments into placeholders in the first argument according to a given **format specifier**. The specifier `%d` in our example renders the corresponding argument as a decimal integer.

Continuing our sneak peek, here's a script that writes the permutations of its command line arguments to standard output. Permutations are generated via Heap's Algorithm [120] to illustrate that Lua is just fine with recursion:

```
function print_permutations(a, n)
  if n == 0 then
    print(table.unpack(a))
  else
    for i = 1, n-1 do
      print_permutations(a, n-1)
      local j = n % 2 == 0 and i or 1
      a[j], a[n] = a[n], a[j]
    end
    print_permutations(a, n-1)
  end
end

print_permutations(arg, #arg)
```

Here the built-in table `arg` contains the command line arguments and `#` is the length operator. The function `table.unpack` returns all of the table's elements (as Lua functions can return multiple values) and `print` outputs them separated by tabs. Parallel assignments make an appearance here too: the expressions on the right are evaluated then assigned to the variables on the left, so swapping array elements can be written concisely.

```
$ lua permutations.lua I like carrots
I       like    carrots
like    I       carrots
carrots I       like
I       carrots like
like    carrots I
carrots like    I
```

Lua can do some data science, too. In our third and final teaser example, we're given a text file containing rows of comma-separated WNBA player data encoding the team name, player name, games played, and points scored:

```
ATL,Betnijah Laney,16,263
ATL,Courtney Williams,14,193
CHI,Kahleah Copper,17,267
CHI,Allie Quigley,17,260
CHI,Courtney Vandersloot,17,225
CONN,DeWanna Bonner,16,285
CONN,Alyssa Thomas,16,241
```

```
DAL,Arike Ogunbowale,16,352
DAL,Satou Sabally,12,153
IND,Kelsey Mitchell,16,280
IND,Tiffany Mitchell,13,172
IND,Candice Dupree,16,202
LA,Nneka Ogwumike,14,172
LA,Chelsea Gray,16,224
LA,Candace Parker,16,211
LV,A'ja Wilson,15,304
LV,Dearica Hamby,15,188
LV,Angel McCoughtry,15,220
MIN,Napheesa Collier,16,262
MIN,Crystal Dangerfield,16,254
NY,Layshia Clarendon,15,188]
PHX,Diana Taurasi,13,236
PHX,Brittney Griner,12,212
PHX,Skylar Diggins-Smith,16,261
PHX,Bria Hartley,13,190
SEA,Breanna Stewart,16,317
SEA,Jewell Loyd,16,223
WSH,Emma Meesseman,13,158
WSH,Ariel Atkins,15,212
WSH,Myisha Hines-Allen,15,236
```

We wish to write a script to display the top ten scorers by points per game across all players appearing in at least 15 games:

```lua
players = {}

for line in io.lines() do
  team, name, games, points = line:match("([^,]+),([^,]+),(%d+),(%d+)")
  if tonumber(games) >= 15 then
    table.insert(players, {name=name, team=team, ppg=points/games})
  end
end

table.sort(players, function (p1, p2) return p1.ppg > p2.ppg end)

for index, player in ipairs(players) do
  if index > 10 then
    break
  end
  format = "%-22s%-4s%8.2f"
  print(string.format(format, player.name, player.team, player.ppg))
end
```

The details of this rather dense script will become apparent through the remainder of the chapter, but a few are worth a brief mention. Lines of standard input are iterated with `io.lines`; at each line, we extract the four comma-separated fields using a **pattern** and populate the `players` table with a subtable holding the name, team, and points per game

of any player appearing in 15 or more games. After sorting the table by points-per-game descending, we print the items of the table, terminating (via `break`) the loop after 10 players have been displayed. Lua, like most programming languages, features mini-languages for both pattern matching and output formatting. Our pattern for matching input lines is one or more noncomma characters (`[^,]+`), a comma, one or more noncomma characters, a comma, a sequence of one or more digits, a comma, and finally one or more digits. The output format string specifies a 22-character left-justified string field (`%-22s`), a 4-character left-justified string field (`%-4s`), and an eight-character field holding a floating point number with three digits after the radix point (`8.3f`).

Assuming the data above is stored in the file *wnba.txt*, we can use command line redirection (a feature of most standard operating system shells) to run our script:

```
$ lua top_ten_scorers.lua < wnba.txt
Arike Ogunbowale      DAL   22.000
A'ja Wilson           LV    20.267
Breanna Stewart       SEA   19.812
DeWanna Bonner        CONN  17.812
Kelsey Mitchell       IND   17.500
Betnijah Laney        ATL   16.438
Napheesa Collier      MIN   16.375
Skylar Diggins-Smith  PHX   16.312
Crystal Dangerfield   MIN   15.875
Myisha Hines-Allen    WSH   15.733
```

These three scripts were meant only to give a sneak peek of the language, leaving details to the remainder of the chapter. We now turn to our technical overview of Lua, beginning with a look at the basic language elements.

2.2 THE BASICS

Lua scripts are sequences of statements. Statements include assignments, function declarations, function calls, `if`-statements for conditional execution, blocks (introduced with the `do` keyword), and three kinds of loops: `while`, `repeat`, and `for`. The `return` and `break` statements allow exiting from functions and loops, respectively.

All values in Lua belong to one of eight types:

- **nil**, whose only instance is the value `nil`
- **boolean**, with exactly two instances, `true` and `false`
- **number**, whose instances are convenient numeric values of the underlying hardware
- **string**, whose instances are sequences of 8-bit bytes
- **table**, whose instances are key-value pairs, described in Section 2.6
- **function**, whose instances are parameterized code blocks, callable as subroutines
- **thread**, whose instances are parameterized code blocks, callable as coroutines
- **userdata**, whose instances hold data from a host language (as Lua programs are often embedded within programs written in other languages)

We can explore values and types in the Lua REPL (read-eval-print loop), accessible from the command line:

```
$ lua
> 9 ^ 2
81.0
> type(false)
boolean
> #"hello"
5
```

While we encourage you to use a REPL for exploration and practicing your skills, we'll be illustrating code with runnable scripts featuring **assertions**. When run, these scripts will either satisfy all assertions and exit cleanly or crash if any assertion were to fail. We've pre-run each of the scripts in this book, so we can be confident that the evaluations shown are correct. Here's an illustration of the Lua built-in function `type`:

```lua
function first_few_primes()
  return {2, 3, 5, 7, 9, 11, 13}
end

assert(type(7.299E-3) == "number")
assert(type(true and false) == "boolean")
assert(type("café") == "string")
assert(type(nil) == "nil")
assert(type(first_few_primes) == "function")
assert(type(first_few_primes()) == "table")
assert(type(coroutine.create(first_few_primes) == "thread"))
```

Lua (as of version 5.4) has 25 operators. Table 2.1 lists them from highest to lowest precedence, and the following script shows them in action:

```lua
assert(8 + 5 * 3 == 23)          -- illustration of precedence
assert(8 - 5 - 2 == 1)           -- illustration of left associativity
assert(3 ^ 5 == 243)             -- exponentiation
assert(19 / 5 == 3.8)            -- normal division
assert(19 // 5 == 3)             -- floor division
assert(0x1000 == 4096)           -- hexadecimal numbers
assert(9.3E-5 == 0.000093)       -- E is "times ten to the"
assert(3 ~= 5)                   -- not equals
assert(-10 <= 3)                 -- numbers can be compared
assert("dog" < "dogs")           -- so can strings
assert(#{5, 8, 13} == 3)         -- tables have a length
assert("Got" .. "7" == "Got7")   -- string concatenation
```

Though users only "see" a single `number` type, Lua uses different internal representations for integers and floating-point numbers. The operators `&`, `|`, `~`, `<<` and `>>` *only* accept integers; furthermore, arithmetic works a bit differently between the two subtypes. The technical details describing working with the two subtypes, and the rules for arithmetic, can be found in the Lua Reference Manual [68].

Operator(s)	Assoc	Description
`^`	R	exponentiation
`not # - ~`	N/A	logical negation, length, numeric negation, bit complement
`* / // %`	L	multiplication, division, floor division, modulo
`+ -`	L	addition, subtraction
`..`	R	concatenation
`<< >>`	L	left shift, right shift
`&`	L	bitwise and
`~`	L	bitwise xor
`|`	L	bitwise or
`< <= ==` `~= >= >`	L	comparison operators
`and`	L	(short-circuit) return first operand if false, else second
`or`	L	(short-circuit) return first operand if true, else second

Table 2.1 Lua operators

Strings are written as **graphemes** enclosed in single or double quotes. A grapheme is user-received unit of *writing*, made up of one or more units of textual *information*, known as **characters**. Lua implementations are likely to use the Unicode standard, which assigns to each character a name and a **code point** in the range 0..1114111.[1] A few assignments, showing code points in hexadecimal per custom, are:

U+002C	COMMA
U+13EA	CHEROKEE LETTER WE
U+20B4	HRYVNIA SIGN
U+2248	ALMOST EQUAL TO
U+30FE	KATAKANA VOICED ITERATION MARK
U+AAB9	TAI VIET VOWEL UEA
U+1201F	CUNEIFORM SIGN AK TIMES SHITA PLUS GISH
U+1D107	MUSICAL SYMBOL RIGHT REPEAT SIGN
U+1F0CE	PLAYING CARD KING OF DIAMONDS
U+1F353	STRAWBERRY

Graphemes, also known as **grapheme clusters** [20] are built up from sequences of encoded characters. Examples include:

Encoded Character Sequence	Grapheme
U+0052 LATIN CAPITAL LETTER R U+030A COMBINING RING ABOVE	Latin capital R with ring above
U+0BA8 TAMIL LETTER NA U+0BBF TAMIL VOWEL SIGN I	Tamil ni
U+1F3C4 SURFER U+1F3FE EMOJI MODIFIER FITZPATRICK TYPE-5 U+200D ZERO-WIDTH JOINER U+2640 FEMALE SIGN	Woman Surfer, medium-dark skin-tone
U+1F1F1 REGIONAL INDICATOR SYMBOL LETTER L U+1F1E7 REGIONAL INDICATOR SYMBOL LETTER B	Flag for Lebanon

[1] However, only code points in the ranges 0..55295 and 57344..1114111 are "valid." These 1112064 code points are called **Unicode scalars**.

String literals can also contain **escape sequences** for convenient representations of non-printable characters or characters that are inconvenient to type or are not available in many fonts. For example, the string literal `"\t\n"` consists of the two characters U+0009 CHARACTER TABULATION (the "tab" character) and U+000A LINE FEED. Using `\u` you can insert characters via code point: the string `"\u{1F1F1}\u{1F1E7}"` is a string containing the flag of Lebanon. It consists of one grapheme, and two code points. So what is the length of this string? Experimenting in the REPL we see it is 8:

```
> s = "\u{1F1F1}\u{1F1E7}"
> #s
8
```

Although you can write strings with graphemes, Lua strings are, internally, byte sequences. The length of the string is the number of bytes required to encode the grapheme sequence. The popular UTF-8 encoding scheme encodes code points into 1–4 bytes as follows:

Code points	Encoding
`0...7f`	$0x_7 \ldots x_0$
`80...7ff`	$110x_{10} \ldots x_6\ 10x_5 \ldots x_0$
`800...ffff`	$1110x_{15} \ldots x_{12}\ 10x_{11} \ldots x_6\ 10x_5 \ldots x_0$
`10000...1fffff`	$11110x_{20} \ldots x_{18}\ 10x_{17} \ldots x_{12}\ 10x_{11} \ldots x_6\ 10x_5 \ldots x_0$

Let's explore with a detailed example. The string literal `"Ć←afé"` contains five graphemes, (and coincidentally five code points), but is stored as a sequence of 9 bytes. We can easily work with both code points and byte values. These can be injected into strings using the `\u` and `\x` escape sequences, respectively. The code points of a string are accessed via `utf8.codepoint` and its bytes via `string.byte`. The latter two functions return each of the code points or each of the bytes within a given range of the string. Range bounds are inclusive and start at 1, so 1 through `#s` covers the entire string.

```
s = "Ć←afé"
-- Character                          CodePoint   Bytes
-- LATIN CAPITAL LETTER C WITH ACUTE       262   0xc4 0x86
-- LEFTWARDS ARROW                        8592   0xe2 0x86 0x90
-- LATIN SMALL LETTER A                     97   0x61
-- LATIN SMALL LETTER F                    102   0x66
-- LATIN SMALL LETTER E WITH ACUTE         233   0xc3 0xa9

assert(#s == 9)                 -- counts bytes
assert(utf8.len(s) == 5)        -- counts code points

-- Strings can be assembled from code points in two ways
assert(utf8.char(262, 8592, 97, 102, 233) == s)
assert("\u{106}\u{2190}\u{61}\u{66}\u{e9}" == s)

-- Or assembled from UTF-8 bytes (must be in hex)
assert("\xc4\x86\xe2\x86\x90\x61\x66\xc3\xa9" == s)

print(utf8.codepoint(s, 1, #s))     -- print each code point
print(string.byte(s, 1, #s))        -- print each byte
```

Variables *directly* hold values of the types nil, boolean, number, and string, but hold references to values of types table, function, thread, and userdata. Lua variables come in three flavors: global variables, local variables (discussed in Section 2.4), and table fields (discussed in Section 2.6). Variables *always* hold a value; there is no concept of an "undeclared" or "undefined" variable. The value of any variable that has never been previously assigned is simply `nil`.

```
print(x)              -- this does not fail!
assert(x == nil)      -- unassigned global x is simply nil
local y               -- also fine
assert(y == nil)      -- missing initializer is implicitly nil
t = {a=1, b=2}
assert(t.c == nil)    -- unassigned table fields are just nil
print(y, c, t.d)      -- happily prints three nils
```

Assignments are not limited to binding a single value. In the statement

```
x, y = 0, 0
```

we have a **name list** on the left and an **expression list** on the right. An expression list isn't a value in its own right: it just identifies expressions that are evaluated then assigned to the variables on the left side. "Extra" expressions on the right are ignored; extra names on the left become `nil`.

```
a, b = 5, 6, 7          -- extra results ignored
c, d, e, f = 5, 6, 7    -- extra vars become nil

assert(a == 5 and b == 6)
assert(c == 5 and d == 6 and e == 7 and f == nil)
```

Lua sometimes enforces the expected types of operands, but at other times will **coerce** a value of the unexpected type to a similar value of the expected type. Anything can be coerced to a boolean, with `nil` and `false` becoming false, and everything else true, though that works one-way only: `true` and `false` are not coerced to anything (not even 1 and 0, common in other languages). Strings and numbers are coercible to each other; however, the attempt to coerce a nonnumeric-looking string to a number generates an error.

```
-- strings are coerced to numbers
assert("55" + "3" == 58)     -- because + is only for numbers
assert("3" * 5 == 15)        -- because * is only for numbers

-- and numbers to strings
assert(55 .. 3 == "553")     -- because .. is only for strings

-- and anything to a boolean
assert(3 or false)
assert(0 or false)           -- yes, 0 is a true value
assert("" or false)          -- yes, the empty string is a true value
assert({} or false)          -- yes, the empty table is a true value
assert(not (nil or false))   -- Only nil and false are false values
```

```
-- The following are errors
-- false + 1                -- cannot do arithmetic on booleans
-- "the answer is" .. true  -- cannot concatenate booleans
-- false < true             -- cannot compare booleans
-- {} < {1, 2}              -- cannot compare tables
-- nil .. 3                 -- cannot concatenate nil
```

Generating errors when operands have unexpected types is known as **strong typing**; applying coercions is known as **weak typing**. While many languages try to employ strong-typing everywhere and others favor weak typing nearly everywhere, Lua is quite happy mixing things up.

2.3 FUNCTIONS

Lua functions can be named via function declarations or appear anonymously within function expressions. When called, the **arguments** are first fully evaluated and their values assigned to the called functions's **parameters**. If you pass too many arguments, the extras are ignored; pass too few and the extra parameters hold `nil`. Function values can be freely passed as arguments and returned from functions. While there is nothing particularly special about this behavior, functions that accept functions as parameters or return function values have a special name in programming language theory: they are known as **higher-order functions**.

```
-- Function values can be declared or assigned
function odd(x) return x % 2 == 1 end
square = function (x) return x * x end
twice = function (f, x) return f(f(x)) end

-- We can call anonymous functions
assert((function (x) return x + 5 end)(10) == 15)

-- We can pass function values to other functions
assert(twice(square, -3) == 81)
assert(twice(function (x) return x + 1 end, 5) == 7)

-- We can create and return new functions on the fly
function compose(f, g)
  return function (x) return f(g(x)) end
end

is_odd_when_squared = compose(odd, square)
assert(is_odd_when_squared(7))
assert(not is_odd_when_squared(0))

-- Too many arguments are fine, extras are ignored
assert((function (x) return x + 1 end)(8, 2) == 9)

-- Not passing enough arguments is fine, extras get nil
do (function (x, y) assert(y == nil) end)(1) end
```

Parameters are always freshly allocated when a function is called. They are distinct variables from the arguments, so changing a parameter will not affect its corresponding argument. However, as Lua requires that for certain types, including tables, variables hold references only, passing a table as an argument results in the parameter and argument holding a reference to the same table. Therefore *fields* of a table argument may be changed through the parameter:

```
x = {1, 2, 3}
y = {4, 5, 6}

function f(a, b)
  a = 10                  -- changing the parameter
  assert(x[1] == 1)       -- ...does NOT affect the argument
  b[2] = 20               -- but changing a field
  assert(y[2] == 20)      -- ...DOES affect the argument
end

f(x, y)
```

We can better understand the effects of passing references through the visualization in Figure 2.2. At the point of call (the left side of the figure), references to the tables are passed into parameters `a` and `b`. The function updates the value of `a` (leaving `x` unchanged), but updates a *field* of the shared table, leaving the effect visible to the argument `y` (on the right side of the figure).

Figure 2.2 Arguments passing and updating through parameters

When errors occur, whether from adding booleans, using `nil` as a function, or explicitly calling the `error` function, they are **propagated** together with some optional diagnostic information to the caller of the failing operation. If you wish to trap an error, make a **protected call**. Invoking `pcall(f,x,y)` performs `f(x,y)` and returns two values: a success indicator (`true` or `false`), and the proper result or error message. If an error is trapped, the first returned value will be `false`.

```
function add(x, y)
  return x + y
end

success, result = pcall(add, 5, 3)
assert(success == true and result == 8)  -- Success! 5 + 3 = 8

success, result = pcall(add, 5, false)
assert(success == false)                 -- Fail! Can't add booleans
assert(string.match(result, "arithmetic on a boolean"))
```

2.4 SCOPE

In language theory, the **scope** of a binding is the region of program text in which that binding is active. Informally, a language's scope rules tell us *which* entity a name refers to. The rules are needed because a name can refer to different things in different places.

As mentioned earlier, Lua variables are global, local, or table fields. Variables are global unless defined using `local` or inside a table. The reference manual tells us exactly how local variables are scoped: "The scope of a local variable begins at the first statement after its declaration and lasts until the last nonvoid statement of the innermost block that includes the declaration." [68] Blocks are the bodies of `do`, `if`, `while`, `repeat`, and `for` statements, as well as function bodies. The use of a name prior to its declaration in a block (or within the initializing expression of its declaration) will refer to a nonlocal ("outer") entity currently bound to the name, if any:

```
x = 1
do
  assert(x == 1)     -- global x because local not yet seen
  local x = x + 2    -- uses global x on right-hand side
  assert(x == 3)     -- now, FINALLY, we see the local x
end
assert(x == 1)       -- back in the global scope!
```

The declaration of a local variable with the same name as an "outer" variable currently in scope is known as a **shadow**.

Because a local variable is not in scope until *after* its declaration, the following attempt to define a factorial function fails, because the helper function, f, is not recursive:

```
factorial = function (n)
  local f = function (n, a)            -- WRONG !!!!
    return n==0 and a or f(n-1, a*n)   -- refers to GLOBAL f, crashes
  end                                  -- because nil is not callable
  return f(n, 1)                       -- calls local f
end

ok, reason = pcall(factorial, 10)      -- hoping for 3628800
assert(ok == false)                    -- but it fails!
print(reason)                          -- your output will vary
```

This can be fixed by declaring a variable then assigning the function to the variable in a subsequent statement:

```
local f
f = function (n, a) ... end
```

Because recursion is common, and the fix a little awkward, Lua provides an equivalent sugared form:

```
local function f (n, a) ... end
```

Since the scope of a variable extends to the end of its block, it by definition encompasses local functions following the declaration. This impacts the behavior of **free variables**—variables used but not declared within a function. Consider:

```
x = 'MAIN';
function second() print(x) end
function first() local x = 'FIRST'; second() end
first()
```

The variable x within `second` is free. When printed, does x take on the value from its caller, (`"FIRST"`) or from the enclosing source text (`"MAIN"`)? Languages that use caller's values for free variables are **dynamically scoped**; those that look outward to textually enclosing regions are **statically scoped**. Lua is statically scoped.

2.5 CLOSURES

The ability to nest functions inside other functions comes with interesting philosophical questions but also enables new programming techniques. Consider this script:

```
function second(f)
  local name = "new"
  f()   -- how do we handle free variables in f?
end

function first()
  local name = "old"
  print_name = function () print(name) end
  second(print_name)
end

first()
```

The function assigned to `print_name` in `first` has a free variable; it is passed to `second` (as f) and then executed, printing the value of `name`. When the function was created, it sees `name` defined within `first`, but when called (as f), might it see the `name` variable in `second`? If a system binds the free variables of a passed function after passing, we speak of **shallow binding**; if bound where the function is defined, we have **deep binding**.

What does Lua do? When a nested function with free variables is sent outside its environment, either by being passed to or returned from another function, the function carries the bindings of those variables from the enclosing environment *of its definition* with it. These

bindings "close over" the inner function (in our example, `print_name`), so the function, together with its bindings, is called a **lexical closure**, or **closure** for short. Lua thus exhibits deep binding.

Closures can be used to make **generators**.[2] A generator function produces a "next" value each time it is called. A generator for a sequence of squares would produce 0 on its first call, 1 on its second, then 4, 9, 16, and so on. Implementing a generator with a function incrementing a *global* variable prior to returning its square would be **insecure** because other parts of the code could change the global variable, disrupting future calls to the generator! Fortunately, we can use the fact that a function's local variables are invisible to code outside the function:

```
next_square = (function ()
  local previous = -1
  return function ()
    previous = previous + 1
    return previous ^ 2
  end
end)()

assert(next_square() == 0)
assert(next_square() == 1)
assert(next_square() == 4)
```

The value assigned to `next_square` is the result of calling an anonymous function. We call the right-hand side of the assignment an **immediately invoked function expression**, or IIFE. The call returns a closure. The variable that holds the number to be squared is local to the enclosing function. The generator (`next_square`) can see this value, but no other parts of the code can. The generator is secure.

2.6 TABLES

Lua provides a single structure that can be used to hold both named and numbered fields, known as the **table**. While tables may frequently contain only numbered fields or only named fields, field types can be mixed. To access table fields, place the field in square brackets. If the field name is a literal string, Lua allows a convenient dot notation.

```
colors = { "red", "green", "blue" }              -- numbered only
circle = { x = 0, y = 0, radius = 10 }           -- named only
choices = { "kim", "sam", "maria", correct = 3 } -- mixed

assert(colors[2] == "green")
assert(circle["radius"] == 10)
assert(circle.radius == 10)    -- dot notation is convenient
assert(choices[1] == "kim")
assert(choices.correct == 3)
assert(choices[choices.correct] == "maria")
```

[2] Generators are more commonly created with coroutines, but we're using a low-level approach here simply to illustrate closures in action.

Here is a contrived example to show how complex tables are created and iterated over. The functions `pairs` and `ipairs` produce **iterators**. Iterators are not tables, but rather functions. You don't call them directly; instead, you place them in your for-loop and let them work their magic.

```
producer_key = "maufacturer"

widget = {
  weight = 5.0,                    -- key is "weight"
  ["part number"] = "C8122-X",     -- key is "part number"
  "green",                         -- key is 1
  "round",                         -- key is 2
  [4] = "magnetic",                -- key is 4
  imported = false,                -- key is "imported"
  "metal",                         -- key is 3
  [producer_key] = "Acme"          -- key is "manufacturer"
}

print("pairs iterates through ALL pairs in arbitrary order")
for key, value in pairs(widget) do
  print(key .. " => " .. tostring(value))
end

print("ipairs iterates integer-keyed pairs from 1 in order")
for key, value in ipairs(widget) do
  print(key .. " => " .. tostring(value))
end
```

Fields can be specified in the form [*key*] = *value* or just *value*. The former evaluates the expression *key* and uses the result as the field's key. You can omit the brackets if the key is a simple string (letters, digits, and underscores, not beginning with a digit). Unspecified keys are auto-generated as the integers 1, 2, 3, and so on, as you likely surmised by reading through several earlier examples in this chapter.

Keys needn't be only strings and numbers—they just cannot be `nil`. Values can't be `nil` either, since `nil` is the response to requesting a value at a nonexistent key: a field value of `nil` would be indistinguishable from the key not being present. This implies that to remove an element from a table, set the value at the desired key to `nil`.

The length of a table (obtained via operator `#`) is roughly the number of contiguous integer-keyed fields starting at key 1. A table of only string-valued keys will have a length of 0; to count the number of pairs in a table, a manual count is necessary. In other words, the length operator makes sense only for tables that function as traditional lists.

As mentioned earlier, variables are not permitted to hold tables directly; they may only hold references to tables. Therefore tables are shared when assigning, passing as arguments, or being returned from functions. Sharing can be avoided by making copies of objects, but note that simply copying each pair will lead only to a **shallow copy** if a pair's value is itself a reference! Performing a **deep copy** requires a recursive walkthrough of a structure. The following script's shallow copy is illustrated in Figure 2.3:

```
a = {{x=0, y=0}, {x=8, y=21}, {x=3, y=5}}

b = a;                  -- copies the reference: b and a share

c = {}
for key, value in pairs(a) do
  c[key] = value
end                     -- makes a shallow copy of a into c

b[1] = 100
assert(a[1] == 100)     -- demonstrate a and b are shared
c[2] = 200
assert(a[2].x == 8)     -- c[2] is a copy, so a is unaffected
c[3].x = 300
assert(a[3].x == 300)   -- changed because copy was not deep
```

Figure 2.3 A Shallow copy

Lua defines a pre-existing table called the **global environment**, containing:

- A string at index `_VERSION`
- Functions, including `assert`, `pairs`, `ipairs`, `print`, `pcall`, `tonumber`, `tostring`, `require` among others
- Tables, including `coroutine`, `io`, `debug`, `string`, `utf8`, `table`, `math`, `os`, `package`, among others

When you create your own global variables, they are added to this table.

2.7 METATABLES

Tables are the go-to structure for representing structured objects in any Lua program, so we might try to use them to encode new data types. But how? The following attempt feels like something is missing—the types don't appear to be "real":

```
-- An unsatisfying way to represent datatypes
east = {type="vector", x=1, y=0}
west = {type="vector", x=-1, y=0}
me = {type="player", level=90}
world = {type="rectangle", height=5000, width=5000}
```

We would like direct language support for making our own datatypes. Lua provides the **metatable** for just this purpose. By attaching a metatable to a single table, we can customize the way the table behaves. By attaching the same metatable to multiple tables, the tables in effect become members of the same datatype. Let's see how.

To attach metatable m to a table t, invoke `setmetatable(t,m)`.[3] Every metatable contains zero or more of the following fields, which will define how t behaves:

```
__add __sub __mul __div __mod __pow __unm __idiv __band __bor __bxor
__bnot __shl __shr __concat __len __eq __lt __le __index __newindex
__call __mode __tostring __metatable __gc
```

When evaluating `x+y`, Lua first checks if x has a metatable m with an `__add` entry, and if so, produces `m.__add(x,y)`. If not, y is checked in a similar fashion. The next 18 properties work in much the same way: `__sub` is used for subtraction, `__unm` for unary minus, `__concat` for `..`, `__len` for `#` and so on.[4] The `__index` entry is magical as well: its value is a table of *additional key-value fields for the tables attached to this metatable*. That is, any table attached to a metatable with an `__index` property **inherits** these fields. For example, here are two circle objects, derived from a "prototype" unit circle:

```
unit_circle = {x = 0, y = 0, radius = 1, color = "black"}
c = {x = 4, color = "green"}
setmetatable(c, {__index = unit_circle})
assert(c.x == 4 and c.radius == 1)
```

As multiple circle objects sharing a metatable each inherit fields from a prototypal unit circle, we have the beginnings of a custom circle data type. Recall that types encapsulate both state and behavior, so let's now see how to build a complete "type" of two-dimensional vectors useful in graphics applications. We'll use metatable features to make `+` and `*` represent vector addition and vector dot product (**operator overloading**), and to make the global `tostring` render vector descriptions. Figure 2.4 provides an illustration of the variables and types from the script, where thick red arrows are used to show an object's metatable):

```
Vector = (function (class, meta, prototype)
  class.new = function (i, j)
    return setmetatable({i = i, j = j}, meta)
  end

  prototype.magnitude = function (self)
    return math.sqrt(self.i * self.i + self.j * self.j)
  end
```

[3] Actually, in Lua every value, not just every table, can have a metatable. However, setting the metatable for nontables requires the C API, which is beyond the scope of our overview.

[4] There are some restrictions here and there for some of the operator stand-ins; details are in the Lua Reference Manual.

```
  meta.__index = prototype

  meta.__add = function (self, v)
    return class.new(self.i + v.i, self.j + v.j)
  end

  meta.__mul = function (self, v)
    return self.i * v.i + self.j * v.j
  end

  meta.__tostring = function (self)
    return string.format('<%g,%g>', self.i, self.j)
  end

  return class
end)({}, {}, {})

u = Vector.new(3, 4)
v = Vector.new(-5, 10)
assert(tostring(u) == "<3,4>")
assert(tostring(v) == "<-5,10>")
assert(u.j == 4)
assert(u:magnitude() == 5.0)
assert(tostring(u + v) == "<-2,14>")
assert(u * v == 25)
```

Figure 2.4 Vectors in Lua

Each call to `Vector.new` produces a vector **instance**, namely a table with `i` and `j` components whose metatable is shared among all vector instances. The metatable contains

operator and `tostring` implementations directly, and holds a reference to a separate table for custom vector behavior—in our case the single `magnitude` property. To invoke the `magnitude` function on vector `v` we can write:

```
v.magnitude(v)
```

As the repetition of `v` is clunky, Lua allows:

```
v:magnitude()
```

as sugar for this expression, which is better because it evaluates v only once.

2.8 COROUTINES

Functions run until they terminate (normally or with via an error), then return to the caller. A **coroutine** can do the same, but also **yield**, in which case it will be *resumed* where it left off when next called. to create a coroutine, pass a function to `coroutine.create`. Coroutines can implement on-demand sequences:

```lua
nextSquare = coroutine.create(function ()
  for value = 1, 5 do
    coroutine.yield(value * value)
  end
  return "Thank you"
end)

for i = 1, 8 do
  local status = coroutine.status(nextSquare)
  local success, values = coroutine.resume(nextSquare)
  print(status, success, values)
end
```

A call of `coroutine.resume` returns a success flag (whether the coroutine has yielded or returned without error) followed by the values that were yielded or returned. The script above produces:

```
suspended   true   1
suspended   true   4
suspended   true   9
suspended   true   16
suspended   true   25
suspended   true   Thank you
dead        false  cannot resume dead coroutine
dead        false  cannot resume dead coroutine
```

A coroutine's status is one of:

- **suspended**, if it has not yet started or has yielded and is waiting to be resumed,
- **running**, if it's running (and is the caller to `status`)
- **normal**, if it has resumed another coroutine and is waiting to be resumed itself, and
- **dead**, if it has terminated via a return or unprotected error.

Rather than explicit calls to `coroutine.resume`, you will often find coroutines created with `coroutine.wrap`, which returns a function that, when called, resumes the coroutine. You won't have direct access to the coroutine in this case; instead, you will a slightly simpler interface where the function invocation produces the next yielded value.

```
nextSquare = coroutine.wrap(function ()
  for value = 1, 5 do
    coroutine.yield(value * value)
  end
  return "Thank you"
end)

for i = 1, 8 do
  local success, value = pcall(nextSquare)
  print(success, value)
end
```

Coroutines allow us to structure an application into concurrent activities, each written independently of each other. A game might contain coroutines for players, volcanos, tornados, waterfalls, and an event manager to listen for user input. The program runs on a single thread of execution: each coroutine must contain explicit `yield` and `resume` calls, and not execute too long between these calls. This architecture is known as **cooperative multitasking**, and differs from the alternative **preemptive multitasking**, where the operating system can preempt a thread at any time and give control to another thread.

As Lua is designed to be lightweight and efficient, the language does not provide direct support for preemptive multitasking. All coroutines interleave their execution on a single operating-system managed **thread**. Managing multiple "real" threads requires some overhead, which Lua prefers to avoid. One of Lua's targeted application areas is interactive gaming, where coroutines spend a fair amount of time waiting for user and system interaction—application areas in which sharing coroutines on a single thread is more performant than having the operating system juggle threads. We'll have more to say in future chapters, where we'll encounter languages that allow both preemptive and cooperative multitasking.

LUA WRAP UP

In this chapter, we were introduced to Lua. We learned that:

- Lua is a relatively small language, with only a few statements, operators, and types—and only one data structure.
- Variables store nil, booleans, numbers, and strings directly; for all other types, only references are stored.
- Both strong and weak typing are evident: anything can be coerced to booleans (`false` and `nil` to `false`, everything else to `true`), and strings and numbers are somewhat interchangeable.
- Lua functions may both accept multiple values and return multiple values. The multiple return value capability is used prominently in the `pcall` function, which is Lua's preferred mechanism for error checking and handling.

- Lua variables come in three flavors: (1) global, (2) local, and (3) table fields. Local variables are introduced with the keyword `local`. Their scope begins on the statement after their declaration and runs until the end of the block. While this rule is simple, it does mean that local functions can only be made recursive either by declaring the function prior to assignment or using the sugared form `local function`.
- Lua uses static scoping and dynamic binding.
- Tables are used for both classic arrays (1-based sequences of numerically indexed values) and dictionaries (sets of key-value pairs). Both numeric and nonnumeric indexed values can be mixed in a single table; it is up to the language implementation to keep the "array-part" access efficient.
- Lua's standard library is actually a collection of tables (e.g., `io`, `os`, `math`, etc.) These tables, along with predefined functions and variables, are stored in a special table called the global environment.
- A metatable can be attached to a table. The metatable can customize the manner in which table fields are read and written, and redefine the meaning of nineteen operators, including those for arithmetic, comparison, and logic.
- Metatables can also be used to build a prototypal inheritance mechanism, effectively adding new user-defined data types. This is done by assigning a table of the shared properties and methods to the `__index` property of the metatable.
- Lua directly supports coroutines: calling `coroutine.create` on a function object produces a coroutine object of type `thread`. Coroutines can yield and be resumed, and their status can be queried.

For further study in Lua, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Lua, see the Lua page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/lua.

EXERCISES

Now it's your turn. Continue exploring Lua with the activities and research questions below, and feel free to branch out on your own.

2.1 Find the Lua home page and the Lua Reference Manual online. Read the "About" page. Read Chapters 1–3 and 9 of the Reference Manual in full; skim other chapters according to your interests.

2.2 Find and practice with the Lua REPL. Try both simple expressions such as `3*5` as well as function definitions, calls, and loops.

2.3 What do you think about Lua's choice of allowing you to read undeclared and uninitialized variables without an error? Do you find this error-prone, or useful? Why? How does it compare to other languages you know?

2.4 Based on the precedence and associativities given in the operator table in this chapter, show the order of operations applied in the following Lua expressions: (a) `7^8*~3+5|5~3`, (b) `8*3/7^2^2*200`.

2.5 Evaluate the following in the Lua REPL. Explain any evaluations that produce an error: (a) `~3`, (b) `~3.0`, (c) `~3.5`, (d) `7^20|0`, (e) `7^200|0`.

2.6 Integers in Lua are not simply whole numbers, but rather a particular set of whole numbers in a certain range. Arithmetic on integers may result in values outside of the specified range, in which case Lua finds a suitable value within this range as the result. On one machine, for example, evaluating `2000000000000 * 2000000000000` produced the value `8015056820827586560`, while evaluating `2E12 * 2E12` produced `4e+24`. Do some research to determine why the integer expression produced the value it did.

2.7 Here is your chance to do some research into Unicode. For each of the following graphemes, give both the sequence of code points that describe the grapheme, as well as the UTF-8 byte sequence that encodes the grapheme.

a. Woman scientist with skin-tone modifier 5

b. Flag of Cape Verde

c. Saluting face

d. Head shaking vertically

e. Person with red hair

f. Man farmer, skin tone 3

2.8 Assignment in Lua is a statement, not an expression. What does this mean exactly? Why might this be a good thing? In your answer consider side-effects, and the possible confusion with equality tests were an assignment to appear in the middle of an expression.

2.9 Lua scripts are comprised of statements, which are fairly typical as high-level programming languages go: declarations, assignments, conditionals (`if`), loops (`for`, `while`, `repeat`), blocks, function calls, and disruptors (`break`, `return`, `goto`). We did not encounter all of these in this chapter. Construct example scripts in Lua using each type of statement not covered.

2.10 Write a Lua function that returns a random integer in the range 1 through 6 inclusive. Next, generalize the function to return a random integer in a user-supplied range.

2.11 In a parallel assignment statement, how does Lua handle extra variables on the left-hand side? How does it handle extra expressions on the right?

2.12 Many if not most popular programming languages feature a *conditional expression*, often written `x?y:z` which produces `y` if `x` is true, otherwise `z`. Some suggest simulating this expression in Lua with x `and` y `or` z. Is the simulation correct? If so, provide a justification for all boolean values of x, y, and z. If not, how exactly does it differ from the intent of the conditional?

2.13 What happens when a programmer intends to, but forgets, to mark a variable with `local`?

2.14 Research the issue of static vs. dynamic scoping, and list the advantages and disadvantages of each. Do any (nontoy, nonresearch) languages use dynamic scoping? If so, which ones?

2.15 Lua uses static scoping and deep binding. Do you think it could have been designed to use shallow binding instead? What would be the difficulties in implementing shallowing binding, if any?

2.16 Contrast the behavior of the following Lua and C programs. How do you think C's scoping rules are defined?

```
x = 3
function main()
  local x = x
  print(x)
end
main()
```

```
#include <stdio.h>
int x = 3;
int main() {
    int x = x;
    printf("%d\n", x);
}
```

2.17 In Lua, `function f()...end` is sugar for `f = function()...end`. Knowing this fact, run and explain the output of the following script:

```
function outer()
  function inner()
    print(1)
  end
end

outer()
inner()
```

How would this script be different if the call to `outer` were removed? Why?

2.18 Explain the difference between `pairs` and `ipairs`.

2.19 Research the table functions `table.concat`, `table.insert`, `table.remove`, `table.sort`, `table.pack`, and `table.unpack`. Write a sample script to illustrate their behavior.

2.20 Write a Lua script to list the contents of the global environment. Print each of the items, one per line, to standard output as follows: first the key, then a tab character, then the type of the value, then a tab character, then the value.

2.21 What happens when you call `setmetatable` on a number? On a boolean? On `nil`? On a thread?

2.22 Research the metatable properties `call`, `__gc`, `__mode`, and `__metatable`. Build example Lua scripts that illustrate their behavior.

2.23 Rewrite the vector example in this chapter so that the metatable of each instantiated vector is `Vector` itself (Hint: it is a very small change). What are the advantages and disadvantages of this approach?

2.24 True or false: A function is just a coroutine that doesn't call `yield`. Explain.

2.25 Modify the points-per-game program from the beginning of the chapter to use coroutines. Write one coroutine to fetch a line of text from standard input and break it up into its constituent values. Write another coroutine to add the player to the table of players appearing in 15 games or more. The first coroutine should be able to send a signal to the second to tell it there are no more lines to be read. When the second coroutine has added each player, it can call a (noncoroutine) function to sort, and output the result table.

2.26 Lua is often used in applications that are partially written in C (or C++) and partially in Lua. Find an online tutorial or reference on embedding Lua in C++ applications. Write a small example program showing how a Lua function can be called from C++, and a C++ function called from Lua.

CHAPTER 3

JavaScript

JavaScript is, by some measures, the most popular programming language in the world, powering browser-based applications, high-performance servers, and mobile applications.

First appeared	1995
Creator	Brendan Eich
Recognized for	First-class functions, Weak typing, Prototypes
Notable uses	Web application clients, Asynchronous servers
Tags	Imperative, Functional, Dynamic, Prototypal, Event-Driven
Six words or less	Not just for the web anymore

JavaScript was designed and implemented in ten days in 1995 by Brendan Eich, as an amateur-friendly scripting language embedded into a web browser. The rapid production and novice programmer focus led to some clean, expressive syntax, but also to some notorious features, including **weak typing**, where expressions of the wrong type are automatically coerced to "something that works," strings built from 16-bit character codes, and a `typeof` operator that behaves oddly. Douglas Crockford [17] has cataloged these and a few of JavaScript's other "Bad Parts" and "Awful Parts," while at the same time praising it for "[having] some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders." [17, p. 2]

In the three decades since, the language has grown extremely powerful and has been used in thousands of successful, sophisticated applications. Every major web browser comes with a JavaScript engine, and many products convert JavaScript web applications into native applications running on iOS, Android, and other mobile platforms. Beyond the browser and mobile devices, the language powers server-based applications supporting thousands of concurrent users, making JavaScript very close to a "run anywhere" language.

This chapter will have much to say about this large and sophisticated language. We'll see how higher-order functions, pattern-matching, and generators qualify JavaScript as a modern language. We'll show how prototypes and their sugared class-syntax can create user-defined datatypes, and explore the omnipresent `this` expression. We'll cover asynchronous programming in some detail, as it may be JavaScript's most signature feature.

3.1 HELLO JAVASCRIPT

Let's get a feel for JavaScript with three teaser programs, the first of which prints some lucky numbers:

```
for (let c = 1; c <= 40; c++) {
  for (let b = 1; b < c; b++) {
    for (let a = 1; a < b; a++) {
      if (a * a + b * b === c * c) {
        console.log(`${a}, ${b}, ${c}`)
      }
    }
  }
}
```

To iterate through a range of numbers, we employ a `for`-statement with three clauses in its header: the first executes before the loop begins, the second tells us what must hold to keep going, and the final clause runs at the end of each loop iteration. The spelling of the equality operator is somewhat unique: `x === y` is true if and only if x and y have the same value and the same type. We don't use the `==` operator, as it only computes whether two objects are *similar* to, rather than equal to, each other, and can behave unexpectedly (e.g., `[] == ![]` and `false == " \t "`). Within backquote-delimited strings, the construct `${`e`}` **interpolates** the stringified value of expression e into the string.

Our second teaser writes the permutations of its command line arguments to standard output. Access to the command line arguments is provided through the array `process.argv`. The first two elements (at positions 0 and 1) of this array are reserved for the program runner and the name of the program;[1] the "real" arguments therefore come from **slicing** this array beginning at index 2.

```
function printPermutations(a, n) {
  if (n === 0) {
    console.log(a.join("\t"))
  } else {
    for (let i = 0; i < n; i++) {
      printPermutations(a, n - 1)
      const j = n % 2 === 0 ? 0 : i
      ;[a[j], a[n]] = [a[n], a[j]]
    }
    printPermutations(a, n - 1)
  }
}

const args = process.argv.slice(2)
printPermutations(args, args.length - 1)
```

This example illustrates combining array elements into a string with `join`, assignment via pattern matching, and the conditional operator (`?:`). Also snuck into the example is

[1] A typical invocation for this script is `node permutations.js I like carrots`, in which we see `process.argv === ["node", "permutations.js", "I", "like", "carrots"]`.

an artifact of **automatic semicolon insertion**. JavaScript requires most statements to be terminated with a semicolon unless ending with a right brace (`}`); however, you need not include them. JavaScript will be able to insert some, but not all, of them for you. In particular, it does not insert a semicolon between the declaration of j and the array swap, since it leaves open the possibility that we actually meant to write the array indexing expression `i[a[j], a[n]]`, which while legal, is not at all what we want. We must include an explicit semicolon.

Our third teaser is the JavaScript version of the "data science" program introduced in the previous chapter on page 17:

```
import readline from "readline/promises"
const reader = readline.createInterface(process.stdin)

const players = []

for await (const line of reader) {
  let [team, name, games, points] = line.split(",")
  if (games >= 15) {
    players.push({ name, team, ppg: points / games })
  }
}
const topTen = players.sort((p1, p2) => p2.ppg - p1.ppg).slice(0, 10)
for (let { name, team, ppg } of topTen) {
  console.log(
    `${name.padEnd(22)}${team.padEnd(4)}${ppg.toFixed(2).padStart(8)}`
  )
}
```

The script instantiates a line reader object tied to standard input. Readers can be iterated with the `for await` statement to produce each line of the file. The keyword `await` is used because lines are read **asynchronously**, delivered to our program only when ready. We'll have much to say later in this chapter about JavaScript's support for asynchronous programming, which is likely one of its most recognizable features.

3.2 THE BASICS

JavaScript programs consist of scripts and modules, each containing a sequence of statements and function declarations. Statements include variable declarations, assignments, function calls, conditionals, loops, breaks, continues, returns, and a few others. The language has exactly 8 types:

- The type containing the sole value `undefined`.
- The type containing the sole value `null`.
- **Boolean**, the type containing the two values `true` and `false`.
- **Number**, the type containing all values from the IEEE 754 binary64 specification, [51] including -98.88, 22.7×10^{100}, `Infinity`, `-Infinity`, and the infamous `NaN`, the number meaning "not a number." Whole numbers can be written in binary notation, e.g., `0b01101001`, and in hexadecimal, e.g., `0x1a4`.

- **BigInt**, the type of integers whose size is bounded only by the amount of memory in the runtime system, e.g., 98123191821923151253127127538129191021932312888 82n.
- **String**, roughly, the type of character sequences. String literals are delimited by either single quotes, double quotes, or backquotes, with the latter allowed to span lines and contain interpolated expressions.
- **Symbol**, the type of symbols (not covered in this chapter).
- **Object**, the type of all other values, including arrays and functions. Objects have named properties each holding a value, for example `{x: 3, y: 5}`. Properties of an array include `0`, `1`, `2`, and so on, and `length`.

Variables and object properties hold values of nonobject types directly, but only hold references to objects, so the expression `a === b` where *a* and *b* are objects compares two references and *not* object properties.

You'll find the usual arithmetic operators for negation (`-`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`). The `Math` object provides dozens of more functions. Interestingly, the bitwise (`&`, `|`, `~`) and shift operators (`<<`, `>>`, `>>>`) *can* be applied to noninteger numbers, when doing so, the operand is coerced (somehow) to an integer in the range −2147483648 to 2147483647. Many of the numeric operators are extended to instances of BigInt, though regular numbers and BigInts cannot be mixed:

```
import assert from "assert/strict"

assert(3 + 2 ** (3 ** 2) - 10 === 505)
assert(13 / 5 === 2.6)
assert(Math.floor(13 / 5) === 2)
assert(Math.hypot(3, -4) === 5)
assert(Math.random() < 1.0)
assert((237462938920 | 0) === 1239737640)
assert(3n + 5n === 8n)
assert.throws(() => 3 + 5n, TypeError)
```

Strings can be uppercased, lowercased, split, spliced, concatenated, padded, trimmed, and searched. Strings internally use an obscure encoding known as UTF-16 [125], yielding strange results for the `length` method. The two-character string consisting of SUSHI and QUARTER NOTE has a "length" of three, despite being made up of two Unicode characters and having a length of 7 bytes when UTF-8 encoded! Obtain code points or UTF-8 encoding by converting the string into an array or buffer, respectively:

```
import assert from "assert/strict"

const s = "\u{1f363}\u{2669}" // SUSHI, QUARTER NOTE
assert(s.length === 3) // based on internal storage units
assert(Array.from(s).length === 2) // ok, 2 code points
assert(Buffer.from(s, "utf8").length === 7) // ok, 7 bytes utf-8
```

Strings natively support Unicode **normalization** to convert between different code point sequences that are semantically equivalent. For example, the sequence `U+0065` LATIN SMALL LETTER E `U+0303` COMBINING DIAERESIS represents the same "text" as the single code point

U+00EB LATIN SMALL LETTER E WITH DIAERESIS (ë). Combining many to fewer characters is called a **canonical composition** (denoted NFC in Unicode); breaking down few into many is a **canonical decomposition** (NFD). Normalization also applies between code point sequences that, while different, should behave the same in search. For example, U+01F2 LATIN SMALL LIGATURE FF (ff) has a **compatibility decomposition** (NFKD) into the two-character sequence U+0066 LATIN SMALL LETTER F U+0066 LATIN SMALL LETTER F. Similarly, the single character U+01C4 LATIN CAPITAL LETTER DZ WITH CARON decomposes via NFKD to (in order) D (U+0044), Z (U+005A), and the combining caron (U+030C). Compatibility compositions (NFKC) apply a canonical composition after a compatibility decomposition, so DŽ normalizes via NFKC to D and Ž. Let's explore how normalizations are expressed in JavaScript:

```
import assert from "assert/strict"

// Canonical composition and decomposition
const zWithDotAbove = "\u{017c}"
const zThenCombiningDotAbove = "z\u{0307}"
assert(zWithDotAbove !== zThenCombiningDotAbove)
assert(zWithDotAbove.normalize("NFD") === zThenCombiningDotAbove)
assert(zThenCombiningDotAbove.normalize("NFC") === zWithDotAbove)

// Compatibility decomposition of ligatures and archaic letters
const ffiLigature = "\u{fb03}"
assert(ffiLigature !== "ffi")
assert(ffiLigature.normalize("NFKD") === "ffi")
const longS = "\u{017f}"
assert(longS !== "s")
assert(longS.normalize("NFKD") === "s")

// NKFC is actually two steps: compatibility decomposition
// followed by canonical composition
const dzWithCaron = "\u{01c4}"
const dThenZThenCaron = "DZ\u{030c}"
const dThenZCaron = "D\u{017d}"
assert(dzWithCaron !== dThenZThenCaron)
assert(dzWithCaron.normalize("NFKD") === dThenZThenCaron)
assert(dThenZThenCaron.normalize("NFKC") === dThenZCaron)
```

Use arrays for positionally-indexed structures and objects when properties are named. Both arrays and objects can be **spread** into other arrays and objects:

```
import assert from "assert/strict"

const weekdays = ["Mo", "Tu", "We", "Th", "Fr"]
const days = ["Su", ...weekdays, "Sa"]
assert.equal(days.join(""), "SuMoTuWeThFrSa")

const center = { x: 3, y: 5 }
const circle = { ...center, radius: 13, color: "blue" }
assert.deepEqual(circle, { radius: 13, x: 3, y: 5, color: "blue" })
```

The `for-of` statement iterates array elements and object properties:

```
import assert from "assert/strict"

const colors = ["red", "green", "blue"]
const tiles = { e: 1, q: 10, k: 5 }
let scratchPad = ""
for (let color of colors) {
  scratchPad += color
}
for (let [character, score] of Object.entries(tiles)) {
  scratchPad += character + score
}
assert(scratchPad === "redgreenbluee1q10k5")
```

The value `undefined` is used when something is unknown, or known but irrelevant. Use `null` to be *explicit* about the nonexistence of something:

```
// This means my supervisor is Tina:
let supervisor = "Tina"

// This means I absolutely do not have a supervisor:
supervisor = null

// This means I might have a supervisor but know nothing
// about them, or I may or may not have a supervisor, or
// that I don't even know that I have one, or I have one
// but it is none of your business who it is:
supervisor = undefined
```

We call `undefined` and `null` the **nullish** values; they interest us because as representations of a lack of information, they cannot have properties. Accessing properties of `null` and `undefined` throws a `TypeError`:

```
import assert from "assert/strict"
assert.throws(() => null.toString(), TypeError)
assert.throws(() => undefined.toString(), TypeError)
```

In practice, nullish values often appear assigned to variables whose data is optionally present, as opposed to being required. For example, we may be interested in the population of the capital of a winner's country. But we may not have a winner, the winner may not be associated with any country, the database may not hold a country's capital, or the population may be unknown. We avoid type errors via **optional chaining**:

```
winner?.country?.capital?.population
```

where o`?.`x produces the x property of o if o exists and `undefined` if o is nullish. This makes sense: if we don't know the winner, we don't know the winner's home country.

Interestingly, variables and properties, though they both hold values, differ in one respect. Referencing an undeclared variable throws a `ReferenceError`, while referencing a missing property produces `undefined`:

```
import assert from "assert/strict"

const p = { x: 3, y: 5 }
assert(p.z === undefined) // There's no p.z
assert.throws(() => z, ReferenceError) // No variable z
```

The JavaScript runtime comes with a **global object** containing dozens of properties including `isFinite`, `String`, `Object`, and `Date`, as well as a property that references the global object itself, called `globalThis`. Properties of the global object are exactly the **global variables** of the language.

JavaScript gives us a few options for specifying function values, including the arrow (`=>`), the construct `function(`*params*`){`*body*`}`, and the standard function declaration. Functions that accept functions as parameters or return functions are called **higher-order functions**. Higher-order functions facilitate a style of coding—called **functional programming**—in which function composition replaces assignment statements and explicit loops.

```
import assert from "assert/strict"

// Three ways to introduce a function value
function square(x) {
  return x * x
}
const odd = x => Math.abs(x % 2) === 1
const lessThanTen = function (x) {
  return x < 10
}

// Essentially: { let x = 10; x + 5 }
assert((x => x + 5)(10) === 15)

// We can pass function values to other functions
const twice = (f, x) => f(f(x))
assert(twice(square, -3) === 81)
assert(twice(x => x + 1, 5) === 7)

// We can create and return new functions on the fly
const compose = (f, g) => x => f(g(x))
const isOddWhenSquared = compose(odd, square)
assert(isOddWhenSquared(7))
assert(!isOddWhenSquared(0))

// Array functions often take the place of loops
const a = [9, 7, 4, -1, 8]
assert(!a.every(odd))
assert(a.some(odd))
assert(a.every(lessThanTen))
assert.deepEqual(a.filter(odd), [9, 7, -1])
assert.deepEqual(a.map(square), [81, 49, 16, 1, 64])
```

In JavaScript, **arguments** are fully evaluated before the call and their values are assigned (copied) to the **parameters** left-to-right. If you pass too many arguments, the extras are ignored; pass too few and the extra parameters begin as `undefined`. You can, however, mark your final parameter with `...` to pack extra arguments into an array; this is known as a **rest parameter**. You may also use `...` on the argument side, where it is called the **spread operator**, to unpack an array into multiple parameters:

```
import assert from "assert/strict"

const f = (x, y) => [x, y]
const g = (x, ...y) => [x, y]

assert.deepEqual(f(1), [1, undefined]) // too few args
assert.deepEqual(f(1, 2, 3), [1, 2]) // too many args

assert.deepEqual(g(1, 2, 3), [1, [2, 3]]) // args packed
const args = [1, 2]
assert.deepEqual(f(...args), [1, 2]) // args unpacked
```

Missing arguments can take on defaults other than `undefined`:

```
import assert from "assert/strict"

function f(a, b = 3, c, d = 8) {
  return [a, b, c, d]
}

assert.deepEqual(f(1, 2), [1, 2, undefined, 8])
```

JavaScript favors **weak typing**, because more often than not, values of one type can appear where a value of another type is expected. For example, almost anything can be **coerced** to a boolean, number, or string:

- In `if` and `while` statements expecting a boolean condition, any value can appear. `0`, `null`, `undefined`, `false`, `NaN`, and the empty string act as false and are called **falsy**; all other values act as true and are called **truthy**.
- When a string is expected, `undefined` acts as `"undefined"`, `null` acts as `"null"`, `false` acts as `"false"`, `3` acts as `"3"`, and so on. To use an object x in a string context, JavaScript evaluates x`.toString()`.
- When a number is expected, `undefined` acts as `NaN`, `null` as `0`, `false` as `0`, `true` as `1`, and strings act as the number they "look like" or `NaN`. To use an object x in a numeric context, JavaScript evaluates x`.valueOf()`.

Table 3.1 provides concrete examples.

While coercions predominate, some elements of strong typing do exist: `TypeError`s do occur when using a symbol as a number, when using `null` or `undefined` as an object, when mixing numbers and bigints, or when attempting to call a nonfunction.

Value	as Boolean	as String	as Number
`undefined`	`false`	`'undefined'`	`NaN`
`null`	`false`	`'null'`	`0`
`false`	`false`	`'false'`	`0`
`true`	`true`	`'true'`	`1`
`0`	`false`	`'0'`	`0`
`858`	`true`	`'858'`	`858`
`858n`	`true`	`'858'`	*throws* `TypeError`
`NaN`	`false`	`'NaN'`	`NaN`
`'0'`	`true`	`'0'`	`0`
`'858'`	`true`	`'858'`	`858`
`''`	`false`	`''`	`0`
`'dog'`	`true`	`'dog'`	`NaN`
`Symbol('dog')`	`true`	`'Symbol(dog)'`	*throws* `TypeError`
any object x	`true`	*result of* x`.toString()`	*result of* x`.valueOf()`

Table 3.1 JavaScript automatic type conversions

3.3 DESTRUCTURING

How do names receive values in a program? For many novice programmers, the answer "Assignment statements!" may be the first thing to pop into their heads. With a little thought, passing arguments to parameters in function calls may come up, too. But assignment and parameter passing are simply specific instances of the generalized concept of **binding**, where variables take on values based on their appearance in some matching expression—you may have noticed that `for-of` statements introduce bindings as well. How might this look in general? Here's a hypothetical construct:

```
{x:1, y: [z, "hello"]} <=> {y: [2, q], x: p}
```

The idea is to match up a variable (on either side) with the corresponding value on the other side. Our example would bind 1 to p (look at the values of the x keys on both sides), 2 to z (first element of the y-arrays), and `"hello"` to q. JavaScript doesn't go quite so far as binding variables on both sides, but it does allow an expressive **pattern** on the left. The entire right-hand side is evaluated first (as is the case in classic assignment), then the variables on the left become bound. Some examples follow:

```
import assert from "assert/strict"

let [x, y] = [1, 3]
let { a, b: z, c } = { a: 5, b: 8, c: 13 }
// Note the shorthand property syntax: {a} means {a:a}

assert.deepEqual([x, y, a, z, c], [1, 3, 5, 8, 13])
```

This pattern-matching mechanism is called **destructuring assignment** because the runtime breaks up (destructures) the pattern to find the variables to be bound.

Destructuring is particularly powerful and useful in argument passing! Using an object pattern as a sole parameter gives you the effect of indexargument labelargument labels

at the point of call, which not only increases readability but gives you some flexibility in ordering the arguments:

```
function drawLine({ from: p, to: q, withStyle: style }) {
  // Code using p, q, and style
}

drawLine({ from: [3, 5], to: [8, -13], withStyle: "dashed" })
drawLine({ to: [3, 55], from: [8, 1], withStyle: "dotted" })
drawLine({ withStyle: "solid", from: [1, 2], to: [8, 21] })
```

3.4 GENERATORS

JavaScript's treatment of free variables and lexical closures is similar to Lua's, which we discussed in the previous chapter. Static scoping and deep binding are the law of the land. So we could, in principle, implement secure objects to generate values on demand using closures, as we did in Lua on page 28.

But just as in Lua, it is more common to use a higher-level structure for on-demand sequence generation. A JavaScript generator is a function built from a `function*`, that suspends execution via a `yield` expression and resumes where it left off when called again. Here's the JavaScript version of the first-five-squares generator we previously built in Lua on page 33. Observe the three ways to use JavaScript generators:

```
import assert from "assert/strict"

function* firstFiveSquares() {
  for (let i = 1; i <= 5; i++) {
    yield i ** 2
  }
}

// Three ways to use a generator

// 1. Obtain the values on demand
const g = firstFiveSquares()
assert.deepEqual(g.next(), { value: 1, done: false })
assert.deepEqual(g.next(), { value: 4, done: false })
assert.deepEqual(g.next(), { value: 9, done: false })
assert.deepEqual(g.next(), { value: 16, done: false })
assert.deepEqual(g.next(), { value: 25, done: false })
assert.deepEqual(g.next(), { value: undefined, done: true })

// 2. Use for-of to iterate through generated values
for (let s of firstFiveSquares()) {
  console.log(s)
}
```

```
// 3. Roll up generated values all at once
assert.deepEqual([...firstFiveSquares()], [1, 4, 9, 16, 25])
```

3.5 METHODS

An object can have properties whose values are functions:

```
const circle = {
  radius: 10,
  area: function () {
    return Math.PI * this.radius * this.radius
  },
  circumference: function () {
    return 2 * Math.PI * this.radius
  },
  expand: function (scale) {
    this.radius *= scale
  },
}
```

When we call a function via property access notation (e.g., `circle.area()`), we say the function is a **method** and the object is the **receiver**. Using the long syntax for function values (`function (` *params* `) {` *body* `}`) enables the special expression `this` to refer to the receiver. Functions defined with `(` *params* `) => {` *body* `}`, or **arrow functions**, do not get a special `this`. Because the use of long-form, or **nonarrow**, function values is so common within JavaScript object literals, a shorthand notation is available:

```
const circle = {
  radius: 10,
  area() {
    return Math.PI * this.radius * this.radius
  },
  circumference() {
    return 2 * Math.PI * this.radius
  },
  expand(scale) {
    this.radius *= scale
  },
}
```

3.6 PROTOTYPES

We've just seen how to create *one* circle object. To create *several* circle objects in JavaScript, we start with an initial object then *derive* additional objects from it. These new objects have the original object as their **prototype**. Prototypes allow us to **inherit** properties: when we encounter the expression $q.x$, we look for an x property in q. If found, we have our value; if not, we look for the property in q's prototype, and if necessary, in the prototype's prototype, and so on, until we find the property or reach the end of the "prototype chain." If no object on the chain has the property, the lookup produces `undefined`.

```
import assert from "assert/strict"

const unitCircle = {
  x: 0,
  y: 0,
  radius: 1,
  color: "black",
  area() {
    return Math.PI * this.radius * this.radius
  },
  circumference() {
    return 2 * Math.PI * this.radius
  },
}

const c1 = Object.create(unitCircle)
c1.x = 3
c1.color = "green"

const c2 = Object.create(unitCircle)
c2.radius = 5

const c3 = Object.create(unitCircle)

assert.deepEqual([c1.x, c1.y, c1.radius, c1.color], [3, 0, 1, "green"])
assert.deepEqual([c2.x, c2.y, c2.radius, c2.color], [0, 0, 5, "black"])
assert.deepEqual([c3.x, c3.y, c3.radius, c3.color], [0, 0, 1, "black"])
assert(c2.area() === 25 * Math.PI)
assert(c3.area() === Math.PI)
```

The expression `Object.create(unitCircle)` creates a new object whose prototype is `unitCircle`. Our script creates a black circle of radius 1, centered at the origin, as the prototype of three other circles. JavaScript's **delegation** of property lookup conveniently gives each circle four properties, without the need to store all four properties in each object.

In the object referenced by `c1`, `x` and `color` are called **own properties**, while `y` and `radius` are called **inherited properties**.

Note that in each of the newly created objects, we store only those properties whose values *differ* from those in the prototypal circle. Thus we can have each circle inherit all of the methods from the prototype: since every circle computes its area and circumference the same way, it would be wasteful to store copies of these functions in each circle.

Figure 3.1 shows the four circles and the prototype links to `unitCircle`. For simplicity, our diagram omits the prototype links from `unitCircle` to its prototype, and from the two functions to their prototype.

In practice, programmers will want a function to construct each instance of a family of objects that each share a prototype. Interestingly, JavaScript provides a mechanism to do just that. We'll illustrate the technique with an example, and then discuss:

Figure 3.1 Objects sharing a prototype

```
import assert from "assert/strict"

function Circle(centerX = 0, centerY = 0, radius = 1, color = "black") {
  this.x = centerX
  this.y = centerY
  this.radius = radius
  this.color = color
}

Circle.prototype.area = function () {
  return Math.PI * this.radius * this.radius
}

Circle.prototype.circumference = function () {
  return 2 * Math.PI * this.radius
}

const c = new Circle(1, 5)
assert(c.x === 1 && c.y === 5 && c.radius === 1 && c.color === "black")
assert(c.area() === Math.PI)
assert(c.circumference() === 2 * Math.PI)
assert(Object.getPrototypeOf(c) === Circle.prototype)
assert(c.constructor) === Circle
assert(typeof c === "object")
```

This simple looking script illustrates a lot of JavaScript magic. Every JavaScript function has three properties, `name`, `length` (the number of parameters), and `prototype`, the object that will be assigned as the prototype of all objects created by calling the function with the operator `new`. It's primed for you with a `constructor` property, referencing the function. This allows you to determine the function that created an object as a kind of run time "type check," since, as the last line of our example shows, `Circle` isn't a JavaScript type. Figure 3.2 illustrates the relationship between these objects.

Figure 3.2 Construction of Objects using `new`

The declaration `let c = new Circle(1, 5);` creates a new object with prototype `Circle.prototype`, passes it to the function `Circle` as `this`, and binds the now-initialized object to the variable *c*. The area and circumference methods are stored as properties in the prototype, just as in our earlier example.

Creating a constructor function to build **instances** of a user-defined type, and loading up the shared state and behavior into a common prototype occurs so often in JavaScript that there is a shorthand syntax for this pattern:

```
import assert from "assert/strict"

class Circle {
  constructor(centerX = 0, centerY = 0, radius = 1, color = "black") {
    this.x = centerX
    this.y = centerY
    this.radius = radius
    this.color = color
  }
  area() {
    return Math.PI * this.radius * this.radius
  }
  circumference() {
    return 2 * Math.PI * this.radius
  }
}
```

```
const c = new Circle(1, 5)
assert(c.circumference() === 2 * Math.PI)
assert(typeof Circle === "function")
```

The last line of the script above shows us that the `class` keyword *does not* create a "class object"—`Circle` is still a function—and the "typecheck" `c.constructor === Circle` still holds. The class construct is **syntactic sugar**—a syntax that makes the standard form easier to read. It does no more and no less than defining the function and assigning methods to the prototype.

The class syntax makes it convenient to express **IS-A** relationships, which we can illustrate with some friendly animals. All animals have a name, and when asked to speak, they say their name and make a species-specific sound. Animals come in three kinds: cows, horses, and sheep:

```
import assert from "assert/strict"

class Animal {
  constructor(name) {
    this.name = name
  }
  speak() {
    return `${this.name} says ${this.sound()}`
  }
}

class Cow extends Animal {
  sound() {
    return "moooo"
  }
}

class Horse extends Animal {
  sound() {
    return "neigh"
  }
}

class Sheep extends Animal {
  sound() {
    return "baaaa"
  }
}

const h = new Horse("CJ")
assert(h.speak() === "CJ says neigh")
const c = new Cow("Bessie")
assert(c.speak() === "Bessie says moooo")
assert(new Sheep("Little Lamb").speak() === "Little Lamb says baaaa")
```

Read the `extends` keyword as "is a"—that `Horse` extends `Animal` means a horse *is an* animal. Since an animal can speak, so can a horse. The internal workings of the class extension is quite elegant: `Animal.prototype` is simply made the prototype of `Horse.prototype` (see Figure 3.3), so horses naturally speak.

Figure 3.3 Implementation of IS-A in JavaScript

Properties whose names begin with a `#` are not visible outside the class, and are often used to **hide** an object's internal representation, and define methods that should not be part of a public interface. In the following example, queue objects can only be added to and removed from with the `add` and `remove` methods; no outside code can maliciously or accidentally modify the **private** `#data` array directly, as doing so would violate the first-in-first-out protocol expected of a queue.

```
class Queue {
  #data
  #validateNotEmpty() {
    if (this.size <= 0) throw new Error("Queue empty")
  }
  constructor() {
    this.#data = []
  }
  add(item) {
    this.#data.push(item)
  }
  remove() {
    this.#validateNotEmpty()
    return this.#data.shift()
  }
  peek() {
    this.#validateNotEmpty()
```

```
    return this.#data[0]
  }
  get size() {
    return this.#data.length
  }
}
```

3.7 SCOPE

JavaScript has a *lot* of scope rules, influenced by whether variables are truly global (properties of the global object), defined at the top level of a module, local to a function or block, or declared with `let`, `const`, or the obsolete `var`. We don't have the space to cover every rule, but we do wish to highlight a significant difference from Lua. Scopes do not begin at the point of declaration; rather, bindings introduced with `var` are scoped to the *entire* innermost function, and bindings introduced with `let` or `const` are scoped to the *entire* innermost enclosing block.

With such wide scopes, you may ask: what if one tries to use a variable in its scope but prior to its declaration—in its so-called **temporal dead zone** (TDZ)? The following exploration illustrates that the TDZ rules differ between `var` and `let`/`const`:

```
import assert from "assert/strict"
const [a, b] = [1, 2]

;(function () {
  assert(a === undefined) // the local `a` is in scope
  assert(b === 2) // we see the outer `b`
  if (true) {
    var a = 100 // scoped to whole function!
    let b = 200 // scoped only inside this block
    const c = 300 // scoped only inside this block
  }
  assert(a === 100) // it's been initialized
  assert(b === 2) // outer, because local used `let`
  assert.throws(() => c) // there's no `c` out here at all
})()
```

We see that temporal dead zone access for `var` produces `undefined` and for `let` and `const` a `ReferenceError` is thrown.[2]

3.8 PROMISES

For programs to run efficiently, they should not stop and wait for users to click buttons, move cursors, or enter keyboard data. They should not stop and wait for file and network accesses to complete. How can a language guarantee programs never wait idly, and stay running while a user ponders their next action or a network request has yet to be answered? That is, how can we specify that computation is **asynchronous**?

[2]Since the introduction of `let` and `const` in 2015, there is no reason to ever use the `var` keyword, but it is quite prominent in old JavaScript code and remains an official part of the language.

In an asynchronous architecture, code responds to **events** such as button clicks, cursor movements, or data becoming ready from a file. When fired, an event is added to the **event queue**. When an event is eventually pulled from the queue, it is passed to any **event handler** function that has been **registered** with the type of the event. In a web browser, handlers are registered on document objects with the `addEventListener` method. The following web document contains a little canvas you can sketch in (run it by giving this file an .html extension and simply opening the file in a browser):

```
<canvas width="500" height="500"></canvas>
<script type="module">
  const canvas = document.querySelector("canvas")
  const ctx = canvas.getContext("2d")
  let drawing = false
  canvas.style.border = "2px solid purple"
  canvas.addEventListener("mousedown", e => {
    drawing = true
    ctx.moveTo(e.offsetX, e.offsetY)
  })
  canvas.addEventListener("mousemove", e => {
    if (drawing) {
      ctx.lineTo(e.offsetX, e.offsetY)
      ctx.stroke()
    }
  })
  const stopDrawing = e => (drawing = false)
  canvas.addEventListener("mouseup", stopDrawing)
  canvas.addEventListener("mouseout", stopDrawing)
</script>
```

Mouse event objects hold the cursor position in the `offsetX` and `offsetY` properties; touch events for phones and tablets work in a similar fashion. Each of the handler functions is run to completion, one after the other. That is, all the JavaScript handlers "run on the same thread" though they may run in parallel with any operating system-level operations.

In practice, we do not always register events explicitly—many functions only "kick off" a single long-running operation and immediately return a **promise** object. Generally, we attach success and failure **callbacks** to the promise object to run after the long-running operation completes. Node's standard filesystem module (`fs/promises`) has many functions that work this way. For a trivial example, let's print the contents of a file:

```
import { readFile } from "fs/promises"

const promise = readFile("/etc/profile", "utf8")
promise.then(
  data => {
    console.log(data)
    console.log("This is printed only AFTER the file contents")
  },
  error => console.error(`${error}`)
```

```
)
console.log("This is always printed BEFORE the file contents")
```

The function `readFile` kicks off the reading operation and *does not wait for the read to complete*. Though the code following the call and the actual reading take place concurrently, the code following the call will run to completion *before* the JavaScript engine examines the event queue. Therefore, the callbacks do not run until the file has been read *and* the currently executing handler has finished. The example script emphasizes this important JavaScript behavior with its "printed before" message.

A promise object begins life in the pending state and eventually **settles** to either the fulfilled or rejected state. Invoking `p.then(f,g)` registers f to run after promise p enters the fulfilled state and g after it enters the rejected state. The invocation `p.catch(g)` registers an error callback only. Both `then` and `catch` return promises, so asynchronous operations can be chained elegantly. Promise rejections propagate through the promise chain and only need to be handled once, at the end of the chain. As a contrived example, we'll create a folder, copy the file */etc/profile* into this folder, and print the copied file:

```
import { copyFile, mkdir, readFile } from "fs/promises"

mkdir("backups")
  .then(() => copyFile("/etc/profile", "backups/profile"))
  .then(() => readFile("backups/profile", "utf8"))
  .then(data => console.log(data))
  .catch(error => console.error(`${error}`))
```

A promise-returning function can be invoked with the `await` expression. This effectively treats the code following the invocation as a success callback, and, if invoked within a `try` statement, the `catch` clause as a failure callback:

```
import { copyFile, mkdir, readFile } from "fs/promises"

try {
  await mkdir("backups")
  await copyFile("/etc/profile", "backups/profile")
  const data = await readFile("backups/profile", "utf8")
  console.log(data)
} catch (error) {
  console.error(`${error}`)
}
```

You may recall the teaser program early in this chapter, where successive lines of the file were read. Rather than explicitly await each line, JavaScript's `for-await` statement allowed a much cleaner expression of the sequencing of asynchronous operations.

For web applications, browser-side JavaScript has a global function `fetch` whose simplest invocation returns, for a given URL, a promise that settles to an HTTP response[3]. Responses can be parsed with various methods depending on the type of response data. Most web services return data in the JSON format [52]. The response's `json()` method parses the all-text HTTP response body into an object. Parsing is a potentially long-running operation

[3]See [109] for detailed information on the HTTP protocol.

so it too returns a promise. The typical pattern for retrieving data from a Web API thus features a promise chain, as illustrated in the following HTML document that displays an image of Pikachu retrieved from the free and open Pokémon API:

```
<img alt="pikachu" />
<script type="module">
  const picture = document.querySelector("img")
  const url = "https://pokeapi.co/api/v2/pokemon/pikachu"
  const response = await fetch(url)
  const pokemon = await response.json()
  picture.src = pokemon.sprites.front_default
</script>
```

We've only scratched the surface of JavaScript asynchronous programming. In the end-of-chapter exercises you will research and explore questions such as "how do I execute code after all of a set of asynchronous operations finishes?" or "how do I execute code after just one of a set of asynchronous operations finishes?" We'll also have the chance in future chapters to look at other mechanisms to describe asynchronicity.

JAVASCRIPT WRAP UP

In this chapter, we were introduced to JavaScript. We learned that:

- All modern web browsers run JavaScript; in addition, engines such as Node.js run JavaScript on the command line or server. Scripts are either classic scripts or module scripts. Modules run in the safer "strict mode" and allow several features unavailable in classic scripts.
- JavaScript expressions manipulate values of exactly eight types: Undefined, Null, Boolean, Number, BigInt, String, Symbol, and Object. Variables store values of the first seven types directly, but store references to objects.
- JavaScript generally employs weak typing. Values coerced to false are called **falsy**; all other values are called **truthy**. The only falsy values in JavaScript are `undefined`, `null`, `0`, `NaN`, the empty string, and the empty object.
- The values `null` and `undefined` are called **nullish** and are frequently operated upon by the `?.` and `??` operators.
- When calling functions, arguments are fully evaluated and then passed to parameters by copying values. Extra arguments are ignored; extra parameters are assigned `undefined`, or a default value if specified.
- Variable bindings can be created not only by simple assignment and pattern passing but by a rich pattern-matching facility known as destructuring.
- JavaScript is statically scoped and uses deep binding. It supports both function-scoped (`var`) and block-scoped (`let`, `const`) bindings.
- Object properties are either **own properties** or **inherited properties**. Property lookup traverses the prototype chain, if required.
- The value of the expression `this` is context-dependent, and is frequently seen in method definitions and constructors.

- The keyword `class` provides sugar for defining a constructor function and methods that populate a prototype object.
- The design of JavaScript facilitates asynchronous, event-driven programming. Events are fired by actions within a web browser as well as within server applications dealing with system resources and networking. Events are generally handled with promises with explicit promise methods or the `await` keyword.

For further study in JavaScript, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using JavaScript, see the JavaScript page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/js.

EXERCISES

Now it's your turn. Continue exploring JavaScript with the activities and research questions below, and feel free to branch out on your own.

3.1 Find out how to execute JavaScript code in your web browser's "Developer Tools."

3.2 If you have not already done so, install Node.js on your machine. Run the first two examples from this chapter from the command line.

3.3 Evaluate the following JavaScript expressions in the Node REPL or browser console: (a) `"16" == 16`, (b) `16 == "0x10"`, and (c) `"0x10" == "16"`. Were the results surprising? Why or why not? If you said they were not, why might a reasonable person be surprised? (Hint: Consider transitivity.)

3.4 Using a REPL, browser console, web search, or a suitably trained AI chatbot, determine the difference between `a.reverse()` and `a.toReversed()` for some array `a`. Explain the difference in two ways: first to someone that is not an experience programmer, and then using technical language.

3.5 Suppose we had the following declaration `let a = {x: 1, y: {a: 2, b: 3}}`. The spread operator introduced in this chapter makes a shallow copy, while the built-in `structuredClone` function makes a deep copy. Sketch a diagram showing the result of executing `let b = [...a]; let c = structuredClone(a);`.

3.6 Find out how to list all of the properties of the global object in your favorite JavaScript environment.

3.7 Define a JavaScript function `raise` that can be called either with `raise({ theValue: 3, toThePower: 5 })` or `raise({ toThePower: 5, theValue: 3 })`. In the body of your function definition, use `base` and `exponent` in your computation.

3.8 The expression `winner?.country?.capital?.population` encountered in this chapter can be written without the `?.` operator as follows:

```
const cityName = (() => {
  if (employee === null || employee === undefined) {
    return undefined
  }
  const _supervisor = employee.supervisor
```

```
  if (_supervisor === null || _supervisor === undefined) {
    return undefined
  }
  const _city = _supervisor.city
  if (_city === null || _city === undefined) {
    return undefined
  }
  return _city.name
})()
```

Can operator `?.` be faithfully simulated with a shorter expression?

3.9 What are the values of the variables `mountain`, `lat`, and `lon` after the following binding (use a REPL to test your answer):

```
let {
  place: {
    name: mountain,
    loc: [lat, lon],
  },
} = {
  place: { name: "Sagarmatha", loc: [27.9881, 86.9253] },
}
```

3.10 `Let`-declarations in `for` loops introduce a new binding for *each execution* of the loop, while `var`-declarations in `for` loops give the loop variable function scope. Given this knowledge, predict the output of this script, before executing it:

```
let [a, b] = [[], []]
for (var x of [1, 2, 3]) {
  a[x] = () => x
}
for (let y of [1, 2, 3]) {
  b[y] = () => y
}
console.log(a[1]())
console.log(b[1]())
```

Execute the script to see if your analysis was correct.

3.11 Rewrite the first five squares generator from this chapter using a closure and an IIFE (similar to the Lua implementation in the previous chapter). Do you find this version more or less readable than the generator?

3.12 The expression `this` can take on the value of our choosing via `call`, `apply`, and `bind`:

```
import assert from "assert/strict"

function talkTo(message, suffix) {
  return message + ", " + this.name + suffix
}
```

```
const alice = { name: "Alice", address: talkTo }
const bob = { name: "Bob" }

assert(alice.address("Hello", ".") === "Hello, Alice.")
assert(alice.address.call(bob, "Yo", "!") === "Yo, Bob!")
assert(alice.address.apply(bob, ["Bye", "..."]) === "Bye, Bob...")
assert(alice.address.bind(bob)("Right", "?") === "Right, Bob?")
```

Research these methods and explain the behavior of each in your own words.

3.13 For a function f, object o, and expression e, show that the expressions `f.call(o,e)` and `f.bind(o)(e)` produce the same result. What, then, can `bind` do that `call` cannot? Hint: Look up the term "partial application."

3.14 What object is assigned as the prototype of all objects created with the object literal syntax (e.g., `{}`)? Which object is assigned as the prototype of all functions? What properties do these objects have? What are the prototypes of those prototypes?

3.15 Find out about the method `Object.assign` and use it in place of the direct assignments to `Circle.prototype` in the example script from the chapter that introduced the `new` operator.

3.16 Complete Figure 3.1 to show the prototypes of the unit circle object and its two methods. Show the prototypes of the prototypes where they exist.

3.17 In the example at the end of Section 3.6, we see a constructor function designed to be called with operator `new`. What happens if it is called without `new`?

3.18 Add to the Queue class in this chapter a private method called `#validateNotFull` which throws an error if the queue's current size is already at some maximum. Invoke this method from the `add` method.

3.19 We saw, but did not explain, the array methods `every`, `some`, `filter`, and `map`. Each have an optional second argument. What is this second argument and why is it needed? Write a script that illustrates the need.

3.20 Try out this (presumably incorrect) script in a browser:

```
// BAD CODE! Using var alerts 10 for every button!
for (var i = 0; i < 10; i++) {
  const button = document.createElement("button")
  button.innerHTML = i
  button.addEventListener("click", e => {
    alert(i)
  })
  document.body.appendChild(button)
}
```

What happens when each button is pressed? Why? Does changing `var` to `let` change the behavior? Why or why not?

3.21 Enhance the sketching program in this chapter so that double-clicking on the canvas clears the canvas. If you feel ambitious, add a slider to your application for changing the line thickness, or a color selector for changing the line color.

3.22 In this chapter we saw how file operations from the Node library `fs/promises` return promises for asynchronous file operations. The library `fs` contains file operations in which an explicit callback is passed as an argument and invoked upon completion of the asynchronous operation. Describe the following program, using precise technical language to convey the intended high-level effect of the code. Do not hesitate to visit the documentation!

```
import fs from "fs"

fs.readFile(process.argv[2], (error, data) => {
  if (error) {
    console.error(`${error}`)
    process.exit(1)
  }
  const counts = new Map()
  const pattern = /[\p{L}'']+/gu
  for (let [word] of `${data}`.toLowerCase().matchAll(pattern)) {
    counts.set(word, (counts.get(word) ?? 0) + 1)
  }
  for (let [word, count] of [...counts.entries()].sort()) {
    console.log(`${word} ${count}`)
  }
})
```

3.23 Rewrite the small Pikachu-displaying web app from this chapter to display the images of 5 Pokemon of your choice, using promise objects directly (rather than `await`.) Use the `Promise.allSettled` function to produce a fancy effect of your choice (confetti, fireworks, flashing the background, or whatever your imagination and web-design research skills lead you to perform) *after* all of the images have been loaded.

3.24 Repeat the previous question using the `for-await` statement.

3.25 One restriction of the `await` expression not mentioned in the chapter proper is that it can only appear at the top level of a module or within a function marked `async`. Research the `async` keyword. How does marking a function `async` change its behavior?

3.26 Write a small web application that fetches the image of various Pokemon whose names are selected from a dropdown element or text field. Use `await` in your fetch expression. Since the images will be fetched in response to a user action, the event callback will need to be marked `async`.

3.27 How does one kick off a number of asynchronous operations and invoke an operation after the first promise settles?

CHAPTER 4

TypeScript

TypeScript is a superset of JavaScript with the capability to detect type errors before a program is ever run.

First appeared	2012
Creator	Microsoft
Recognized for	Adding static types to JavaScript
Notable uses	Web application clients
Tags	Imperative, Functional, Event-Driven
Six words or less	"Typed JavaScript at Any Scale"

JavaScript's status as one of the world's most popular languages has led to a variety of so-called "compile-to-JavaScript" languages that seek to improve upon JavaScript while still making use of the massive ecosystems of the browser, Node.js, and NPM. Several of these, including CoffeeScript, Elm, and PureScript, depart widely from JavaScript's syntax. TypeScript takes a different approach: it is a pure superset of JavaScript. Every JavaScript program is also a TypeScript program. If you know JavaScript, you can start using TypeScript right away.

As you can guess from the name, TypeScript's additions to JavaScript mainly involve support for types. Programmers can annotate their code with type expressions, allowing the compiler to check the type safety of most (or sometimes all) expressions before the program is even run. The type system is quite rich, with support for interfaces, generics, union types, and more. TypeScript's ability to infer types is quite impressive, even adjusting type constraints on variables depending on a program's control flow. In practice, you tend to get away with far fewer type annotations than you might expect.

In this chapter, we'll tour TypeScript the language, making heavy use of what we learned in our study of the language of the previous chapter. After our standard teaser programs, we'll provide an overview of the type system, followed by coverage of selected topics, including type validation, type annotation, the flexibility one has in typing objects, contextual typing with narrowing and widening, type guards, generics, index signatures, utility types, and type assertions.

4.1 HELLO TYPESCRIPT

Before getting to the technical overview, we'll look at our usual three example scripts. First up is our script to generate integer right triangle measurements.

```
for (let c = 1; c <= 40; c++) {
  for (let b = 1; b < c; b++) {
    for (let a = 1; a < b; a++) {
      if (a * a + b * b === c * c) {
        console.log(`${a}, ${b}, ${c}`)
      }
    }
  }
}
```

Exactly like JavaScript! TypeScript is, after all, a superset of JavaScript so every JavaScript program is also a TypeScript program. However, TypeScript will, should you choose to allow it, analyze your program for potential errors—especially type errors—before your program is even run!

Having type checks performed before execution saves the programmer from having to track down type clashes that occur at run time—or worse, subtle errors that arise from JavaScript's silently applied coercions. But to ensure all JavaScript programs are TypeScript programs, the programmer must *opt-in* to type checks, and different degrees of checking are possible. In this book, we will always assume that we have opted into the maximal degree of type analysis.[1]

With type analysis turned on, variables are constrained to hold values of a certain type. In the triple script above, TypeScript noticed that we initialized a, b, and c with values of type `number`, then verified that all subsequent uses of these variables were consistent with the number type. Inferring type constraints is neither required nor always possible, so TypeScript permits **type annotations**, as shown in our second teaser:

```
function printPermutations(a: string[], n: number) {
  if (n === 0) {
    console.log(a.join("\t"))
  } else {
    for (let i = 0; i < n; i++) {
      printPermutations(a, n - 1)
      const j = n % 2 === 0 ? 0 : i
      ;[a[j], a[n]] = [a[n], a[j]]
    }
    printPermutations(a, n - 1)
  }
}

const values = process.argv.slice(2)
printPermutations(values, values.length - 1)
```

[1] See [74] for a detailed look at the various levels of type checking and how to employ them.

Annotating a function's parameters and return type is considered a best practice as it adds documentation regarding the function's intent. Parameter types are not always inferable anyway, so annotating *all* parameters gives your entire script a consistent look and feel. For local variables initialized at the point of declaration, however, annotations are generally omitted when the inferred type is more-or-less obvious.

Our third teaser adds a couple annotations to its JavaScript counterpart and introduces a TypeScript-only construct, the interface:

```
import * as readline from "readline"
const reader = readline.createInterface(process.stdin)

interface Player {
  name: string
  team: string
  ppg: number
}
const players: Player[] = []

for await (const line of reader) {
  let [team, name, games, points] = line.split(",")
  if (Number(games) >= 15) {
    players.push({ name, team, ppg: Number(points) / Number(games) })
  }
}
const topTen = players.sort((p1, p2) => p2.ppg - p1.ppg).slice(0, 10)
for (let { name, team, ppg } of topTen) {
  console.log(
    `${name.padEnd(22)}${team.padEnd(4)}${ppg.toFixed(2).padStart(8)}`
  )
}
```

Here any object with a string `name` property, a string `team` property, and a number `ppg` property conforms to the `Player` interface and can be added to the `players` array, as this array is explicitly constrained to hold `Player` elements.

Our technical overview of TypeScript follows. As we covered JavaScript in the previous chapter, our overview will focus exclusively on the differences between the two languages.

4.2 THE BASICS

TypeScript values belong to same the eight types as JavaScript. These types are accessible to the program at runtime, and are thus known as **dynamic types**. TypeScript works with a large set of *additional* types at compile-time, known as **static types**. These include:

- **Literal Types**. Every value in TypeScript gives rise to a type of one instance—the value itself. For example, `3` is the type containing only the value `3`, `true` is the type containing only the value `true`, `undefined` is the type containing only the value `undefined`, and `null` is the type containing only the value `null`.

- **Numbers, Strings, and Symbols**. The static types `number`, `bigint`, `string`, and `symbol` correspond exactly to the runtime types of similar names.
- **The Object Type**. All objects belong to the static type `object`.
- **Object Types**. The type expression `{`$x_1 : T_1$`; ...;` $x_n : T_n$`}` specifies the type of all objects having properties x_i with corresponding static types T_i.
- **Interface Types**. Like object types, interfaces describe the "shape" of an object, namely the (typed) properties that objects of the interface type must have. The differences between interface types and explicit object types will be covered later in the chapter.
- **Classes**. A class gives rise to its own static type.
- **Union Types**. T_1`|...|`T_n is the static type containing all of the elements from each static type T_i. These are commonly used to form types with a small, fixed set of values, such as `"north"|"east"|"south"|"west"`, or "optional" types such as `number|null`. The union type `true|false` (where `true` and `false` are literal types), goes by the name `boolean`.
- **Tuple Types**. `[`T_1`,...,`T_n`]` is the type containing all ordered n-tuples made from each static type T_i. For example, the type `[boolean, 1|2|3]` has the six values `[true,1]`, `[true,2]`, `[true,3]`, `[false,1]`, `[false,2]`, `[false,3]`.
- **Array Types**. T`[]` is the type containing all arrays whose elements belong to static type T.
- **Intersection Types**. T_1 `&...&` T_n is the type containing the elements that belong to each static type T_i. These are most useful when combining interfaces, as we will see in more detail later in this chapter.
- **Function Types**. A function's static type is denoted $(p_1 : T_1, \ldots p_n : T_n) \Rightarrow T_0$, where the p_1 are the parameter names and the corresponding T_i are the parameter types, with T_0 being the return type of the function.
- **unknown**. *Every* value belongs to the type `unknown`.
- **void**. TypeScript uses `void` as the return type of functions without an explicit return. For example, a function that does nothing if its argument is true and throws an error if false might be typed `(condition: bool) -> void`.
- **never**. TypeScript introduces `never` as the return type of functions that never return, perhaps due to entering an infinite loop or throwing an error.

In practice, the TypeScript programmer annotates selected variables, parameters, and functions with static type expressions, while the compiler infers the types of unannotated expressions and validates that the program conforms to the type assignments. If the program does not conform, a **compile-time error** occurs and the translation to JavaScript fails. For example, the program `isFinite("1")` is rejected with the error "`Argument of type 'string' is not assignable to parameter of type 'number'`."

When inferring the types of compound expressions, TypeScript will compute the **best common type**. For arrays, this is often a union type formed from its elements; for example, the value `[3, "dog"]` is typed as `(number|string)[]`. The best common type for functions with multiple return statements is computed a little differently (see [62] for details).

When performing type validation, TypeScript checks whether an expression of type t can be sensibly used in a context in which an expression of type u is expected. In some theoretical contexts, we write this relation as t `<:` u. Generally, this relationship is rather intuitive, with t `<:` u whenever t is u, or t is literal type whose value is an instance of u, (e.g. `3 <: number`) or when t is a class extending u. Things get more complicated when dealing with arrays, tuples, and other collections; we'll cover those cases later in the chapter.

There is a special annotation that does not conform to an actual type (though it certainly looks like one). Annotating a variable, parameter, property, or return type with `any` tells the compiler not to type check its use at all, pushing the type checking to run time. The following program throws (and catches) a `TypeError` at run time.

```
import { strict as assert } from "assert"

function next(n: number): any {
  return n + 1
}

assert.equal(next(1), 2)
assert.throws(() => next(1)(2), /TypeError/)
```

When using the maximal level of static checking, TypeScript infers the types of expressions without annotations, but when being loose, nonannotated expressions are implicitly typed with `any`. It is this "implicit `any`" mode which allows us to say that all JavaScript programs are TypeScript programs. When transitioning from JavaScript to TypeScript, a programmer can choose which parts of the program require static type checking and leave the rest of the program unchecked, while gradually adding more type-checking over time (allowing TypeScript to be called **gradually typed**). Of course, it is recommended that maximal checking always be employed (except in rare instances that existing JavaScript code is slowly being ported to TypeScript), as the benefits of static type checking are often well worth the effort.

As type annotations can get rather large, you can employ a **type alias** to give a name to a type expression. The alias can be used as an annotation, for example:

```
import { strict as assert } from "assert"

type smallPrime = 2 | 3 | 5 | 7 | 11 | 13
type Point = { x: number; y: number }
type NumberPredicate = (n: number) => boolean

const five: smallPrime = 5

function checkYValue(test: NumberPredicate, p: Point): boolean {
  return test(p.y)
}

assert(checkYValue(n => n > 3, { x: 1, y: five }))
```

When determining compatibility of an object compatible with an object type, the object must have *at least* the properties mentioned in the type:

```
function move(entity: { x: number; y: number }, dx: number, dy: number) {
  entity.x += dx
  entity.y += dy
}

const robot = { name: "Mari", x: 2, y: 2 }
const mars = { type: "Planet", x: 3, y: 20, radius: 100 }
move(robot, 7, 20)
move(mars, 55, -13)
assert(robot.y == 22)
```

Interestingly, however, object *literals* have an extra compatibility rule: they cannot have excess properties! So directly invoking `move({ name: "Mari", x: 2, y: 2 }, 7, 20)` would produce an error, even though passing this value through a variable (as in the example above) succeeds.

Object types can be introducted with interfaces. Unlike aliased object types (e.g. `type Point = { x: number, y: number}`), interfaces gain new properties from subsequent declarations:

```
interface Coordinate {
  latitude: number
  longitude: number
}

interface Coordinate {
  when: Date
}

const party: Coordinate = {
  latitude: 33.931999,
  longitude: -118.437333,
  when: new Date("2022-08-25T00:30:00Z"),
}
```

Rather than adding properties to an interface, you can also create a new interface type from an existing interface with the `extends` keyword:

```
interface Person {
  name: string
}

interface Employee extends Person {
  title: string
}

let e: Employee = { name: "Allie", title: "CFO" }
```

An **intersection type** also combines all the properties of existing types:

```
type Duration = {
  seconds: number
}

type Instant = {
  when: Date
}

type Period = Instant & Duration

let p: Period = { seconds: 120, when: new Date(1969, 6, 20) }
```

TypeScript has JavaScript's classes, but adds new syntax for placing type constraints on properties. In addition, at the maximally strict setting, all properties must be initialized either at the property declaration or in a constructor.

```
class Ball {
  radius: number = 1
  color: string = "white"
}

let smallBall: Ball = new Ball()
assert(smallBall.radius === 1 && smallBall.color === "white")
let largeBall: Ball = { radius: 20, color: "red" }
assert(largeBall.radius === 20 && largeBall.color === "red")

class Sphere {
  radius: number
  color: string
  constructor(radius: number, color: string) {
    this.radius = radius
    this.color = color
  }
}

let sphere: Sphere = new Sphere(2, "red")
assert(sphere.radius === 2 && sphere.color === "red")
```

Properties may be marked `readonly` and may have an **access modifier**—one of `public`, `private`, or `protected`. If a modifier is present on a constructor parameter, TypeScript automatically generates and initializes a corresponding class property. The idiomatic way to write the `Sphere` class in the previous example is:

```
class Sphere {
  constructor(public radius: number, public color: string) {}
}

let sphere: Sphere = new Sphere(2, "red")
assert(sphere.radius === 2 && sphere.color === "red")
```

A public property is visible anywhere the type is visible; a private property is visible only within the type itself; a protected property is visible within the type or its subtypes. The `readonly` modifier prohibits the property from being assigned after its initialization. An example with all four modifiers follows:

```
import { strict as assert } from "assert";

abstract class Animal {
  constructor(private readonly name: string) {}
  protected abstract sound(): string;
  public speak(): string {
    return `${this.name} says ${this.sound()}`;
  }
}

class Cow extends Animal {
  sound() {
    return "moooo";
  }
}

class Horse extends Animal {
  sound() {
    return "neigh";
  }
}

class Sheep extends Animal {
  sound() {
    return "baaaa";
  }
}

const horse = new Horse("CJ");
assert(horse.speak() === "CJ says neigh");
const cow = new Cow("Bessie");
assert(cow.speak() === "Bessie says moooo");
assert(new Sheep("Little Lamb").speak() === "Little Lamb says baaaa");
```

For each of the animal types, only the public `speak` method is available outside the classes. Each class has access to, and can override, the protected `sound` method, which is customized for each animal. The `name` field is private, set only in the constructor and can subsequently never be changed nor even read directly except within the `Animal` class itself. We've chosen to make `Animal` an **abstract class**. Abstract classes cannot be instantiated, which models the absence of "plain" animals: only **concrete** cows, sheep, and horses may exist.

In addition to being able to extend a class (a JavaScript capability), subtyping can be achieved via the keyword `implements`, which enables TypeScript to check that any object constructed with a given class will conform to a specific interface:

```
import { strict as assert } from "assert"

interface Shape {
  area: () => number
}

class Circle implements Shape {
  constructor(public radius: number) {}
  area() {
    return Math.PI * this.radius ** 2
  }
}

class Rectangle implements Shape {
  constructor(public width: number, public height: number) {}
  area() {
    return this.width * this.height
  }
}

let shapes: Shape[] = [new Circle(10), new Rectangle(5, 10)]
assert(shapes[0].area() === 314.1592653589793)
assert(shapes[1].area() === 50)
```

Having object types fully specified allows TypeScript to reject expressions such as `c.x` at compile time if the property `x` is not a known property of `c`'s type—a powerful guard against bugs due to mistyped property names. But what if we need to access a property through a variable (e.g., `console.log(c[prop])`), or add or remove properties at run time? TypeScript provides **index signatures** for this purpose. In the following example, `[key: string]: Score` means that any number of addition properties with string keys and `Score` values are permitted:

```
type Score = 1 | 2 | 3 | 4 | 5

interface Rating {
  plot: Score // a plot score is required
  originality: Score // an originality score is required
  [key: string]: Score // you can create other scores of your choice
}

let rating: Rating = {
  plot: 5,
  originality: 4,
  idea: 3,
}

const metric = "clarity"
rating[metric] = 3 // Computed property access
```

```
for (const key in rating) {
  // Iteration through property names
  console.log(`${key} score is ${rating[key]}`)
}
```

Index signatures do have some limitations. The only allowed key types are `number`, `string`, and `symbol`; you cannot use literal types or unions, so `"north" | "east" | "south" | "west"`, while tempting, is not allowed. Also, TypeScript secretly adds `undefined` to your value types. In the example above, although we typed all values as `Score` rather than `Score | undefined`, you will still receive `undefined` when accessing a nonexistent property of `rating`!

4.3 GENERICS

Occasionally we wish to write flexible code that works on a variety of types. For example, adding an element to the beginning and end of an array, would, in TypeScript, be an infinite number of distinct functions: one to add a number to the beginning and end of a number array, one to add a string to the ends of a string array, and so on. Using `unknown` is too permissive—the element type of the array and the type of the element to add should be the same. TypeScript allows the use of a *type variable* to give us what we need:

```
import { strict as assert } from "assert"

function bracketed<T>(array: T[], item: T) {
  return [item, ...array, item]
}

assert.deepEqual(bracketed([1, 2, 3], 0), [0, 1, 2, 3, 0])
assert.deepEqual(bracketed(["x"], "o"), ["o", "x", "o"])
// bracketed([1, 2, 3], "x") will not compile
```

Here `bracketed` is a **generic** function to be **instantiated** with concrete functions as needed, replacing T with an actual type. We can even create generic functions with more than one type variable. Function composition, where functions of type $T \to U$ and $U \to V$ are chained to produce a function of type $T \to V$, is a classic example:

```
import { strict as assert } from "assert"

function compose<T, U, V>(f: (x: T) => U, g: (y: U) => V) {
  return (y: T) => g(f(y))
}

const lengthThenSqrt = compose((s: string) => s.length, Math.sqrt)
const encodedSquare = compose((x: number) => `${x * x}`, encodeURI)
const fourthRoot = compose(Math.sqrt, Math.sqrt)

assert.deepEqual(lengthThenSqrt("nineChars"), 3)
assert.deepEqual(encodedSquare(2.5), "6.25")
assert.deepEqual(fourthRoot(256), 4)
```

Interfaces and classes can be generic as well:

```
class Queue<T> {
  #data: T[] = []
  constructor() {
    this.#data = []
  }
  enqueue(item: T) {
    this.#data.push(item)
  }
  dequeue(): T | undefined {
    return this.#data.shift()
  }
  get size(): number {
    return this.#data.length
  }
}

let queue: Queue<number> = new Queue<number>()
queue.enqueue(1)
let anotherQueue = new Queue<string>()
anotherQueue.enqueue("first")
```

A number of built-ins in TypeScript are generic:

- `Array<T>` and `ReadonlyArray<T>`: JavaScript's array type with elements constrained to static type T, with mutable and immutable elements, respectively.
- `Set<T>` and `ReadonlySet<T>`: JavaScript's `Set` with elements constrained to static type T
- `Map<K, V>` and `ReadonlyMap<K, V>`: JavaScript's `Map` with keys of static type K and values of V
- `Promise<T>` JavaScript's promise that resolves to a value of type T.
- `Awaited<T>` The type of an await expression that is awaiting an expression of type T. If T is not a promise, then `Awaited<T>` is just T; if T is `Promise<U>` then it's U and if T is a promise of a promise of a ... promise resolving to a V, then the awaited type is V.
- `Record<K, V>` the type of an object whose keys are of type K and values of V. It is typical, but not necessary, for K to be a small union of string literals—something we saw in the last section that index signatures could not do.[2]

Do note that TypeScript's constraints are not perfect. The types `Array<string>`, `Set<string>`, `Map<string, string>`, and `Record<string, string>` can, as we saw with index signatures in the previous section, deliver the value `undefined` when accessing objects of these types at nonexistent properties. You may encounter some other very surprising cases where static typing fails to prevent run-time type errors. Here's a surprising example using the animal classes above:

[2] You can explore the differences between records and objects with index signatures in the end-of-chapter exercises.

```
let animals: Animal[] = [];
let cows: Cow[] = [new Cow("Bessie")];
animals = cows;
animals.push(new Sheep("Little Lamb"));
assert(cows[1].speak() === "Little Lamb says baaaa"); // WHAT?!
```

Even with all checks turned on, we managed to put a sheep into an array of cows (see Figure 4.1). How so? The first two lines constrain `animals` to an array of animals and `cows` to an array of cows. The third assigns `cows` to `animals` which TypeScript allows because a cow array *is* an animal array. Line 4 is permitted as the type checker sees only a sheep being added to an array of animals, which makes sense because a sheep is an animal. However, array assignment induces sharing, so the array referenced by `cows` now has a sheep. Not a good look.

Where was the problem? Technically three aspects of TypeScript clashed:

1. TypeScript sees `Cow[]` as essentially a subtype of `Animal[]`, allowing the assignment of the former to a variable constrained to the latter
2. Arrays are stored by reference, so array assignment induces sharing
3. A shared (aliased) object was allowed to be mutated

Figure 4.1 Aliasing of arrays

The first of these, treating `Array<Cow>` as a subtype of `Array<Animal>`, is called **covariance**, a topic we will cover more fully in Section 7.6. When combined with aliasing and mutation, run time type errors may occur. In fact, the TypeScript designers specifically did *not* set out to create "a sound or provably correct type system. Instead [they sought to] strike a balance between correctness and productivity." [75] For the TypeScript programmer, this means that the static type of an expression may not reflect reality at run time. In our example above, TypeScript will infer that `cows[1]` has static type `Cow`[3], even though runtime value of this expression was created with the `Sheep` constructor.

There is much more to the study of generics that we have covered here, including generic constraints and variance annotations on type parameters. Generic constraints, in particular, will appear in subsequent languages on our tour.

4.4 OPTIONALITY

TypeScript, as a superset of JavaScript, provides the value `null` to represent the intentional absence of information, and the value `undefined` to represent a lack of knowledge of a value. If a variable or property named `supervisor` were `null`, we are certain no supervisor exists,

[3]In most text editors, hovering over an expression will give you an indication of the inferred type

but if `undefined`, there may or may not be a supervisor—we may not know or we may just be refusing to say. If the presence of a supervisor is not required, we can use a union type to express that it is optional:

```
function optionalsAsUnionTypesExample() {
  type Person = {
    id: number
    name: string
    supervisor: string | null | undefined
  }

  const people: Person[] = [
    { id: 1, name: "Alice", supervisor: "Tina" },
    { id: 2, name: "Bob", supervisor: null },
    { id: 3, name: "Carol", supervisor: undefined },
  ]
}
```

Here any object of this person type must have a property called `supervisor` and explicitly be given a value, thought that value may be `null` or `undefined`. We can arrange to make the property itself optional:

```
function optionalPropertyExample() {
  type Person = {
    id: number
    name: string
    supervisor?: string
  }

  const people: Person[] = [
    { id: 1, name: "Alice", supervisor: "Tina" },
    { id: 2, name: "Bob" },
  ]
}
```

TypeScript infers the property type for `supervisor` to be `string | undefined`, but the convenience of being able to omit the property completely may express the programmer's intent better than explicitly supplying the value `undefined`. A similar situation occurs for function declarations, where we may wish to omit an argument in a call:

```
function log(message: string, prefix?: string) {
  if (prefix) {
    console.log(`${prefix}: ${message}`)
  } else {
    console.log(message)
  }
}

log("Hello, world!")
log("Hello, world!", "INFO")
```

Without marking the `prefix` parameter optional, TypeScript would have required all calls of this function to supply exactly two arguments. Granted, the declaration here is exactly equivalent to writing `function log(message: string, prefix: string | undefined = undefined)`, but the optional parameter syntax is arguably much clearer.

4.5 NARROWING

The TypeScript compiler helps us by inferring the (static) type of each expression in a program, preventing us (in most cases) from using an expression incorrectly. If an expression is inferred to have type `string | number`, we cannot treat it as a string. However, the following compiles and runs perfectly:

```
let x: string | number = Math.random() < 0.5 ? "small" : 100
if (typeof x === "string") {
  console.log(x.toUpperCase())
} else {
  console.log(x ** 2)
}
```

Here we used a **type guard** in the if-statement condition to **narrow** the type of x in the first arm to `string`, and hence to `number` (the only remaining alternative from the original union type) in the second. Narrowing can also occur in the presence of simple assignments:

```
let y: number | string
y = "hello"
console.log(y.toUpperCase())
y = 3
console.log(y ** 2)
```

Control flow analysis can also aid narrowing. Here TypeScript is able to determine that the final return statement is reached only if the if-condition was false:

```
function nextStep(supervisor: string | null) {
  if (supervisor === null) {
    return "Nothing to do"
  }
  return `Call ${supervisor}` // supervisor is a string
}
```

TypeScript's ability to narrow can seem pretty clever at times. In the following, x and y are both narrowed to type `number` since this is the only type they could have when equal:

```
function f(x: string | number, y: number | boolean) {
  if (x === y) {
    console.log(x ** 2) // x and y are both numbers
  }
}
```

Type guards for narrowing can be implemented in several ways, including `typeof` (for distinguishing run-time types), `instanceof` (for types within type hierarchies—classes and interfaces related via `extends` and `implements`), and the `in` operator (when constituents

of a union can be distinguished by properties). If you would like to abstract the type guard into its own function, you write the return type of this function as a **type predicate** as in this example:

```
type Sphere = { radius: number }
type Box = { length: number; width: number; depth: number }
type Shape = Sphere | Box

function isSphere(shape: Shape): shape is Sphere {
  return "radius" in shape
}

function volume(shape: Shape) {
  if (isSphere(shape)) {
    return (4 / 3) * Math.PI * shape.radius ** 3
  } else {
    return shape.length * shape.width * shape.depth
  }
}
```

If the return type for our `isShape` function were simply `boolean`, the inferred type inside the first arm of the if-statement would be `Shape`; the type predicate narrows this to the `Sphere`.

In a similar fashion, you can abstract error checking into an **assertion function**, saving your main code from noisy if-statements. Annotate your function with a condition that you state will hold *if* your function returns. While type predicates return true or false, assertion functions either return nothing or throw:

```
type Access = "read" | "write" | "append";

function assertRead(access: Access): asserts access is "read" {
  if (access !== "read") {
    throw new Error("Access is not read");
  }
}

function viewAsGuest(access: Access) {
  // Code looks great as the if-and-throw are elsewhere
  assertRead(access);
  // Actual reading would take place here
  // The type of the variable `access` is narrowed to "read" here
  // (Hover over both occurrences of access to see the difference)
  access;
}
```

Sometimes annotations and type inference work together in ways you might not wish for, but TypeScript has a convenient way to make you happy. In the code below, “things” have a name and a color, and colors can be specified either by a name, or by their red, green, and blue components:

```
type Color = string | { r: number; g: number; b: number }

type Thing = {
  name: string
  color: Color
}

const thing1 = { name: "thing1", color: "red" }
const thing2: Thing = { name: "thing2", color: "red" }
const thing3 = { name: "thing1", color: "red" } satisfies Thing

thing1.color.toUpperCase()
// thing2.color.toUpperCase() // ERROR: thing2.color could be an object
thing3.color.toUpperCase()
```

We've left off a type annotation for `thing1`, so TypeScript infers a type of `{ name: string, color: string }`. We have the desired inferred type, so we can treat `thing1.color` as a string. But the lack of an annotation means that TypeScript will not be able to check any typos we might make when defining the thing. We *do* annotate `thing2`, enabling conformance checking, but now the inferred type is `Thing`, which means `thing2.color` is a `color`, not a string. To get the best of both worlds—conformance checking against the `Thing` type and inference to the narrowed color variant in the constant, we use the `satisfies` operator to conformance check without widening the property types.

There's more to narrowing than we've covered in this short section. For a full run-through of all of the narrowing powers of the language, see the Narrowing section in the TypeScript Handbook [73].

4.6 TYPE OPERATORS

TypeScript goes to great lengths to help analyze your code for type errors. It provides several operators to create new types from existing types, enables you to work with different types under different conditions, and predefines a decent set of utility types to specify rather precisely what you expect of your code.

For an object type T, `keyof T` produces a union of T's keys. In a type context, `typeof` x, where x is a variable or a property, produces the (static) type of its operand.

```
type Widget = {
  id: string
  name?: string
  price?: number
  status: "soldOut" | "inStock" | "discontinued"
}

let ex01: Widget
let ex02: keyof Widget // "id" | "name" | "price" | "status"
let ex03: typeof ex01 // Widget
```

The type expression $T[K]$ for object type T and a type K suitable for indexing, provides the type(s) of all index operations on objects of type T with those keys:

```
let ex04: Widget["id" | "price"] // string | number
let ex05: Widget["name"] // string | undefined
let ex06: Widget["status"] // "soldOut" | "inStock" | "discontinued"
```

We present a handful of pre-defined **utility types** here:

```
let ex07: Partial<Widget> // all properties optional
let ex08: Required<Widget> // all properties required
let ex09: Readonly<Widget> // all properties readonly
let ex10: Pick<Widget, "id" | "name"> // only id and name props
let ex11: Omit<Widget, "id" | "name"> // all props except id and name
let ex12: Exclude<"a" | "b" | "c", "a"> // "b" | "c"
let ex13: Extract<"a" | "b" | "c", "a" | "b"> // "a" | "b"
```

Further utility types provide a means to describe components of function types:

```
function f1(w: Widget, factor: number): number {
  return (w.price ?? 0) * factor
}

let ex14: Parameters<typeof f1> // [Widget, number]
let ex15: ReturnType<typeof f1> // number
```

A few more utility types are useful in a class context: `ConstructorParameters<T>`, `InstanctType<T>`, and ThisType<T>!.

Not surprisingly, there are quite a few more kinds of types that can be expressed. Among those that we'll leave you to explore on your own are: **conditional types**, which can express types with different forms based (usually) on a generic parameter, **mapped types**, which are generated by property transformations on existing types, and **template literal types**, which are unions of string literals computed rather than listed explicitly.

4.7 TYPE ASSERTIONS

We've seen how TypeScript provides extraordinary compile-time analysis by allowing annotation with a rich set of static types (including unions, interfaces, functions, and more) and its powerful inference capabilities (including control flow analysis in narrowing). We've also seen that it prioritizes convenience over 100% inference accuracy (as in the earlier covariant array example). In the spirit of convenience, TypeScript even lets *you* assert the types of expressions yourself. Perhaps you may do so for stylistic reasons:

```
let scores = [] as number[]
// We could have also written let scores: number[] = []
scores.push(100)
```

You may also have received a value from somewhere in the code that was annotated with a type that was quite general, but *you* know better:

```
function dataFromSomeoneElse(): unknown {
  return "Hello friends, I am a string"
}

let shouted = (dataFromSomeoneElse() as string).toUpperCase()
// Note dataFromSomeoneElse().toUpperCase() is an error
```

A classic example of TypeScript not knowing the most narrow type comes from web development:

```
function enableGamePlay() {
  // getElementById returns HTMLElement | null
  let playButton = document.getElementById("play") as HTMLButtonElement
  playButton.disabled = false
}
```

Type assertions may even be necessary when accessing properties indirectly:

```
let order = {
  "doro wot": 2,
  shiro: 5,
  fatira: 3,
}

let favorite = "fatira"
const quantity = order[favorite as keyof typeof order]
// Note order[favorite] is an error!
console.log(`I ordered ${quantity} plates of ${favorite}`)
```

Not all assertions use the `as` operator. The `keyof` and `typeof` "assertions" can also appear in a function signature:

```
function serve(name: keyof typeof order) {
  console.log(`Serving ${order[name]} plates of ${name}`)
}

serve("doro wot")
```

Use type assertions with care—you are free to assert a type constraint that simply does not hold, but TypeScript will not complain:

```
const offMenu = "pepperoni"
serve(offMenu as keyof typeof order) // undefined at runtime!
```

Remember, TypeScript is your friend and does whatever it can to help you. But it does not force you into anything. Letting you assert types, even if you may be wrong, is a way to ensure the programmer is always the one in control. In practice, you may never use this great power, but it is there if you need it.

TYPESCRIPT WRAP UP

In this chapter, we were introduced to TypeScript. We learned that:

- TypeScript is a superset of JavaScript. Any JavaScript program is also a TypeScript program.
- TypeScript has a rich set of **static types** used by the compiler to check the type-consistency of expressions before running the program, and a set of eight **dynamic types** that can be queried at run time.
- The static type system includes literal types, basic types, object types, intersections, unions, tuples, arrays, functions, and more. The bottom type is `never` and the top type is `unknown`.
- In practice, very few expressions must be annotated with static types, as the compiler does quite a good job of inferring the best type.
- Objects can be typed with object types or interfaces. Properties can be marked `readonly`. In a class, the access modifiers `private`, `protected`, and `public` are available. Classes can also be abstract, extend other classes, or implement interfaces.
- Generic, or parameterized, types are useful in creating functions and objects that work on multiple types.
- TypeScript arrays are simultaneously (1) mutable, (2) stored by reference, and (3) covariant. This is an indication that the type analysis is not 100% sound at compile time, but this is intentional, as the language prioritizes convenience and simplicity over type soundness.
- TypeScript makes frequent use of narrowing during type inference. Narrowing is the process of inferring a more specific type for an expression based on surrounding code. Narrowing can be influenced by type guards, assignments, control-flow analysis, and user-defined type predicates and assertion functions.
- TypeScript defines type operators and utility types to give your code more clarity and more precision in type checking.
- TypeScript does not force you to accept its own inferred types—you may *assert* the static type that it should use, even if that type is not correct.

For further study in TypeScript, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using TypeScript, see the TypeScript page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/ts.

EXERCISES

Now it's your turn. Continue exploring TypeScript with the activities and research questions below, and feel free to branch out on your own.

4.1 Familiarize yourself with the TypeScript home page at http://typescript.org and the docs page at http://typescript.org/docs. Browse the TypeScript Handbook (accessible from the docs page), making a note of the language features not covered in this chapter.

4.2 Experiment with the TypeScript Playground at http://typescript.org/play. Set the options to do maximal checking. Practice writing programs with errors and view the messages that appear when hovering the cursor over the error. Practice reading the live translation to JavaScript.

4.3 Research the difference between marking a class property with `private` and using `#` at the beginning of the property name.

4.4 In the `Shape-Circle-Rectangle` example from this chapter, would the script have passed all typechecks and run properly if `Circle` and `Rectangle` did not implement the `Shape` interface? If so, why might a programmer use the `implements` clause in this case?

4.5 What is the difference between `ReadonlyArray<number>` and `Readonly<number[]>`?

4.6 Rewrite the ratings and score example from this chapter to use the `Record` utility type.

4.7 Port the Pokemon API example from the JavaScript chapter to TypeScript, annotating as many variables as you can.

4.8 In this chapter we saw how a variable with static type `List<Cow>` was able to have a sheep appended to it, despite our not using any kind of type assertion or any other direct subversion of the type system. We traced the problem to a confluence of covariance, reference types, and mutation. Determine whether this type system hole still exists when any two of the three scenarios (but not all three) hold.

4.9 TypeScript features a postfix `!` operator that has something to do with "optionals." What exactly does it do?

4.10 Give an example in which the `&&` operator is used for narrowing.

4.11 TypeScript retained the `typeof` operator of JavaScript (it is a superset language after all) for runtime type tests. But it gave the operator additional properties when used "in a type context." Research the full set of capabilities for this operator in the static type system.

4.12 We saw examples of type predicates in this chapter, all of which applied to a function's sole parameter. But for a *method*, the type predicate must refer to the method's receiver. How is this done in TypeScript? Give an example.

4.13 We did not cover TypeScript conditional types, mapped types, or template literal types. Prompt an AI chatbot to explain each of these features and write illustrative code for each. Try out the examples.

4.14 Discuss the claim: "Type predicates and assertion functions can be abused to give incorrect type information that TypeScript will accept and work with." If true, give examples. If false, explain the protections offered by the language to prevent such scenarios.

CHAPTER 5

Python

Python is "a programming language that lets you work quickly and integrate systems more effectively." [97]

First appeared	1991
Creator	Guido van Rossum
Recognized for	Expressiveness
Notable uses	Web servers, Scripting, Data science
Tags	Imperative, Glue, Dynamic, Scripting
Six words or less	Easy to learn and extremely productive

Guido van Rossum started working on Python in 1989, and has overseen much of this popular language's development for three decades. The Python community bestowed on him the title Benevolent Dictator for Life, or BDFL, which he held until 2018.

Python is known for being easy to read and write. The syntax has a light and uncluttered feel. A large number of data types, including tuples, lists, sets, and dictionaries are built-in with convenient syntactic forms. While particularly powerful as a scripting language, it is a truly general-purpose language, having been used in domains from games and entertainment to web applications to statistics and data science. The language is also notable for its culture. Python aficionados, known as *Pythonistas*, take pride in writing elegant, expressive, *Pythonic* code. They learn and practice *The Zen of Python* [87], a collection of best practices and philosophies that help give applications a characteristic feel.

Our survey of Python will begin with our common three introductory scripts. We'll continue with a look at Python's rich type system, in which all types are organized into an ontology that supports something called **multiple inheritance**. We'll see how Python's handling of scope also differs from previous languages on our tour. We'll cover many of the language's modern features including keyword arguments, comprehensions, a powerful match statement, custom operator definition (reminiscent of Lua's metamethods), and its support for iterators and generators. We'll close with a look at Python's decorators.

5.1 HELLO PYTHON

Let's get started in the usual way, listing right triangle measurements:

```
for c in range(1, 41):
    for b in range(1, c):
        for a in range(1, b):
            if a * a + b * b == c * c:
                print(f'{a}, {b}, {c}')
```

Python uses indentation to define structure. The `range` constructor builds a range object, inclusive at the bottom and exclusive at the top. Ranges produce their values on demand, so there is no need to worry that large lists are being created before iteration. String interpolation uses curly braces within **f-strings**.

Here's Heap's algorithm in action permuting command line arguments:

```
import sys

def print_permutations(a, n):
    if n <= 0:
        print("\t".join(a))
    else:
        for i in range(n):
            print_permutations(a, n - 1)
            j = 0 if n % 2 == 0 else i
            a[j], a[n] = a[n], a[j]
        print_permutations(a, n - 1)

print_permutations(sys.argv[1:], len(sys.argv) - 2)
```

Command line arguments are stored in the list `argv` from the `sys` module, which we must explicitly *import*. The script name is contained in `argv` at index 0, so the true arguments come from the slice of `argv` starting at index 1. Python's conditional expression has the form y `if` x `else` z—quite a contrast to JavaScript's x `?` y `:` z.

Here's our third standard teaser script:

```
import sys
from heapq import nlargest
from operator import itemgetter

players = []

for line in sys.stdin:
    team, name, games, points = line.strip().split(',')
    if int(games) >= 15:
        players.append((name, team, int(points)/int(games)))

for name, team, ppg in nlargest(10, players, key=itemgetter(2)):
    print(f'{name:22}{team:4}{ppg:8.2f}')
```

The standard input stream (`sys.stdin`) is *iterable*, line by line, so it can be used in a `for` statement. We split each line into its team, name, games, and points components, then rebuild each player line into a **tuple** with the name, team, and ponts-per-game, and store those tuples for players appearing in 15 or more games into a list. To get the top ten scorers by points-per-game, we do not need to sort the entire list: we have the more efficient `nlargest` function from the `heapq` module. The fancy `itemgetter` indicates that we are determining the largest elements based on the tuple value at index 2, namely points-per-game (as tuples are 0-based).

5.2 THE BASICS

Let's begin with types. Integers (of type `int`) are unbounded, floating-point numbers (`float`) are generally IEEE binary64, [51] the booleans (`bool`) are `True` and `False`, and the utility value `None` has its own type `NoneType`. For text, we have both `str` for sequences of Unicode characters and `bytes` for byte sequences.

Where JavaScript and Lua each have a small, fixed, number of types; Python has an unlimited number. Tuples, lists, sets, dictionaries, functions—and user-defined abstractions such as employees, blog posts, and calendar appointments—would all have type `Object` in JavaScript and type `table` in Lua. Not so in Python. Let's explore:

```
# Many built-in types have built-in names
assert type(5) == int
assert type(True) == bool
assert type(5.7) == float
assert type(9 + 5j) == complex
assert type((8, 'dog', None, False)) == tuple
assert type('hello') == str
assert type(b'hello') == bytes
assert type([1, '', False]) == list
assert type(range(1,10)) == range
assert type({1, 2, 3}) == set
assert type(frozenset([1, 2, 3])) == frozenset
assert type({'x': 1, 'y': 2}) == dict

# Built-in vs. User-defined functions
def plus_two(x):
    return x + 2
assert str(type(plus_two)) == "<class 'function'>"
assert str(type(max)) == "<class 'builtin_function_or_method'>"

# Even modules are types!
import math
assert str(type(math)) == "<class 'module'>"

# Many built-in modules define their own types
from datetime import date, datetime
assert type(date(1969, 7, 20)) == date
assert type(datetime.now()) == datetime
```

Note that Python types are objects! Whereas in JavaScript and Lua, asking for the type of the value 5 produces the *string* `"number"` in reply, Python gives you the object (`int`) that *is* the type of integers. As `int` is an object, it has a type too:

```
assert type(int) == type
assert type(list) == type
assert type(type) == type  # Interesting...or obvious?
```

All sequences are indexed from 0, and support access (e.g., `s[3]`), slicing (e.g., `s[3:5]`, `s[:3]`, `s[2:]`, `s[3:21:2]`),[1] and length (e.g., `len(s)`). List, dictionaries, and sets can be written using **comprehensions**, in which the elements are computed by rules with `for` and `if` clauses, for example:

```
assert [(x,y) for x in range(1,3) for y in range(5) if x < y] == \
    [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4)]
assert {x**2 for x in range(5) if x % 2 == 0} == \
    {0, 16, 4}
assert {k: len(k) for k in ("red", "green", "blue")} == \
    {'red': 3, 'green': 5, 'blue': 4}
```

To define your own types, create a class:

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius * self.radius

c = Circle(10)

assert type(c) == Circle            # Circle is a type!
assert type(Circle) == type         # It really is!

assert str(type(Circle.area)) == "<class 'function'>"
assert str(type(c.area)) == "<class 'method'>"

assert c.area() == math.pi * 100    # A method call...
assert Circle.area(c) == c.area()   # ...is just sugar
```

Here we've created a new type, `Circle`. Classes are **namespace**s: we did not define a global name `area`, we defined `Circle.area`. A class is also **callable**; the expression `Circle(10)` creates a new **instance** of type `Circle` and immediately passes it to `Circle.__init__` for initialization, here setting its `radius` **attribute**. Python provides syntactic sugar very similar to Lua's: `c.area()` is shorthand for `Circle.area(c)`.[2] The sugared form gives you

[1] Slice bounds `i:j` are inclusive of the start index and exclusive of the end. The start index defaults to 0 and the end to the sequence length. An optional third component (as in `i:j:k`) is a step size, selecting "every *k*th element."

[2] Note that Lua's sugared expression uses a colon (`c:area()`), while Python is able to simply use the dot (`c.area()`). Why is Python able to do this, while Lua could not?

the feeling of a method invocation where *c* is the receiver; indeed, our script shows that `c.area` is a method, while `Circle.area` is a plain old function.

Unlike JavaScript, TypeScript, and Lua, whose variables either hold direct values or store references based on type, *all Python variables store references*. Every value is an object, and every object has a unique identifier, accessible to the programmer:

```
a = 5                    # 5 is an object
b = {'x': 5, 'y': 3}     # dicts are objects
c = "hello"              # strings are objects too
d = c                    # two variables sharing an object
e = c.lower()            # should generate a new object
f = 8 * b['y'] - 19      # what happens here?

for obj in (a, b, b['x'], b['y'], c, d, e, f):
    print(id(obj))
```

One run of this program produced bindings of objects to variables shown in Figure 5.1. Variables *c* and *d* reference the same object, since assignment copies references. Interestingly, we see there is only one copy of the integer 5 in the **object pool**; this is acceptable because integers are immutable. Immutability allows sharing because it is impossible for a change through one variable to change the value seen by the aliased variable. Python strings are immutable as well, so our implementation *could have* decided to share a single copy of the string `"hello"` among three variables, but chose not to.

Figure 5.1 Python variables and objects

Immutable values can not only be shared, they can be stored directly inside variables, as long as the built-in `id` function returns a reasonable value. This should be a big relief to readers wondering whether Python engines actually take the time to allocate and indirectly access simple numbers and booleans.

Python types have supertypes and subtypes. While JavaScript, TypeScript, and Lua require the programmer to set up prototype chains to simulate IS-A relationships between types, Python models this relationship directly:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return f'{self.name} says {self.sound()}'

class Cow(Animal):
    def sound(self):
        return 'moooo'

class Horse(Animal):
    def sound(self):
        return 'neigh'

class Sheep(Animal):
    def sound(self):
        return 'baaaa'

s = Horse('CJ')
assert s.speak() == 'CJ says neigh'
c = Cow('Bessie')
assert c.speak() == 'Bessie says moooo'
assert Sheep('Little Lamb').speak() == 'Little Lamb says baaaa'
```

Here we've made `Animal` the **superclass** of its three **subclasses** `Horse`, `Cow`, and `Sheep`. `Animal` has a superclass too, namely `object`, the ultimate superclass of all Python classes. Subclass instances **inherit** attributes of their superclass(es), so, any instance of `Horse` has not only a `sound` attribute, but `__init__` and `speak` attributes too. Classes can have multiple direct superclasses, a condition known as **multiple inheritance**. Consider:

```
from subprocess import call
from animals import Cow

# Disclaimer: This only works on an O.S. with a say command.
class Vocalizer:
    def vocalize(self):
        call(['say', self.name + 'says' + self.sound()])

class VocalCow(Vocalizer, Cow):
    pass

c = VocalCow('Bessie')
print(c.speak())        # speak is inherited from Cow
c.vocalize()            # vocalize is inherited from Vocalizer
```

A `VocalCow` can both speak and vocalize (see Figure 5.2). But what if we had called the vocalizer method `speak` instead of `vocalize`? Would `VocalCow` have two `speak` attributes? Or would one be inherited and the other not? Or would the attempt to build the class with two superclasses fail because of the name clash?

Figure 5.2 Python animal classes

This question is ultimately one of how Python finds attributes of an object that the object's class does not directly define. In the languages we've seen previously, lookup proceeds along a single prototype chain, but in Python we need to know whether to look first in `Cow` *or* `Vocalizer` for attributes not in `VocalCow`. Python addresses this question by giving each class a **method resolution order**, or **MRO**. In our case, evaluating `VocalCow.mro()` produces:

```
[VocalCow, Vocalizer, Cow, Animal, object]
```

This implies attributes in `Vocalizer` take precedence over those in `Cow`. We could, of course, modify the order to suit our needs. It can get tricky, but the mechanism is well-defined. See [104] for a detailed description of the algorithm used to compute the MRO.

It's helpful to see how Python's plethora of built-in types are related (Figure 5.3). The common built-in types are all direct subtypes of `object`, with one notable exception: `bool` is a subclass of `int`. Interestingly, the various sequence types and numeric types do not share any common supertype (other than `object`); we've called out these informal groupings in the figure simply because programmers may find them useful.

Figure 5.3 A few of the built-in types in Python

Python features two different operators for variable assignment. Use `=` when assignment is a statement, and `:=` (the **walrus operator**—look at it sideways, do you see the tusks?) when an expression. The `:=` form cannot appear in just any expression [4], and is generally used only in certain idiomatic situations where readability is actually improved, such as:

```
if (offered := len(candidates)) != 3:
    raise ValueError(f"Expected 3 candidates, received {offered}")

while (item := input('What do you like? ')) != '':
    print(f"I like {item}, too")

for x in range(1000, 10000, 1000):
    print(x, log1 := math.log(x), log2 := math.log(log1), math.log(log2))
```

Python has approximately 20 kinds of statements. Many are fairly common (assignment, break, continue, return, conditionals and loops, function and class definitions, assertions, raising and catching errors and so on), but a few have some uncommon twists. The `for` and `while` loops have an `else` clause, which executes when the loop finishes normally (not via break, return, or raise):

```
for place in places:
    if good(place):
        print(f"Ah, a good place: {place}")
        break
else:
    print("No good places for you")
```

Unlike JavaScript, Python cannot destructure function call arguments nor assignments (other than tuples), but it does perform general pattern matching in the `match` statement.

```
def describe(e):
    match e:
        case 2 | 3 | 5 | 7 | 11:
            return "Small prime"
        case n if n % 5 == 0:
            return "Multiple of 5"
        case (_, x, _):
            return f"A three-tuple with {x} in the middle"
        case {'id': y}:
            # Matches any dictionary that has an id key, binds value to y
            return f"Your identifier is {y}"
        case _:
            return "What?"

assert describe(5) == "Small prime"
assert describe(15) == "Multiple of 5"
assert describe((1, 2, 3)) == "A three-tuple with 2 in the middle"
assert describe({"data": 2, "id": "x"}) == "Your identifier is x"
assert describe("Mu") == "What?"
```

5.3 TYPE HINTS

All Python expressions have a (dynamic) type available at run time, but Python's syntax allows for static types to be specified as well. A type checker processes these annotations, known as **type hints**, and reports any type inconsistencies it can find. The set of static types is much richer than that of dynamic types; for example, Python's only dynamic list type is `list`, but static list types can specify the element type as well. The static types support unions, callables, sequences, generics, and even type variables. As in TypeScript, type narrowing is supported. The following script illustrates only a few of the many features of the static type system:

```
dozen: int = 12
found: bool
found = False
from typing import List, Set, Dict, Tuple
def first_two(a: List[float]) -> Tuple[float, float]:
     return (a[0], a[1])

assert first_two([False, 0, 17, 2.2, []]) == (False, 0)
```

Type use of type hints is completely optional in Python. Variables without annotations are assumed to belong to the type `Any`, compatible with every type. Even when hints are used, they have no run-time effect whatsoever.

5.4 SCOPE

Python variables are bound within blocks, which include top-level modules, function definitions, or class definitions. There is no explicit variable declaration; assigning to a variable inside a function or class creates a new variable, unless marked `global` or `nonlocal`. As in previous languages, Python locals **shadow** in-scope variables of the same name.

```
a, b, c = 1, 2, 3   # Three global variables

def f(x):
    d = 4           # Brand new local!
    a = 5           # Local to f, shadows global
    def g():
        x = 1       # Local to g, shadows f's x
        nonlocal a  # allows access to f's a
        a = 6       # updates the nonlocal
        global c
        c = 7
    g()
    assert x == 10  # g's x did not overwrite f's x
    assert a == 6   # assert that it changed
    assert b == 2   # globals are visible

f(10)
assert a == 1 and b == 2 and c == 7
```

Using a local before it is first written raises an `UnboundLocalError`; using a global before it is first assigned raises a `NameError`:

```
try:
    print(x)
except NameError as e:
    print('NameError raised as expected')

x = "I'm global"

def problem():
    print(x)          # Local but not bound yet
    x = "I'm Local"

try:
    problem()
except UnboundLocalError as e:
    print('UnboundLocalError raised as expected')
```

The scope rules around comprehensions are worth noting. As stated in the Python Reference: "...aside from the iterable expression in the leftmost for clause, the comprehension is executed in a separate implicitly nested scope." [96] However, a variable bound with the walrus operator in a comprehension will be bound in the outer scope [4].

```
x = 3
assert [x for x in range(5)] == [0, 1, 2, 3, 4]
assert x == 3 # x was not changed

print([y for y in range(5)])
try:
    print(y) # will raise an error, since y did not leak
    assert False
except Exception as e:
    assert e.__class__.__name__ == 'NameError'

assert [(y := 1) for x in range(5)] == [1, 1, 1, 1, 1]
assert y == 1 # The := operator creates a new variable!
```

5.5 PARAMETER ASSOCIATION

Python passes arguments to parameters by (conceptually) passing each argument's id. This avoids the expense of copying large arguments and allows both the caller and callee to reference the same object. We call this mechanism **pass-by-sharing**. The language supports both default arguments and argument packing and unpacking. In a definition, a single-starred parameter packs additional arguments into a tuple (recall JavaScript's rest parameter). In a call, the single-starred argument unpacks a sequence to pass constituent elements as multiple parameters (recall JavaScript's spread operator):

```
def f(x, y=1000, *z):
  print(f'x={x} y={y} z={z}')

f(0)                          # x=0, y=1000, z=()
f(0,1)                        # x=0, y=1, z=()
f(0,1,2)                      # x=0, y=1, z=(2,)
f(0,1,2,3,4)                  # x=0, y=1, z=(2,3,4)
f(*range(7))                  # x=0, y=1, z=(2,3,4,5,6)
```

But there is much more to parameter association in Python.

In his essay *Learnable Programming*, Bret Victor lays out nine design principles for building an environment and a programming language suitable for learning to program. One of his principles for a language is to MAKE MEANING TRANSPARENT. He motivates this principle by showing a snippet of code (not in Python) containing the lines

```
fill(161,219,114);
rect(105,20,60,60);
```

and suggests that a learner might ask "What do the numbers after **fill** mean?" and "What units are they in?" and "Why are there so many numbers?" A simple way to help with the first question is to supply the parameter names in a call. And we do exactly that in idiomatic Python:

```
set_fill_color(red=161, green=219, blue=114)
draw_rectangle(corner=(105,20), other_corner=(60,60))
```

Arguments passed in this fashion are called **keyword arguments**. They can be passed in any order, though they must follow the nonkeyword, **positional** arguments. They provide a nice alternative to passing a dictionary:

```
def write_rule(selector, **options):
    print(selector, '{')
    for prop, value in options.items():
        print(f'  {prop.replace("_", "-")}: {value};')
    print('}')

write_rule('h1', font_family='Helvetica', size='20px')
write_rule('p.error', color='red', margin='16px', padding='0')
```

By default, arguments can be passed either as positional arguments or as keyword arguments. However, we can *require* a particular association by partitioning parameters in the function declaration: those preceding a `/` must be positional; those following a `*` must be keyword, as in this example from the Python documentation:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)

f(10, 20, 30, d=40, e=50, f=60)
```

Positional-only arguments are useful when the parameter name is rather generic (e.g., `x`), and keyword-only arguments shine when there are several arguments.

5.6 SPECIAL METHODS

Python is a rich language with dozens of operators and built-in functions. It conveniently allows you to customize nearly all of them on your own classes. As in Lua, Python provides certain special functions that are implicitly invoked in certain contexts; Table 5.1 lists some of the more commonly used ones.

Expression	What is Called	Expression	What is Called
`+a`	`a.__pos__()`	`a > b`	`a.__gt__(b)`
`-a`	`a.__neg__()`	`a >= b`	`a.__ge__(b)`
`~a`	`a.__invert__()`	`a & b`	`a.__and__(b)`
`abs(a)`	`a.__abs__()`	`a | b`	`a.__or__(b)`
`a + b`	`a.__add__(b)`	`a ^ b`	`a.__xor__(b)`
`a - b`	`a.__sub__(b)`	`a << b`	`a.__lshift__(b)`
`a * b`	`a.__mul__(b)`	`a >> b`	`a.__rshift__(b)`
`a / b`	`a.__truediv__(b)`	`a` *(in bool context)*	`a.__bool__()`
`a // b`	`a.__floordiv__(b)`	`len(a)`	`a.__len__()`
`a % b`	`a.__mod__(b)`	`a in s`	`s.__contains__(a)`
`divmod(a,b)`	`a.__divmod__(b)`	`d[k]`	`d.__getitem__(k)`
`a ** b`	`a.__pow__(b)`	`d[k] = v`	`d.__setitem__(k, v)`
`a == b`	`a.__eq__(b)`	`iter(a)`	`a.__iter__()`
`a != b`	`a.__ne__(b)`	`next(a)`	`a.__next__()`
`a < b`	`a.__lt__(b)`	`str(a)`	`a.__str__()`
`a <= b`	`a.__lr__(b)`	`repr(a)`	`a.__repr__()`

Table 5.1 Python special methods

Let's make a little class for two-dimensional vectors, translating as directly as possible our Lua program from the previous chapter:

```
import math

class Vector:
    def __init__(self, i, j):
        self.i = i
        self.j = j

    def magnitude(self):
        return math.sqrt(self.i * self.i + self.j * self.j)

    def __add__(self, v):
        return Vector(self.i + v.i, self.j + v.j)

    def __mul__(self, v):
        return self.i * v.i + self.j * v.j

    def __str__(self):
        return f'<{self.i},{self.j}>'
```

```
u = Vector(3, 4)
v = Vector(-5, 10)
assert u.i == 3
assert u.j == 4
assert u.magnitude() == 5.0
assert str(u + v) == '<-2,14>'
assert u * v == 25
```

There are quite a few similarities with the Lua code (the use of special methods for one), but a few differences, too. Perhaps the most significant is that Python does not require explicit metatables or prototypes—the methods are defined directly inside the class.

5.7 ITERATORS AND GENERATORS

An **iterable** is an object that produces a sequence of values. Lists, tuples, sets, dictionaries, and objects of a few other built-in types are iterables. Iterables can be used in `for` statements and a number of built-in functions, including `min`, `max`, `sum`, `all`, `any`, `filter`, `map`, `sorted`, and `zip`.

```
for x in (1,2,3): print(x)        # elements of a tuple
for x in [1,2,3]: print(x)        # elements of a list
for x in {1,2,3}: print(x)        # elements of a set
for c in 'hello': print(c)        # characters of a string

for k in {'x':1, 'y':2, 'z':3}:
    print(k)                      # keys of a dict

with open('colors') as f:
    for line in f:                # lines of a file
        print(line.strip())
```

When Python needs to iterate an iterable, it applies the built-in `iter` function to the iterable, producing (via a call to the iterable's `__iter__` method—remember Table 5.1?) an **iterator**. The iterator's `__next__` method yields the next element of the sequence or raises a `StopIteration` exception when no more elements remain. This means you can write your *own* iterables:

```
class OneTwoThree:
    def __iter__(self):
        value = 0
        class OneTwoThreeIterator:
            def __next__(self):
                nonlocal value
                value += 1
                if value > 3:
                    raise StopIteration()
                return value
        return OneTwoThreeIterator()
```

```
for i in OneTwoThree():
    print(i)
```

Writing iterables and iterators from scratch can be a bit tedious, so Python provides **generators** to simplify the creation of iterable sequences. A generator results from calling a function containing a **yield** statement. Executing such a function does not invoke the function's body, but rather returns a generator object. Here's a generator that produces successive powers of two, up to some limit:

```
def powers_of_two(limit):
    value = 1
    while value < limit:
        yield value
        value += value

# Use the generator
for i in powers_of_two(70):
    print(i)

# Explore the mechanism
g = powers_of_two(100)
assert str(type(powers_of_two)) == "<class 'function'>"
assert str(type(g)) == "<class 'generator'>"
assert g.__next__() == 1
assert g.__next__() == 2
assert next(g) == 4
assert next(g) == 8
```

Calling a generator function creates an object and assigns to it `__iter__` and `__next__` methods. It assembles, on the fly, the code to raise `StopIteration` at the proper time.

Interestingly, we can write an iterator that *never* raises a `StopIteration` exception—in other words, an infinite sequence.

```
def powers_of_two(limit):
    value = 1
    while True:
        yield value
        value += value
```

If the iteration scheme is sufficiently simple, you can create a generator with a **generator expression**, which looks like a list comprehension with parentheses instead of square brackets. Generator expressions have the advantage over list comprehensions of not having to compute and store the entire data set in memory.

```
# List comprehension.
# Computes all of its elements first.
# If large, it's too slow to produce and wastes memory.
bad = [x*x for x in range(10**9)]
```

```
# Generator expression.
# Produces values on demand, during iteration.
# Computed instantly and consumes almost no memory.
good = (x*x for x in range(10**9))
```

The module `itertools` from the standard library includes a number of functions for constructing and manipulating generators, including `permutations`, which builds a generator that produces permutations of a sequence as tuples. Let's use this module to rewrite our permutations script from the beginning of the chapter:

```
import sys
from itertools import permutations

for permutation in permutations(sys.argv[1:]):
    print(' '.join(permutation))
```

5.8 DECORATORS

Time for some magic. We're going to change the behavior of a function *without ever rewriting its source code*. In Python, functions (and hence methods) can be **decorated**, allowing us to do some extra work when the function is called. Here is one of the simplest decorator examples; it prints the time the function is called as well as the duration taken by the call:

```
import datetime

def logged(f):
    def wrapper(*args, **kwargs):
        start = datetime.datetime.now()
        print(f'{f.__name__} started at {start}')
        f(*args, **kwargs)
        duration = datetime.datetime.now() - start
        print(f'{f.__name__} took {duration}')
    return wrapper

@logged
def say_hello(name):
    print('Hellllllooooooooooo, {name}')

say_hello('Alice')
```

Prefixing the function definition of `say_hello` with the `@logged` decorator is equivalent to following its definition with `say_hello = logged(say_hello)`. So the decorator syntax is not essential, but it adds a great deal of expressiveness: all of the logging code is isolated from the mainline code, allowing us to see at a glance (without clutter) what our application should be doing, and knowing it will be logged.

Decorators give us a nice solution to **cross-cutting concerns** such as logging, transaction maintenance in databases, persistence, authentication, and so on. These concerns are called "cross-cutting" because one does not organize a system around authenticated objects vs.

persistent objects; instead these concerns can apply almost anywhere. You will often see Python code with decorators such as `@persistent`, `@transactional`, and `@authenticated`. And here's another. It's well known that the naïve application of the definition

$$F(n) = \begin{cases} n & n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

leads to horrific performance: computing $F(40)$ results in 331,160,281 calls to the function. Over 40 trillion calls are needed to do $F(50)$, and over a septillion (10^{21}) calls for $F(100)$. The problem is that so many calls are repeated, as we can see from the evaluation:

$$\begin{aligned} F(40) &= F(39) + F(38) \\ &= (F(38) + F(37)) + F(38) \end{aligned}$$

As $F(38)$ is called twice, we should record its value after the first call and then simply reuse this value rather than making the second repeated call. This trick of reusing results, known as **memoization**, is applied also to $F(37)$, $F(36)$, and so on, reducing the number of calls from over 331 million down to 41.

Performing memorization within the `fib` function would violate the separation of concerns principle, so we write a decorator:

```
def memoized(f):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        cache[args] = f(*args)
        return cache[args]
    return wrapper

@memoized
def fib(n):
    return 1 if n <= 1 else fib(n-1) + fib(n-2)

print(fib(100))
```

Python decorators can also be parameterized, constructed with a class syntax (any object of a class with a `__call__` method becomes a decorator) and even applied to classes and methods. Some are part of Python's standard library; one of the most common is `@dataclass`.

The dataclass decorator enhances a class that you write by automatically adding an initializer, several special methods (such as equality testing and comparison) according your your wishes, optionally configures it for use in a `match` statement, and can even ensure all instances are immutable.[3] Essentially, the decorator generates a fair amount of boilerplate so you don't have to. Here is a very brief example:

[3] Technically, you can still mutate a frozen object, but it requires writing some pretty strange code to do so.

```python
from dataclasses import dataclass

@dataclass(frozen=True, order=True)
class Point:
    x: float
    y: float

p = Point(1.0, 2.0)  # decorator autogenerates a constructor
q = Point(1.0, 2.0)
assert(p.x == 1.0)   # decorator autogenerates attribute accessors
assert(p.y == 2.0)
# p.x = 5.0          # is an error because Point is frozen
assert(p == q)       # eq=True is a default for the decorator
assert(p >= q)       # we set order=True in the decorator
match p:             # match_args=True is a default for the decorator
    case (a, b):
        assert(a == 1.0 and b == 2.0)
```

PYTHON WRAP UP

In this chapter, we were introduced to Python. We learned that:

- Python is a mature, general-purpose language popular in scripting, web frameworks, and data science. It was created and is still overseen by Guido van Rossum.
- Python has a large number of built-in types, including many different numeric types and over a dozen collection types.
- All values, regardless of type, are considered objects. All objects have an `id` and a `type`. Types are objects.
- New (dynamic) types are created by defining a `class`.
- Classes, as well as all types in Python, are objects. (Contrast this with JavaScript and Lua, where inquiring the type of an object produces the name of the type as a string.)
- Types in Python can have supertypes and subtypes. All paths through the chain of supertypes of an object will end at the type `object`. The supertype-subtype relationships are used in attribute lookup. Subtypes inherit attributes from their supertypes.
- Because types can have multiple supertypes, the language enforces a linear order in which all ancestor types must be searched to find inherited attributes. This is called the method resolution order, or MRO.
- Static types can be specified and checked at compile-time.
- Python local variables are introduced by assignment in a local scope. Nonlocal variables in enclosing lexical scopes can be read; to write to nonlocals either `nonlocal` or `global` must be used.
- Many standard operators and functions can be overloaded by defining special methods in a class. These special methods have names such as `__add__`, `__eq__`, and `__contains__`.

- Python supports delayed evaluation through iterators and generators. Generators can be created by functions with a **yield** statement, or via generator expressions.
- Decorators enable ways to enhance the behavior of functions and classes, without actually adding any code directly inside the function or class.

Our tour of Python has been rather brief. We did not visit any concurrency features of the language, which are quite rich. Python has libraries for in-process threads that can be managed via thread pools and futures, libraries for multiprocessing and spawning new operating system processes, libraries for scheduling, and a library for asynchronous I/O. While we will be looking into concurrency in other languages on our tour (including Java, Go, Rust, and Swift), the reader is encouraged to look into the official Python documentation for examples and explanations of the language's many features.

For further study in Python, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Python, see the Python page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/python.

EXERCISES

Now it's your turn. Continue exploring Python with the activities and research questions below, and feel free to branch out on your own.

5.1 Locate the Python home page and browse the official tutorial, language reference, and library reference.

5.2 Install Python and experiment with the REPL.

5.3 What is the built-in type `enumerate` commonly used for? Construct a realistic example illustrating one such use.

5.4 Extend the `Circle` class from Section 5.2 with methods `circumference`, `__str__`, and `__repr__`. You will need to research the language conventions for the latter two.

5.5 Read *The Zen of Python* [87].

5.6 We mentioned that Python classes are namespaces. But classes are not the only kind of namespace in the language. Research and describe, with examples, at least two other constructs in Python that serve as namespaces.

5.7 In your own words, describe the difference between a function, a method, and a bound method in Python.

5.8 Research the functions `isinstance` and `issubclass`. What do they do? See if you can find arguments for and against their usage in everyday code.

5.9 The term "multiple inheritance," introduced in this chapter, turns out to be an interesting topic because many have argued that programming languages are often better without it. What are the arguments *against* multiple inheritance?

5.10 Give a description of the Python MRO, in pseudocode.

5.11 JavaScript's `let` and Lua's `local` allow one to define a variable scoped to a block (e.g., the body of a `while`-statement). Is this even possible in Python? Why or why not?

5.12 Given the following class:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Add an instance method called `description`, which returns one of the strings "I", "II", "III", "IV", "X-axis", "Y-axis", or "Origin" computed in a `match` statement, computer in a `match` statement using guards for the x and y values of the matched point.

5.13 Rewrite a few of the scripts in this chapter to use type hints.

5.14 Rewrite the command line permutations program from the beginning of the chapter to require the arguments of `print_permutations` to be passed positionally. Rewrite the `Circle` class from this chapter to require the `radius` parameter to be associated with a keyword argument.

5.15 Perform a detailed comparison between the `Vector` class implemented in this chapter and the `Vector` table written in Lua in Chapter 2.

5.16 If you have some experience with spatial mathematics, create Python classes for points, vectors, rays, and planes, "overloading" as many operators as is convenient. Your classes should include functionality, to among other things, subtract two points to produce a vector, and a point and a vector to get another point, multiply a vector by a number to change its magnitude, compute the dot and cross products of two vectors, and determine the normal vector of a plane.

5.17 Find out, how, in Python to (a) get all of the attributes of a class, (b) get all of the subclasses of a class, and (c) whether or not it is possible to add new attributes to an existing class.

5.18 Research *classmethods* and *staticmethods* in Python. What are they? How are they implemented?

5.19 What is module in Python? (Research is required, as we did not cover this topic.) How do you, programmatically, get the contents of a module?

5.20 In this chapter we saw an example of packing keyword arguments, but no examples of unpacking a dictionary in a call. Supply a realistic example.

5.21 Can a function have both single-starred and double-starred parameters? If so, how many of each are allowed? How might using both ever be considered useful?

5.22 The `OneTwoThree` example in this chapter used separate classes for the iterator and the iterable. Rewrite the example so there is only one class for both. Hint: the `__iter__` method will return `self`. What are the advantages and disadvantages relative to the solution presented in this chapter?

5.23 Is a Python `range` an `iterator`? If not, why might one be inclined to think it is?

5.24 Do a compare-and-contrast study between Lua coroutines and Python generators. Also compare and contrast JavaScript's generators with Python's. Make sure to cover any subtle differences you may notice in the semantics of the `yield` keyword among the three languages.

5.25 Use the `timeit` module to time both the unmemoized and memoized versions of the fibonacci function from this chapter when called with the argument 40.

5.26 A classic problem in computer science is the *0-1 knapsack problem*: Given a sack with capacity C and a set of items $\{(w_1, v_1)...(w_n, v_n)\}$ with weights w_i and values v_i, find the most valuable selection of items with total weight $< C$. There is a straightforward recursive solution, which, like the fibonacci example, ends up making multiple recursive calls of the same argument. Implement the straightforward solution. Generate a random set of items with random weights and values, and time your solution both with and without the `@memoized` decorator.

5.27 Consider the following slight modification to the Vocalizer class from this chapter.

```
import subprocess
class Vocalizer:
    def vocalize(self):
        subprocess.call(['say', str(self)])
```

Now simply adding `Vocalizer` to a classes supertype list should enable any object of that class to have its string representation vocalized. Try it. Can the same functionality be done through decorators? That is, can you decorate a class (perhaps with `@vocalized`) such that the decorator adds a vocalize method?

5.28 Research the class syntax for decorators and write a `@memoized` decorator using this syntax.

5.29 Implement a Python (frozen) dataclass for a playing card. Each card should have a rank (a character from A23456789TJQK) and a suit (spades, hearts, diamonds, clubs). Make sure the class initializer rejects illegal values for rank and suit. Implement a `__str__` method to return the Unicode character representing the card.

5.30 We did not cover Python's approaches to asynchronous computing. Research the use of `async` and `await` and illustrate their use in a small example application of your choice.

5.31 We also did not cover Python's library for thread-based parallelism. While threads will be covered in later chapters in this book, try your hand at some simple thread-based programs of your own design. Your research will take you into locks, conditions, semaphores, timers, and barriers. You may also encounter thread pools and queues.

5.32 If you did the previous two exercises, how would you characterize the difference between threading and the `asyncio` library?

CHAPTER 6

Ruby

Ruby's designer, Yukihiro Matsumoto (Matz), expressed his language's purpose as follows: "I hope to see Ruby help every programmer in the world to be productive, and to enjoy programming, and to be happy. That is the primary purpose of Ruby language." [70]

First appeared	1995
Creator	Yukihiro Matsumoto
Recognized for	OOP, Metaprogramming, DSL construction
Notable uses	Web servers, Scripting
Tags	Object-oriented, Expression-oriented, Dynamic
Six words or less	"A Programmer's Best Friend"

Matz designed Ruby in the mid-1990s as a language in which one could be productive and have fun programming. Rather than building a language with a small, simple core (for which Lisp and Smalltalk are famous), Matz looked to be pragmatic, and aimed for human-friendliness. The language has many fans who find the language beautiful, powerful, and expressive.

Many programmers today know Ruby from the web frameworks built upon it. But Ruby is truly a *general-purpose* language. It's a scripting language (influenced by Perl), with operating system interfaces, string processing, regular expressions, and various syntactic niceties like flexible quoting syntaxes. It is object-oriented in the way (another influencer) Smalltalk is object-oriented, expressing nearly all computation with message passing. Ruby's blocks come from Smalltalk, too. And the metaprogramming facilities of Ruby are, as in Smalltalk, an area in which the language really shines. Ruby's uncanny ability to implement domain-specific languages (DSLs) also fuels its popularity across domains.

Our Ruby tour will begin with our standard three example programs and a brief overview of the language. We will then introduce the object-oriented paradigm and show how Ruby makes object-oriented programming not only possible but natural. We'll then proceed to two important features of Ruby, blocks and mixins, and continue with a section on access control, a feature Ruby makes relatively easy compared to the four languages we have seen before. We'll round out our coverage with Ruby's metaprogramming facilities.

6.1 HELLO RUBY

We begin with our familiar script to list small right triangle measurements:

```
1.upto(40) do |c|
  1.upto(c - 1) do |b|
    1.upto(b - 1) do |a|
      puts "#{a}, #{b}, #{c}" if a * a + b * b == c * c
    end
  end
end
```

Here we see convenient syntactic features such as string interpolation with `#{}` and a trailing `if`-clause, as well as an iteration construct that is definitely not a traditional `for`-statement. Instead, we start the script by sending the **message** `upto` to the **object** `1`. The message is sent with the argument `40` and a **block** (here delimited with `do` and `end`). The `upto` method executes its block for each value in the range from its receiver up to, and including, its argument.

Next up is the command line permutations using Heap's Algorithm:

```
def print_permutations(a, n)
  if n <= 0
    puts a.join "\t"
  else
    0.upto(n-1) do |i|
      print_permutations a, n-1
      j = n.even? ? 0 : i
      a[j], a[n] = a[n], a[j]
    end
    print_permutations a, n-1
  end
end

print_permutations(ARGV, ARGV.length - 1)
```

The constant `ARGV` for the command line argument array does not need to be explicitly imported or tagged with a module name. And for something new, calls do not always require parentheses.

Like Python, Ruby already knows how to do permutations, so the entire script above could be done in one line:

```
ARGV.permutation.each {|s| puts s.join "\t"}
```

The `.permutation` method produces an *enumerator* for all of the permutations of the command line argument array. An enumerator is not an actual array but a generator producing values on demand. The `each` method requests the values, which are passed to the block where the permuted array is joined up into a string and written on a line to standard output.

Here is the data science application to compute the top ten scorers:

```
players = []

ARGF.each do |line|
  team, name, games, points = line.strip.split ','
  if games.to_i >= 15
    players += [[name, team, points.to_f/games.to_i]]
  end
end

players.sort {|p1, p2| p2[2] - p1[2]}[0, 10].each do |player|
  puts "%-22s%-4s%8.2f" % player
end
```

This short, flowing script shows how writing Ruby is, for many, pleasant and productive. `ARGF` was created with the scripter's happiness in mind. If no command line arguments are present, `ARGF` is standard input; if arguments are present, they're treated as filenames from which input will be read (in order). We pull each line from standard input, split to get the data components, and construct an array of arrays with player information (for those players appearing in more than 15 games). The second statement sorts by points-per-game descending, takes the first 10 players, and prints according to the desired specification.

6.2 THE BASICS

Ruby has variables, assignments, the usual control structures, and even exceptions. As in Python, all variables store references to values; you can find the unique identifier of any object x with x`.object_id`. Every object has a unique class:

```
fail unless nil.class == NilClass
fail unless false.class == FalseClass
fail unless true.class == TrueClass
fail unless (2**1000).class == Integer
fail unless 2.0.class == Float
fail unless :dog.class == Symbol
fail unless "dog".class == String
fail unless (1..5).class == Range
fail unless [1,2,3,4,5].class == Array
fail unless {x: 1, y: 2}.class == Hash
fail unless {'x' => 1, 'y' => 2}.class == Hash
```

Ruby features a large number of built-in classes. A class can have at most one **superclass**. Superclasses allow us to express `IS-A` relationships: each of the expressions `5.is_a? Integer`, `5.is_a? Numeric`, `5.is_a? Object`, and `5.is_a? BasicObject` evaluate to `true`. Figure 6.1 shows part of this hierarchy.

Like Python, Ruby classifies floating point numbers differently than integers, and allows integer values to grow as large as available memory will allow them to grow: computing `2 ** 100` happily produces 1267650600228229401496703205376. The language is "strongly typed" in the sense of allowing very few implicit coercions: you can't, for example, add strings and numbers. But everything *can* be coerced to a boolean. Like Lua, and very unlike Python, the *only* expressions that are falsy are `false` and `nil`.

```
                    BasicObject
                        │
                      Object
                        │
┌──────────────────────┬──────────────────────┬──────────────────────
├─ Module              ├─ Array               ├─ Proc
│   └─ Class           ├─ Hash                ├─ Method
├─ NilClass            ├─ Set                 ├─ UnboundMethod
├─ FalseClass          ├─ Struct              ├─ Thread
├─ TrueClass           ├─ Data                ├─ ThreadGroup
├─ Numeric             ├─ Regexp              ├─ Thread::ConditionVariable
│   ├─ Integer         ├─ Range               ├─ Thread::Queue
│   ├─ Float           ├─ IO                  ├─ Thread::Mutex
│   ├─ Rational        │   └─ File            ├─ Fiber
│   └─ Complex         ├─ Dir                 ├─ Ractor
├─ String              ├─ Enumerator          ├─ Monitor
├─ Symbol              ├─ Random              ├─ TracePoint
├─ Date                ├─ Binding             ├─ RubyVM
└─ Time                └─ Encoding            └─ Exception
```

Figure 6.1 Part of the standard class hierarchy in Ruby

Ruby distinguishes strings and **symbols**. There is one—and only one—symbol `:dog` in the object pool, but there can be many strings with the three characters `"dog"`. Symbol names begin with a colon unless used as a hash key separating its value with a colon, e.g., `{count: 0}` sugars `{:count => 0}`. You can use nonsymbols as hash keys under certain conditions (to be explored in an end-of-chapter exercise), but only with the "hash rocket" (`=>`) or `Hash.new`. Like JavaScript, Ruby supports a shorthand notation for hash entries: `{x:, y:}` sugars `{x: x, y: y}`. Perhaps due to the string-symbol separation, Ruby's strings, unlike those of most other languages, are mutable.

Ruby has much in common with the language of the last chapter Python. Python's pattern matching `match-case` in Ruby is `case-in`; Python's `try-except` is `do-rescue` in Ruby; Ruby also shares with Python positional vs. keyword arguments, default parameters, and handles arbitrary number of arguments:

```ruby
def positional(x, y, z) = "#{x} #{y} #{z}"
def named(x:, y:, z:) = "#{x} #{y} #{z}"
def splatted(x, *y) = "#{x} #{y}"
def defaulted(x, y=100) = "#{x} #{y}"
def defaulted_and_named(x, y:2) = "#{x} #{y}"

fail unless positional(1, 2, 3) == "1 2 3"
fail unless named(z:3, x:1, y:2) == "1 2 3"
fail unless splatted(1) == "1 []"
fail unless splatted(1, 2, 3, 4) == "1 [2, 3, 4]"
fail unless positional(1, *[2, 3]) == "1 2 3"
fail unless defaulted(1) == "1 100"
fail unless defaulted(1, 2) == "1 2"
fail unless defaulted_and_named(1) == "1 2"
fail unless defaulted_and_named(1,y:3) == "1 3"
```

Here the method `positional` must be called with exactly three arguments, in order. `named` must be called with exactly three arguments as well. The names must be present in the call, but they can appear in any order. Callers must pass *at least one* argument to `splatted`; all arguments beyond the first are packed into the array bound to y. The method `defaulted` must be called with either one or two arguments, and ditto for `defaulted_and_named`. It's probably worth mentioning that positional arguments should not be called with names—the call will be interpreted as something very strange.

Ruby has a number of other convenient, and pragmatic, features that make programming fun. Ranges are actual objects, with `..` including the end value and `...` excluding it; ranges can even be beginless (e.g., `(..3)`) or endless (e.g., `(1..)`). Small, fully immutable, value types (with shallow copy and shallow equality comparison with `==`) are easy to define:

```
Point = Data.define :x, :y
p = Point[1, 2]
fail unless p.x == 1 && p.y == 2
fail unless p.with x: 3 == Point.new(3, 2)
```

For control flow, Ruby sports not only `while` and `if`, but `until` and `unless`. And the language has not two, but *four* ways to disrupt a loop:

- `break`: Break out of the loop completely
- `next`: Immediately begin the next iteration of the loop
- `redo`: Restart the current iteration of the loop (without retesting a condition, if any)
- `retry`: Restart the entire loop from the first iteration

We really can't go much farther into our exploration of Ruby without covering the fact that the language is something we call object-oriented, in an essential way. It's time to see what that means.

6.3 OBJECT ORIENTATION

A **programming paradigm** is a style of expressing computations. The **functional paradigm** views computation as the composition of functions: the output(s) of one function are fed into another, and so on. The **imperative paradigm** views computation as a succession of actions, such as assignment statements and I/O operations, updating a global state. The **logic paradigm** views computation as the systematic satisfaction of constraints, specified by facts and inference rules. The **event-driven paradigm** builds a program from a series of event handlers that run in response to user actions, sensor inputs, or explicit firings. Wikipedians have described dozens of additional programming paradigms in the *Programming Paradigm* article [122].

The popular **object-oriented paradigm** views computation as message passing among objects: an object responds to a message (when it can) by executing a **method**, possibly updating the object's internal state. The key insight of object orientation is that a computer (or a complex software system) can be decomposed into smaller computers (objects) rather than into the dissimilar and less powerful data structures or procedures.[1] There is a

[1] Alan Kay, who is said to have invented the term "object oriented," credits the genesis of this idea to his teacher, Bob Barton, who said "The basic principle of recursive design is making the parts as powerful as the whole." [60]

spectrum of opinions on what characterizes object orientation, from simply bundling code inside of data structures on one end to specifically requiring that (1) objects have completely encapsulated (hidden) state, (2) objects be instances of classes, and (3) classes be grouped into inheritance hierarchies allowing for specialization of method behaviors. An excellent overview of object orientation covering its goals, origins, mechanisms, models, and alternatives, is [116]. Ruby is an object-oriented programming language because it facilitates object-oriented programming. Let's see how Ruby models messaging, encapsulation, inheritance, and polymorphism.

Like Python, all Ruby values are objects, even simple numbers and booleans, and with few exceptions, computation is carried out exclusively by sending messages to objects. Send a message with the expression *obj*`.send(`*message*`,`*args*`)`, denoting the message with an expression evaluating to a Ruby symbol (or a string literal, which will be automatically converted to a symbol.) If the message is known for sure, you can use the shorthand *obj*`.`*message*`(`*args*`)`.

```
# Zero-argument messages
fail unless 5.send(:abs) == 5
fail unless 5.send('abs') == 5
fail unless 5.abs == 5

# A one-argument method
fail unless 5.send(:+, 3) == 8
fail unless 5.send('+', 3) == 8
fail unless 5.+(3) == 8
fail unless 5 + 3 == 8

# Messages can be stored in variables
operator = '-'
fail unless 5.send(operator, 2) == 3
```

Like Python, Ruby's objects are instantiated from classes. Unlike Python, which exposes all of an object's attributes (both callable and noncallable), Ruby distinguishes **instance variables** (that hold state) from **methods** (which *respond* to messages), and does not allow code outside of the class to directly access the object's internal variables. *All* object interaction must occur through methods. The **state** of the object is therefore fully **encapsulated**.

Let's illustrate with a little circle class:

```
class Circle
  def initialize(x, y, r)
    @x = x
    @y = y
    @r = r
  end

  def center() = [@x, @y]

  def area() = Math::PI * @r * @r
```

```
  def to_s()
    "Circle at (#{@x}, #{@y}) with radius #{@r}"
  end
end

c = Circle.new(5, 3, 10)
fail unless c.center == [5, 3]
fail unless c.area == 100 * Math::PI

# to_s is automatically called when in string context
fail unless "#{c}" == "Circle at (5, 3) with radius 10"
```

The definition states a circle (1) is initialized with two center coordinates and a radius, (2) responds with an array of its center coordinates when queried,[2] (3) responds with its area when queried, and (4) can produce a string representation of itself. The code outside the class illustrates the creation of a circle by sending the message `new` to the class `Circle`. The variable *c* references the new object, to which we send `center` and `area` messages. There's a little magic here: Ruby automatically sends `initialize` upon object creation and `to_s` when an object is used in a string context.

An instance variable's name begins with `@` and is only accessible *within* the class. Were we to try to evaluate `c.@r` outside the class, a `SyntaxError` exception would be raised. And the expression `c.r` raises a `NoMethodError` since the dot notation always refers to a method call. If we wanted to make the radius available to a query (e.g., `c.r`) we would have to supply a method to respond with it:

```
def r()
  @r
end
```

If you wish to write a method that does nothing but update an instance variable, Ruby lets you write:

```
def r=(new_radius)
  @r = new_radius
end
```

and this method will be invoked whenever `c.r` appears on the left of an assignment statement. Incidentally, you can let Ruby create these methods for you so you don't have to type them. Writing `attr_reader :r` creates the first method, `attr_writer :r` the second, and `attr_accessor :r` creates both.[3]

As mentioned earlier, simply bundling operations with encapsulated data counts as "object oriented" in some circles. Cardelli and Wegner [13], however, define the term as having an additional ingredient, *inheritance* of instance variables and methods by subclasses. Here's our recurring example of inheritance seen in some previous chapters:

[2] The `return` keyword is optional. When omitted, the value of the last evaluated expression in a method body is returned.

[3] Just because Ruby allows you to easily write methods to directly read and write instance variables does not mean you should. The use of *getter* and *setter* methods is controversial because it violates encapsulation, exposing implementation details to callers, and encourages clients to pull data from an object to perform work which the object could (should?) have done itself. See [50] for details.

```
class Animal
  def initialize(name)
    @name = name
  end
  def speak()
    "#{@name} says #{sound()}"
  end
end

class Cow < Animal
  def sound() = "moooo"
end

class Horse < Animal
  def sound() = "neigh"
end

class Sheep < Animal
  def sound() = "baaaa"
end

if __FILE__ == $0
  s = Horse.new "CJ"
  fail unless s.speak == "CJ says neigh"
  c = Cow.new("Bessie")
  fail unless c.speak == "Bessie says moooo"
  fail unless Sheep.new("Little Lamb").speak == "Little Lamb says baaaa"
end
```

Cows, horses, and sheep inherit the `initialize` and `speak` methods from their superclass, as well as the instance variable `@name`. (We've also illustrated two of Ruby's global variables `__FILE__`, the currently executing file, and `$0`, the script being run. The code at the end of the file only executes when the file is run directly and not imported.)

While all kinds of animals in our previous example have names and speak the same way, they all make different sounds. The expression `a.sound` will produce the proper sound of the animal referenced by the receiver *a*, even if we don't know the class of that animal until the script is run. This behavior is called **dynamic polymorphism**. The name `sound` is *polymorphic* (Greek: "of many forms") since it is bound to three entities, and the dispatch to the correct method is *dynamic* (happens at run-time).

6.4 SINGLETON CLASSES

In each of the four languages of the previous chapters, we have the ability to attach functions directly to objects, regardless of the object's class or prototype. But Ruby objects don't contain slots for methods—all methods live in classes. Creating object-specific methods would seem impossible; however, Ruby gives each object its very own **singleton class** (sometimes called an **eigenclass**) to hold object-specific methods. Let's create a dog class and two dogs. All dogs can bark, but we want Spike, and only Spike, to bite:

```
class Dog
  def bark() = "woof"
end

spot = Dog.new
spike = Dog.new

def spike.bite() = "ouch"

spot.bark
spike.bite
```

Spike can bite because method lookup starts at an object's singleton class and then proceeds up the superclass chain. Figure 6.2 shows how these objects are related.

Figure 6.2 An object with a singleton class

6.5 BLOCKS

Every method in Ruby may be passed, in addition to its arguments, a **block**. We saw blocks in this chapter's opening example delimited with `do` and `end`, but curly braces work fine too:

```
3.times {puts 'Hello'}
```

Blocks may be parameterized (the parameters are placed between vertical bars at the beginning of the block):

```
mountains = 'Everest K2 Kangchenjunga Lhotse Makalu'.split
mountains.each_with_index {|name, i| puts "#{i+1}. #{name}"}
```

Block parameters have their own scope and shadow variables of the same name in enclosing scopes. Code within the block body *can* both read and mutate variables in the enclosing scope. In addition to parameters, blocks may have their own block-scoped variables. These

appear after a semicolon in the block's "header," and also shadow variables in the enclosing scope:

```
def main
  x = 100                                # x is a local var of main
  [1,2,3].map{|x| x * x}                 # a new block-local x
  fail unless x == 100                   # "outer" x unchanged at 100
  [1,2,3].each{|y| x = x + y}            # block can mutate outer x
  fail unless x == 106                   # x mutated to 106
  [1,2,3].each{|y; x| x = (x||0) + y}    # ; makes new x in block
  fail unless x == 106                   # x unchanged at 106
end

main
```

To have a method transfer control to its block, use `yield`. Let's see how this works. The following method generates powers of two up to some limit. What happens to those powers of two depends on the block supplied to the method call.

```
def powers_of_two(limit)
  x = 1
  while x <= limit
    yield x
    x += x
  end
end

powers_of_two 90 do |x|
  puts x
end
```

Blocks should remind you of anonymous functions. They have their own parameters and local variables, and can manipulate variables in the enclosing scopes. But to use blocks like first-class functions (assign them to variables and pass them around), we must create `Proc` objects from them. There are two kinds of `Procs`, regular procs and lambda procs; and a variety of ways to denote them:

```
plusTwo = proc {|x| x + 2}
plusThree = Proc.new {|x| x + 3}
plusFour = lambda {|x| x + 4}
plusFive = -> (x) {x + 5}

# How do we call them? What are they? Regular or lambda?
p [plusTwo.call(10), plusTwo.class, plusTwo.lambda?]
p [plusThree.call(10), plusThree.class, plusThree.lambda?]
p [plusFour.call(10), plusThree.class, plusFour.lambda?]
p [plusFive.call(10), plusThree.class, plusFive.lambda?]
```

```
# Extra args are no problem for procs
plusTwo.call(100, 200, 300)

# ...but they are a problem for lambdas
begin
  plusFour.call(100, 200, 300)
rescue ArgumentError
  puts 'Rescued'                # Message will be output
end

a = [10, 20, 30, 40, 50]
p a.map(&plusTwo)               # All passed the same way
p a.map(&plusThree)
p a.map(&plusFour)
p a.map(&plusFive)
p a.map(&(->(x) {x * 10}))      # anonymous procs!
p a.map &->(x) {x * 10}         # Parens optional here too
```

The first form (`Proc.new` or `proc`) should be used sparingly, if at all—these kind of procs don't check the number of arguments passed to them, and a `return` statement actually returns from the enclosing method! The lambda forms behave as expected.

```
def conversation()
  puts 'Calling a lambda proc'
  (lambda {return}).call              # Returns from the proc
  puts 'Calling a non-lambda proc'
  (proc {return}).call                # Returns from conversation()
  puts 'This does not appear'
end

conversation
```

There's a neat little shorthand that you can use when the block contains a single expression that passes a message to the block parameter. The following two lines do the same thing:

```
[1,-2,3,-4,-5].map {|x| x.abs}
[1,-2,3,-4,-5].map &:abs
```

The unary ampersand (`&`) causes the `toProc` message to be sent to a symbol. The `Symbol` class defines a `to_proc` method to do the right thing. If you wish, you can create your own classes and define a `to_proc` method, enabling your own objects to work on methods expecting blocks.

6.6 MIXINS

In the previous chapter, we saw Python's multiple inheritance in action, with the vocal cow and its two supertypes, `Cow` and `Vocalizer`. Ruby classes may have only one superclass, but they may include any number of **modules**. A Ruby module is a container for methods; a

class is a module that is allowed to have instances. Its methods are **mixed in** while defining a class. Here's the vocal cow example from the Python chapter, translated to Ruby:

```
require './animals.rb'

# The original Animal class did not define a name method.
# But in Ruby, we can *add* one! Classes are *open*.
class Animal
  attr_reader :name
end

module Vocalizer
  # Disclaimer: This only works on an O.S. with a say command.
  def vocalize()
    # Multiple arguments required to prevent shell injection attack.
    system 'say', self.name, ' says ', self.sound
  end
end

# A VocalCow will have all the methods of Cow and Vocalizer.
class VocalCow < Cow
  include Vocalizer
end

c = VocalCow.new 'Bessie'
puts c.speak
c.vocalize
```

Ruby has a number of built-in modules. One of the most useful is `Enumerable`, with methods `map`, `select`, `reduce`, `count`, `find`, `detect`, `first`, `min`, `max`, `sort`, `include?`, `all?`, `any?`, `one?`, `take`, `drop`, `cycle`, `find_all`, `group_by`, and many others. The following built-in classes include `Enumerable`:

- Array
- Hash
- Range
- IO
- Dir
- Enumerator

For example, to find all of the array elements, or hash keys, or lines in a file (`File` is a subclass of `IO`, the class of input-output streams), or files in an operating system folder (called a `Dir`), that begin with the letter `'A'`, you need only say:

```
obj.filter {|s| s.start_with? 'A'}
```

where *obj* is an instance of any one of those classes. How does this work? Each of the methods in `Enumerable` relies on the object providing an implementation of the method `each`. We can take advantage of this in classes we write ourselves. As long as we define `each` and include `Enumerable`, our objects can be mapped, filtered, reduced, and so on.

```
class OneTwoThree
  include Enumerable
  def each
    yield 1
    yield 2
    yield 3
  end
end

t = OneTwoThree.new
fail unless t.first == 1
fail unless t.map {|x| x * 100} == [100, 200, 300]
fail unless t.select {|x| x.even?} == [2]
fail unless t.all? {|x| x > 0} == true
fail unless t.one? {|x| x > 2} == true
fail unless t.reduce(0, :+) == 6
fail unless t.take(2) == [1, 2]
```

Two other important predefined modules are:

- `Comparable`, which defines `<`, `<=`, `>`, `>=`, `==`, and `between`. Each of these are implemented to invoke the receiver's `<=>` method, which is intended to work as follows: `a <=> b` should produce -1, 0, $+1$, or `nil` when a is less than, equal to, greater than, or not comparable to, respectively, b. It is mixed into the numeric, string, and time classes.
- `Kernel`, which defines dozens of useful methods, including `object_id`, `==`, `send`, `is_a?`,[4] `chop`, `rand`, `freeze`, `pretty_print`, `exit`, `abort`, `puts`, `system`, `require`, `sub`, `sleep`, and `lambda`. This module is mixed into `Object`, which is an ancestor class of every class (except `BasicObject`), so these methods are universally available.

Modules become "parents" of a class, just like the superclass, for the purpose of method lookup. So Ruby, like Python, must deal with a potentially complex ordering of its ancestors when looking up methods. Ruby looks up methods in this order:

1. The object's singleton class.
2. The modules mixed into the singleton class. Later `include` statements take precedence over earlier ones, but in the construct `include A,B`, A takes precedence over B.
3. The object's class.
4. The modules mixed into the class.
5. The class's superclass.

These rules are applied recursively.

[4] By now, you've seen quite a few method names ending in a question mark. Generally, methods returning `true` or `false` will be named this way; similarly, methods that mutate a receiver will often have names ending in an exclamation mark. Alas, these are only conventions, and not enforced by the language.

6.7 ACCESS CONTROL

With the exception of various control structures (e.g., `while`, `break`, `retry`) and certain special operators (`&&`, `||`, `..`, `...`, `?:`, `and`, `or`, `not`, `defined?`, and the assignment operators), all computation in Ruby is carried out by sending messages to objects.

When applications become complex, we find it useful to have control over when and how methods can be invoked. In Ruby, **public methods** (the default visibility) can be invoked on an object from anywhere. **Protected methods** can only be invoked from within methods of the object's class or its subclasses. **Private methods** can only be invoked without an explicit receiver. We'll explain.

Let's consider a simple account class that logs deposits and withdrawls. (For simplicity, we'll stick to deposits only and log with `puts`.) We can send a `deposit` message to an account, which adjusts the balance and sends a `log` message to *itself*. The expression `self` refers to the object currently executing a method, so here's our first pass:

```
# A demonstration of what not to do. All methods are public.
class Account
  def initialize; @balance = 0; end
  def deposit(amount); @balance += amount; self.log(amount); end
  def log(amount); puts "added: #{amount}, balance: #@balance"; end
end

a = Account.new
a.deposit(100)     # This is fine, we want to make deposits.
a.log(-999)        # Noooo! Shouldn't log from outside the class!
```

Whoops. Logging should not available to anyone. Let's try `protected`:

```
# An attempt to fix the account. But it still can be subverted.
class Account
  def initialize; @balance = 0; end
  def deposit(amount); @balance += amount; self.log(amount); end
protected
  def log(amount); puts "added: #{amount}, balance: #@balance"; end
end

a = Account.new
a.deposit(100)
a.log(-999) rescue puts "Whew: Can't call protected method"
```

That helped a little, but we still have a problem. While we cannot call protected methods from outside the class, we *can* make calls from within the class *or in subclass methods*. While this may be useful in cases where a method needs to manipulate internal variables among multiple objects of a class hierarchy, it does allow one to construct a malicious subclass to invoke the `log` method on our existing account:

```
# But protected does not help you from subclasses
class BadAccount < Account
  def mess_up_log(otherAccount); otherAccount.log(-999); end
```

```
end

BadAccount.new.mess_up_log(a)    # Oh no!
```

It's time for `private`. Private methods can only be sent to `self`. Ruby enforces this by making us write *method*(*args*) instead of *object*.*method*(*args*)[5] and defines the *implicit* receiver to always be `self`.

```
# Making the log method private prevents the previous attack
class Account
  def initialize; @balance = 0; end
  def deposit(amount); @balance += amount; log(amount); end
private
  def log(amount); puts "added: #{amount}, balance: #@balance"; end
end

a = Account.new
a.deposit(100)
a.log(-999) rescue puts "Whew: Can't call private method"
BadAccount.new.mess_up_log(a) rescue puts 'SAFE!!'
```

To be honest, we didn't really make our accounts perfectly secure; there are ways to circumvent access controls.[6] Access controls can even be changed at runtime. But we did show how to use the language to express our intent that `log` was part of an account object's internal implementation and not part of an external interface.

The ability to call methods without a receiver is the key to making Ruby look like it has functions when in reality it has only methods. Whenever we define a method outside of any class (like we have done on more than one occasion throughout this chapter!), Ruby makes it private method of the class `Object`. Let's verify this:

```
def hello = "hello"
fail unless Object.private_instance_methods.include? :hello
```

As the class `Object` is an ancestor of every class, except those explicitly derived from `BasicObject` (see Figure 6.1), its private methods can be used almost anywhere. We get the appearance of top-level functions, but internally everything is object-oriented message passing.

6.8 METAPROGRAMMING

Ruby lets us create classes to model the things we care about. In a typical commerce application, we define customers, items, orders, and purchases. In a calendar application, we create contacts and appointments. We write methods to ask questions such as "How many customers have used a coupon in the last 30 days?" and perform operations such as "Reduce the price of all volleyball shoes by 20 percent." But since classes, modules, methods, and procs are objects, we can query and manipulate them too. Querying and manipulating the program itself is called **metaprogramming**.

[5] There is one exception to this rule: methods whose names end in `=`.

[6] One way to do this is simply write, `a.send(:log, -999)` since `send` does not respect access controls.

The simplest kind of metaprogramming, querying the program elements, is called **introspection**. Ruby provides a large number of introspective methods. Among them, for class c, class or module m, and object o, are:

- c.`superclass`
- m.`name`
- m.`included_modules`
- m.`instance_methods`
- m.`public_instance_methods`
- m.`protected_instance_methods`
- m.`private_instance_methods`
- m.`constants`
- o.`class`
- o.`singleton_class`
- o.`is_a?`
- o.`respond_to?`
- o.`methods`
- `local_variables`
- `global_variables`

The first few methods tell us about *classes*—the class of an object, the superclass of a class, and the methods defined by a module or class. The `respond_to?` and `methods` methods are about *type*. Generally speaking, the type of an expression constrains its behavior; applying operations to the wrong kind of operands, like multiplying symbols with hashes, are type errors, not class errors. Since nearly all Ruby "operations" are essentially method invocations, asking an object whether it can respond to a given message, or asking which methods it does respond to, is a type query, not a class query. Consider:

```
def increment(x)
  x + 1
end
```

You may ask "what types of arguments are allowed for this method?" The answer is that we can pass any object that responds to the `:+` message with the argument `1`.[7] The specific class of the object does not matter; all that matters is the object understands the message.[8]

The last two methods on this list are quite interesting, as Ruby's introspective capabilities allow you to find out which variables—local and global—are currently in scope. Global variable names always start with a `$`. Ruby comes with several dozen predefined globals, including `$;`, `$!`, `$~`, `$0`, `$FILENAME`, `$.`, `$\``, `$&`, `$'`, `$PROGRAM_NAME`, `$*`, `$-a` and many more. You can create new globals by simply assigning to them.[9] Let's explore:

```
$x, $y = 1, 2            # globals
x, z = 10, 20            # private variables of main object

def example()
```

[7] That's because `x + 1` is sugar for `x.+(1)`.

[8] To fully understand why the class does not matter, remember that in Ruby you can add methods to a specific object, regardless of its class.

[9] But don't. There are almost always better ways to solve problems than to bring in global variables.

```
  x, y = 100, 200        # local variables
  p $x, x, $y, y         # globals and locals do not conflict
  p z rescue p 'No z'    # unlike blocks, methods don't capture
  p local_variables      # [:x, :y]
  p global_variables     # long list, including :$x and :$y
end

example
```

The second kind of metaprogramming involves (programmatically) changing the program itself. We can write code that adds or removes classes, adds or removes methods, and adds or removes instance variables. In fact, class definition and even method definition are actually done with message passing:

```
C = Class.new do
  define_method (:f) {|x| x + 1}
end

puts C.new.f(8)
```

This is pretty interesting, so let's break it down.

- We create a class by sending `new` to the class `Class` together with a block.
- We send `define_method` (it's a private method, so we don't mention the receiver) with a symbol for the method name and a block with the method's implementation.

The example shows that message passing is truly fundamental, and Ruby's use of the `class` expression, the use of `def`, and other fancy (but useful!) constructs are, more or less "just syntax."[10] Entire object-oriented languages can be built with hardly any syntax at all if desired—variables, messages, and parameters should suffice. If you find this approach to language design interesting, take a look at Io [21].

RUBY WRAP UP

In this chapter, we were introduced to Ruby. We learned that:

- Almost all computation is done via message passing and an object's state is encapsulated, allowing Ruby to be called an object-oriented programming language.
- All objects in Ruby are instances of some class. Classes can have a single superclass and can have multiple subclasses. Method dispatch is dynamic.
- In addition to an object's class, an object may have a singleton class, in which to store object-specific behavior.
- Methods may be passed blocks in addition to their regular arguments. A method executes its block via the `yield` expression. Blocks provide the foundation for functional programming in Ruby.
- Blocks are closely related to procs. Procs have both lambda and nonlambda forms.

[10] This isn't exactly true, because the scoping rules kick in. Blocks can capture variables in their environments whereas class expressions do not.

- While a class can have only one superclass, it may mix in many modules. (A class is just a module that can have instances.) This means classes can inherit methods from multiple ancestors, and a method resolution algorithm has been defined.
- The two most useful standard modules are `Enumerable`, which provides a means to iterate through arrays, hashes, directories, files, and the like, and `Kernel`, which provides dozens of methods applicable to any object. `Kernel` is mixed in to `Object`, which is the ancestor of (nearly) every other class.
- Ruby methods can be public, protected, or private. Private methods (with one tiny exception) cannot be called with an explicit receiver, enabling code to read as if it had global functions. Access controls can be subverted.
- Ruby has a wealth of metaprogramming facilities. Objects, classes, modules, and other entities can be introspected. Classes and even methods can be created programmatically at runtime.

Our tour of Ruby has been rather brief. Our major omissions are perhaps regular expressions and concurrency features. The Ruby programmer has several options for concurrency, including threads (for preemptive, shared-memory concurrency), fibers (for cooperative threading), and ractors (which communicate exclusively through messages).

For further study in Ruby, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Ruby, see the Ruby page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/ruby.

EXERCISES

Now it's your turn. Continue exploring Ruby with the activities and research questions below, and feel free to branch out on your own.

6.1 Experiment with the Ruby REPL.

6.2 The class hierarchy shown in Figure 6.1 is incomplete. The class `Exception`, for example, is the root of a sizable hierarchy of classes. What does this hierarchy look like? Are any other classes missing?

6.3 Familiarize yourself with the Ruby documentation. Find information on the following methods: `to_s`, `to_i`, `to_a`, and `to_h`. In which class or module are each of these defined? What do they do?

6.4 In Python, we saw that `7/2==3.5`, `7//2==3`, and `7.0//2==3.0`. The latter two expressions use the floor division operator. Does Ruby have a floor division operator? If so, what is it? If not, how we do simulate it?

6.5 Research the following operators: `===`, `<=>`, `**`, `&.`, and `=~`. Write a short script (or scripts) that use each in a meaningful way. What are the Python equivalents to each, if any?

6.6 Perform the following experiment in the Ruby REPL. Generate, by any means you can discover, two string objects that have the value `"dog"` but have different object ids. Send the `to_sym` method to each. Compare the object ids of each symbol. What does the result of this comparison tell you?

6.7 In the Ruby REPL, evaluate the expression `3 => x`. Then evaluate x. This "assignment" is really an instance of single-line pattern matching. Research Ruby's approach to pattern matching (it is much richer than this simple example would indicate), and compose illustrative scripts of your own.

6.8 Can you make a hash whose keys are numeric? Strings? Arrays? Can you make a single hash with both numeric and symbol keys? Research and describe the exact conditions under which an object can be used as a hash key.

6.9 Compare the manner in which Python and Ruby provide a "default value" when looking up values with nonexistent keys.

6.10 Python has a rather infamous issue with what are known as "default mutable arguments." Does Ruby suffer from the same issue? Write an experiment to find out.

6.11 Suppose a Ruby method is defined with a single positional parameter, for example `def f(x); x; end`. What actually happens when the method is called as one would call a named parameter, that is, `f(x:3)`?

6.12 Write a Ruby function that takes a string of the form one-or-more digits followed by an operator from the set $\{+, -, *, /\}$ followed by one or more digits, and returns the evaluation of that string. Evaluate the string using `.send`.

6.13 Read Allen Holub's *Why getter and setter methods are evil* [50]. What do you think about the claim that getters and setters are often a poor programming choice? Find some articles or papers that rebut Holub's position (if any) and summarize their main arguments.

6.14 Finding out all of the technical details of a language's scope rules requires a look at the language's published documentation, or ideally, its formal semantics if one exists. However, we can always conduct experiments. Here is a script we've written to explore some scope questions.

```
def one
  x = 1
  [1,2,3].each {|x| y = 1}
  y
end

def two
  x, y = 1, 2
  [1,2,3].each {|x| y = 1}
  y
end

def three
  x, y = 1, 2
  [1,2,3].each {|x; y| y = 1}
  y
end
```

What does this script output? What questions do you think the script author is trying to discover? What can be learned from this experiment?

6.15 How are Ruby blocks like Lua coroutines? How are they different?

6.16 The following script

```
class C; def a; end; end
class C; def b; end; end

puts C.public_instance_methods(false)
```

features the definition of a class followed by... what? Is the second line a *re*-definition? Or an extension? Run the script to find out the answer to this question. Then run the equivalent "experiment" in Python. Did you notice a difference?

6.17 Ruby has both a `Method` and an `UnboundMethod` class. What is the difference between them?

6.18 In this chapter, we did not mention Ruby's class variables and class instance variables. Find out about these features. Explain them in your own words, and show how they are different from each other. What kind of problem solutions do they make more expressive than equivalent code without them?

6.19 Contrast Ruby's use of `public`, `protected`, and `private` with that of TypeScript. You should notice some major differences with `private`.

6.20 The entity known as a "singleton class" in Ruby has been called an "eigenclass" or a "metaclass" or a "virtual class". The term "eigenclass" is not widely used, but the latter two terms are downright confusing because they have different meanings in different contexts. For example, "virtual class" is a common term in C++, and "metaclass" has a completely different meaning in the Smalltalk community. How exactly is the term used in the Smalltalk world, and how is such usage similar to, or different from, the Ruby singleton class?

6.21 Python has metaclasses, too (which we didn't cover). Compare and contrast the metaclass concept in Python with Ruby's singleton class.

6.22 Write a script that when given an object, produces the array of singleton classes, classes, and modules that are searched (in order) when resolving a message sent to the object.

6.23 Find the complete list of predefined global variables in Ruby. Experiment with them in the REPL.

6.24 One of the built-in classes in Ruby is `RubyVM`. Read the documentation for this class, and experiment with `RubyVM::InstructionSequence`. Would the use of this class be considered metaprogramming?

6.25 For each of the following classes and modules: `BasicObject`, `Object`, `Class`, `Module`, `Kernel`, `Numeric`, `Float`, and `Integer`, give its singleton class, its class, its superclass, and its list of included modules. Try your hand at a diagram of the relationship between these objects.

6.26 We did not cover Ruby's facilities for concurrent programming. Take a brief look at the Ruby documentation for information on threads, fibers, and ractors.

CHAPTER 7

Java

Java is an extremely popular enterprise programming language, designed with, among other things, thread-based concurrency and security in mind.

First appeared	1995
Designer	James Gosling
Recognized for	Classes, Threads, Mature platform
Notable uses	Enterprise computing, Android applications
Tags	Architecture-neutral, Multithreaded, High-performance
Six words or less	"Popular language for enterprise applications"

Engineers at Sun Microsystems began a project in 1991 for programming smart appliances. While the team originally considered C++ for the project's software platform, they found the language lacking in areas such as portability (implementations across machines could vary widely) and threading (as C++ had no built-in concurrency support at the time), and so a new language, Oak, was created. Sun renamed the language Java in 1994 and released it to the world the following year.

Java gained massive popularity in the early days of the World Wide Web as various browser manufacturers, notably Sun and Netscape Communications, added browser support for downloading and executing compiled Java code, known as applets. While Java did not become the most popular language for running apps in the browser (that title has gone to a similarly named—but remarkably different—language, JavaScript), Java ultimately gained huge popularity among developers, particularly for server-side (enterprise) applications, and as the initial language for application development in the Android ecosystem.

Our tour of Java begins with our three introductory programs then moves to its interesting but somewhat clunky type system, fraught with unusual inconsistencies stemming from attempts to make programs simple, efficient, and portable. We will discuss problems surrounding primitives, covariant arrays, type erasure, and the "billion dollar mistake." Java is not without its strengths, though, and we'll explore some good stuff too: records, streams, and the extensive thread-based concurrency model.

7.1 HELLO JAVA

Let's begin our exploration of Java with our familiar triangle program:

```
public class TripleApp {
    public static void main(String[] args) {
        for (var c = 1; c <= 40; c++) {
            for (var b = 1; b < c; b++) {
                for (var a = 1; a < b; a++) {
                    if (a * a + b * b == c * c) {
                        System.out.printf("%d, %d, %d\n", a, b, c);
                    }
                }
            }
        }
    }
}
```

Java programs consist *entirely* of classes, and require all code to appear within a class, even if the program is a simple script. The interpreter passes the command line arguments to a `static` function named `main` that serves as the entry point of the application.[1]

Next we have our command line permutations teaser:

```
import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public class PermutationsApp {
    public static void main(String[] args) {
        printPermutations(Arrays.asList(args), args.length - 1);
    }

    private static void printPermutations(List<String> a, int n) {
        if (n <= 0) {
            System.out.println(String.join("\t", a));
        } else {
            for (var i = 0; i < n; i++) {
                printPermutations(a, n - 1);
                Collections.swap(a, n % 2 == 0 ? 0 : i, n);
            }
            printPermutations(a, n - 1);
        }
    }
}
```

[1] At the time of publication, support for **unnamed classes and instance main methods** [95], which would allow scripts to be written without an explicit class definition and the "public static" ceremony, was a Java PREVIEW feature, meaning it was not yet a permanent feature of the language. While we've chosen to omit this feature in these printed pages, we will update the companion website to use this feature when it does become permanent.

Unlike the languages we've seen so far, the type annotations on method parameters and returns are *required*. All variables in Java are type-constrained. Local variables can have their constraints inferred (using `var` in place of an explicit constraint), provided an initializer is present. The static types of for-loop indices and anonymous function parameters can be inferred as well. The script also illustrates that Java contains *both* arrays and lists. Arrays are low-level with very few operations. The flexible list type comes with a large set of operations, including the useful `swap` operation used above.

Next, we turn to our top ten scorers teaser.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TopTenScorersApp {

    public static void main(String[] args) {

        record Player(String team, String name, int games, double ppg) {
            static Player fromLine(String[] line) {
                var points = Double.parseDouble(line[3]);
                var games = Integer.parseInt(line[2]);
                return new Player(line[0], line[1], games, points/games);
            }

            String reportLine() {
                return String.format("%-22s%-4s%8.2f", name, team, ppg);
            }
        }

        new BufferedReader(new InputStreamReader(System.in))
                .lines()
                .map(line -> line.split(","))
                .map(Player::fromLine)
                .filter(p -> p.games >= 15)
                .sorted((p1, p2) -> Double.compare(p2.ppg, p1.ppg))
                .limit(10)
                .map(Player::reportLine)
                .forEach(System.out::println);
    }
}
```

In this script, an `InputStreamReader` transforms the byte sequence from standard input into a character sequence. This reader is **decorated** by a `BufferedReader`, which buffers the characters for efficiency when reading from a file. The `lines` method produces a stream of lines, which are then split on the comma separator and used to create into player objects—instances of our custom `Player` class. The stream of player objects is filtered to keep only the players appearing in 15 or more games, then sorted descending on points-per-game and the top 10 are kept. Another mapping operation transforms the player into a stream of strings with the player descriptions in their desired final form. Each string is then written to standard output with the famous `println` method of standard output.

This script makes use of classes from Java's standard library (e.g. `BufferedReader` and `InputStreamReader` from the package `java.io`. Unlike Python, Java does not *require* `import` statements; importing a class simply lets us avoid specifying the full **qualified names** of these classes throughout our code.[2] The class `String` comes from the special package `java.lang`, whose classes are imported by default.

7.2 THE BASICS

Java's static types are partitioned into **primitive types** and **reference types**. Instances of the latter are called **objects**. Variables store values of primitive types directly and store references to objects. The eight primitive types are: `byte`, `short`, `int`, and `long` (signed integers of 8, 16, 32, and 64 bits respectively); `float` and `double`, (32 and 64-bit IEEE 754 floating-point numbers respectively); `char`, unfortunately a UTF-16 **code unit**, and `boolean`. All other types, including arrays and strings, are reference types.

The `==` operator on references *always* returns whether its operands refer to the same object. If you desire a custom equality comparison, the convention is for you to override the `equals` method, whose default implementation in the ultimate superclass `Object` behaves as `==`. You will often see this method implement a field-by-field comparison, as in for example, the class `LocalDate` from the standard library:

```
import java.time.LocalDate;

public class EqualsExample {
    public static void main(String[] args) {
        var d1 = LocalDate.of(2020, 2, 29);
        var d2 = LocalDate.of(2020, 2, 29);
        assert d1 != d2; // These are two distinct objects
        assert d1.equals(d2); // But each of the fields are equal
    }
}
```

All variables—local variables, parameters, and object fields—are type-constrained. The type constraints of parameters and object fields must be supplied explicitly; those of local variables may be inferred in certain situations:

```
public class VariableExample {
    public static void main(String[] args) {
        String mood = "happy"; // ok, but redundant
        var message = "WAIT"; // sensibly inferred
        var dozen = 12; // inferred as int
        // dozen = 12.0; // ERROR cannot assign float to int
        // dozen = "dog"; // ERROR cannot assign String to int
        boolean found; // ok to omit initializer if constraint present
        // var status; // ERROR: impossible to infer
    }
}
```

[2] Without the `import` statement, we must write `java.io.BufferedReader` instead of simply `BufferedReader`.

Shockingly, the value `null` belongs not to its own static type, but to *every reference type*. This unfortunate design choice has quite a history in programming language design. Its inventor, Sir Charles Anthony Richard Hoare, refers to it as his **billion dollar mistake** [49], a rough estimate of the cost of bugs resulting from this decision. The mistake has both philosophical and practical ramifications. Philosophically, a variable constrained to type `String` should be bound to strings and only to strings; allowing `null`, which represents the *absence* of a string, to *be* a string, is misleading.[3] As a compiler cannot detect *all* accidental uses of `null` before the program is run, the programmer must deal with these cases at run-time:

```
public class BillionDollarMistakeExample {
    public static void main(String[] args) {
        try {
            String message = null; // WHY IS THIS ALLOWED?
            System.out.println(message.toLowerCase());
            assert false; // Execution will never get here
        } catch (NullPointerException e) {
            System.out.println("Surprise!! Run-type checked!!");
        }
    }
}
```

But don't be too hard on Sir Tony; his accomplishments include Quicksort, Communicating Sequential Processes (a highly influential process algebra), and Hoare logic (used in proving the correctness of programs with assignment statements).

Though Java is mostly strongly typed, it will implicitly perform any type conversion that is considered to be a **widening conversion**. With primitive types, a widening conversion occurs when the target type has a wider range of values (a smaller minimum value and a larger maximum value) than the source type. With reference types, a widening conversion occurs when the type being converted to is a superclass of the original type (e.g. from `String` to `Object`, the ultimate ancestor of all classes). A **narrowing conversion** is the opposite of a widening conversion, and is never performed implicitly.

7.3 CLASSES

As mentioned above, Java values are primitives or objects. Every object is an instance of a class. Java classes can be introduced with the `class`, `record`, `enum`, or `interface` keyword. Each compilation unit in Java defines one or more classes, optionally as part of a **package**. If no package declaration appears, the classes will land in a **default package**.[4] Packages, are bundled into modules. If no module is unspecified, a package is part of an unnamed module.

Classes can contain fields, methods, constructors, static and instance initializers, and other (nested) classes, of which only the first three are common and will be covered in this chapter.

[3] TypeScript, for example, gets it right: the type `string` contains only strings; one must use `string|null` for "optional" strings. The static systems of Python, Kotlin, Rust, Swift, and others also properly avoid conflating nulls with other types unless *explicitly* added (via a union) by the programmer.

[4] Java best practices dictate all code should go in a named package, but examples in this chapter will use the default package for brevity.

We've seen classes above that host a command line program; such classes do not have instances. Our first example of an instantiable class will be a simple, stateful, arithmetic sequence generator:

```
public class ArithmeticSequence {
    private int delta;
    private int current;

    public ArithmeticSequence(int start, int delta) {
        this.current = start;
        this.delta = delta;
    }

    public int next() {
        int toReturn = current;
        current += delta;
        return toReturn;
    }
}
```

Instances of classes are created with the `new` operator:

```
public class ArithmeticSequenceExample {
    public static void main(String[] args) {
        var g1 = new ArithmeticSequence(10, 3);
        assert g1.next() == 10;
        assert g1.next() == 13;
        assert g1.next() == 16;
        var g2 = new ArithmeticSequence(21, -2);
        assert g2.next() == 21;
        assert g2.next() == 19;
        assert g2.next() == 17;
    }
}
```

Because Java stores, assigns, and compares only *references* to objects, two distinct sequence objects—even with the same start and delta values—are never equal to each other. In practice, this is fine; mutable objects are rarely compared for equality!

But for many small immutable objects, value-based equality may indeed be desired. For the first twenty years of Java's existence, programmers had to manually craft immutability through private fields, build value-based equality by overriding the `equals` method from the implicit superclass `Object`, and ensure equal objects would be treated as nondistinct inside sets by overriding the method `hashCode`.[5] For good measure, they would also override `toString`, a method invoked any time an object needs to be printed. Such classes took the following form:

[5] Why `hashCode`? Perhaps surprisingly, a programmer can override the `equals` method to make objects A and B equal to each other but this will *not* prevent both A and B from appearing in the same set. Set inclusion is determined by an object's hash code. Java programers learn to override `hashCode` whenever they override `equals`.

```java
// Obsolete Java code. Superseded by records.
import java.util.Objects;

public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {
        return x;
    }

    public double y() {
        return y;
    }

    @Override
    public boolean equals(Object o) {
        return o instanceof Point other && x == other.x && y == other.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

More recently, the `record` construct was added to Java to denote classes whose objects are immutable with proper value-based semantics. The following code produces *exactly the same class as the one above*:

```java
public record Point(double x, double y) {
}
```

Classes exist in subclass-superclass relationships, with an ultimate ancestor class called `Object`. Subclasses are introduced with the `extends` keyword, and can add new fields and methods as well as override nonfinal methods from the superclass. A Java class may be marked `abstract` to prevent it from having any instances, `final` to prevent subclassing, or `sealed` to permit only a fixed set of subclasses. Our recurring animals example illustrates abstract classes and subclassing:

```java
abstract class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    abstract String sound();
    public String speak() {
        return name + " says " + sound();
    }
}
```

```java
class Horse extends Animal {
    public Horse(String name) {
        super(name);
    }
    @Override String sound() {
        return "neigh";
    }
}
```

```java
class Cow extends Animal {
    public Cow(String name) {
        super(name);
    }
    @Override String sound() {
        return "moooo";
    }
}
```

```java
class Sheep extends Animal {
    public Sheep(String name) {
        super(name);
    }
    @Override String sound() {
        return "baaaa";
    }
}
```

```java
public class AnimalsApp {
    public static void main(String[] args) {
        Animal s = new Horse("CJ");
        assert s.speak().equals("CJ says neigh");
        Animal c = new Cow("Bessie");
        assert c.speak().equals("Bessie says moooo");
        assert(new Sheep("Little Lamb").speak().equals(
            "Little Lamb says baaaa"));
    }
}
```

Inheritance applies to fields and methods only. Subclasses do not inherit constructors from superclasses; they can, however, refer to the superclass constructor within their own constructors via the `super` keyword. A superclass expecting subclasses to implement a particular method must explicitly declare that method an **abstract method**. A class extending an abstract class must either implement all abstract methods of its superclass or declare itself abstract. In none of the previous chapters was such a declaration necessary. Were we to call the `speak` method on a subclass in JavaScript, Python, or Ruby that did not implement the `sound` method, a *run-time error* would occur. In Java, such a subclass could not even exist—attempting to create a concrete subclass without overriding an abstract method is a *compile-time error*.

7.4 INTERFACES

Java's **interfaces** define a set of related constants and methods providing a contract that implementing classes must conform to. An interface acts much like an abstract class; it defines a new type, it can contain abstract methods, and it cannot be instantiated. However, abstract classes can define fields for instances of concrete subclasses; interfaces can not. Interface fields are always public, static, and final (whether these modifiers appear or not!). Interface methods are private if marked `private` and public otherwise. Methods not marked `default` or `static` are abstract. The following interface defines two public abstract methods and a default method:

```
interface Drawable {
    void outline(Color color);
    void fill(Color color);
    default void render(Color outlineColor, Color fillColor) {
        fill(fillColor);
        outline(outlineColor);
    }
}
```

Java allows classes to implement any number of interfaces, so the language must define the semantics of implementing from two interfaces whose fields and methods may conflict—the **multiple inheritance** scenario. Are both interface fields inherited or only one? Are both default interface methods inherited or only one? If one, which one? Is there a method resolution order, as we saw in Python? Let's explore:

```
interface A {
    int X = 1;            // implicitly public static final
    default void f() {}   // implicitly public
}

interface B {
    int X = 2;            // implicitly public static final
    default void f() {}   // implicitly public
}

// class C implements A, B {
//     // WILL NOT COMPILE: unrelated defaults for f()
```

```
//     public static void main(String[] args) {
//         // System.out.println(X); // WILL NOT COMPILE (X ambiguous)
//     }
// }

class D implements A, B {
    @Override
    public void f() {
        System.out.println(A.X + B.X);
    }
}
```

Unimplemented clashing defaults result in the compile time error "class C inherits unrelated defaults for f() from types A and B" so the implementing class is expected to provide a concrete override. Clashing fields cannot be used without qualifying with the name of the interface. This is reasonable, as interface fields can be thought of as global constants with the interface comprising part of its name.

An interface with exactly one abstract method is called a **functional interface**. Java provides nice syntactic sugar for instances of classes that implement a functional interface: you can define instances directly without ever specifying the implementing class!

```
@FunctionalInterface
interface DoubleToDouble {
    double apply(double x);
}

public class TwiceExample {
    private static double twice(DoubleToDouble f, double x) {
        return f.apply(f.apply(x));
    }
    public static void main(String[] args) {
        assert twice(x -> x + 2, 10) == 14;
        assert twice(Math::floor, -11.77) == -12;
    }
}
```

We've marked our `Function` interface with the `@FunctionalInterface` annotation; though optional, it's considered good practice. The sole abstract method within the interface denotes a function with a single `double` parameter and a `double` result. The expression `x->x*2`, known in Java as a **lambda expression**, conforms to this type specification, and thus makes an acceptable first argument to our `twice` method. The same holds for `Math::floor`, a function object known as a **method reference**. Method references sugar lambda expressions, as summarized in Table 7.1.

Operation	Example	Long Form
Static Method	`Math::floor`	`x -> Math.floor(x)`
Instance Method	`Dog::bark`	`d -> d.bark()`
Constructor	`Part::new`	`p -> new Part(p)`

Table 7.1 Java method feferences

7.5 GENERICS

As we saw earlier with TypeScript, Java allows type parameters in classes and methods, making possible types such as "list of integers" and "list of strings" to be generated from a single parameterized type definition. Here is a parameterized type `Pair`, and two concrete types instantiated from it:

```
record Pair<T, U> (T first, U second) {
}

class PairExample {
    public static void main(String[] args) {
        var p1 = new Pair<String, Integer>("Hello", 2);
        var p2 = new Pair<Integer, Boolean>(1, false);
        assert p1.second() == 2;
        assert p2.second() == false;
    }
}
```

Interestingly, Java does not allow primitive types to be type parameters. Therefore, there exist eight classes that "wrap" the eight primitive types, namely `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`, Fortunately, as this example shows, Java automatically converts a primitive to (**autoboxing**) and from (**unboxing**) an instance of its corresponding wrapper class.

Parameterized types appear frequently in Java's standard library:

```
import java.util.List;
import java.util.Set;
import java.util.HashSet;

public class ListAndSetExample {
    public static void main(String[] args) {
        var sizes = new HashSet<Integer>();
        for (var word : List.of("do", "while", "if", "a")) {
            sizes.add(word.length());
        }
        assert sizes.equals(Set.of(5, 2, 1));
    }
}
```

Both `List<E>` and `Set<E>` are interfaces in the package `java.util`. `List` and `Set` are called **parameterized types** or **generic types** with E as a **type parameter**. We've initialized the local variable `sizes` with an object of class `HashSet<Integer>`—one of several kinds of set classes in Java.[6] The variable `words` is initialized to an object of *some* class implementing the list interface; we don't exactly know, nor care, which class this is—we access the object through the methods defined by the interface type with which we have constrained the variable.

[6] `List<E>` and `Set<E>` are actually interfaces; implementing classes include `LinkedList<E>`, `ArrayList<E>`, `HashSet<E>`, `TreeSet<E>`, `EnumSet<E>`, and `ConcurrentSkipListSet<E>`. There are dozens of additional collection interfaces and classes the standard library. For a complete list, see [82].

Another generic type in Java's standard library is `Optional<T>`. Optionals help mitigate the previously mentioned "billion dollar mistake," manifested in Java as the inclusion of `null` in every reference type. For instance, while `s.length()` is clearly a type error when *s* is `null`, the value `null` belongs to the *static type* String, and the type error is only detected at run time. This feels strange because Java's static type system is so rich that many programmers forget to make necessary type checks at run time.

Nulls exist because certain information is not always required, but *optional*. Consider modeling a person with a required name but an optional boss. Type-constraining the name and boss fields as `String` and `Person` respectively forces us to check the name field for `null` on construction, but allow the boss to be `null` if desired. But expressions such as `p.boss().name()` can still fail at run-time. But there is good news: the compiler *can* type check expressions if we bring in the generic type `Optional<T>` from the package `java.util`. Think of an optional as a wrapper: an optional person either wraps a person or is empty (see Figure 7.1).

Figure 7.1 Optionals

```
import java.util.Optional;
import java.util.Objects;

class Person {
    private String name;
    private Optional<Person> boss;

    private Person(String name, Optional<Person> boss) {
        this.name = Objects.requireNonNull(name);
        this.boss = boss;
    }

    public Person(String name) {
        this(name, Optional.empty());
    }

    public Person(String name, Person boss) {
        this(name, Optional.of(boss));
    }

    public String name() {return name;}
    public Optional<Person> boss() {return boss;}
}
```

The field and constructor definitions express our intent quite well: name is required, boss is optional. The static factory methods `Objects.requireNonNull` and `Optional.of` prevent objects being constructed with unwanted nulls. And the expression `p.boss().name()` fails to compile: you'll still have to deal with the no-boss case, but you won't be surprised by it. The compiler forces you to deal with it.

The most common usage patterns for optionals are as follows. Let o be an optional, and x its wrapped value, if present. Then:

- o`.isPresent()` returns whether o wraps a value.
- o`.isEmpty()` returns whether o does not wrap a value.
- o`.ifPresent(`f`)` invokes $f(x)$ if present and does nothing otherwise.
- o`.ifPresentOrElse(`f, g`)` invokes $f(x)$ if present and $g()$ otherwise.
- o`.orElse(`y`)` returns x if present and y otherwise.
- o`.orElseGet(`g`)` returns x if present and $g()$ otherwise.
- o`.orElseThrow()` returns x if present and throws a `NoSuchElementException` otherwise.
- o`.filter(`p`)` returns o if present and $p(x)$ true; returns an empty optional otherwise.
- o`.map(`f`)` returns $f(x)$ wrapped if present, and an empty optional otherwise.
- o`.flatMap(`f`)` returns $f(x)$ (unwrapped) if present, and an empty optional otherwise.

Let's get a feel for these methods with a small example:

```
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        var alice = new Person("Alice");
        var bob = new Person("Bob", alice);

        bob.boss().ifPresent(p -> {
            assert p == alice;
        });
        alice.boss().ifPresent(p -> {
            assert false;
        });
        assert alice.boss().orElse(bob) == bob;
        assert bob.boss().orElse(bob) == alice;

        Optional<Person> b = bob.boss();
        assert b.filter(p -> p.name().startsWith("A")) == b;
        assert !b.filter(p -> p.name().startsWith("B")).isPresent();
        assert b.map(Person::name).orElse("").equals("Alice");
        assert alice.boss().map(Person::name).orElse("").equals("");
    }
}
```

Generic types may be used to constrain parameters or return types, using **wildcards**. A method to compute the length of a list would have signature `int length(List<?> a)`, permitting arguments of type list of integer, list of string, etc. It is even possible to constrain the *element types themselves*. here we illustrate a method that accepts lists of animals, lists of cows, lists of sheep, and lists of horses only (that is, any list whose element type is `Animal` or any descendant of `Animal`):

```
import java.util.List;
import java.util.Arrays;

public class AnimalChorusExample {
    public static void chorus(List<? extends Animal> animals) {
        for (Animal animal : animals) {
            System.out.println(animal.speak());
        }
    }

    public static void main(String[] args) {
        List<Cow> cows = Arrays.asList(new Cow("Bessie"));
        List<Sheep> sheep = Arrays.asList(new Sheep("Wooly"));
        chorus(cows);
        chorus(sheep);
    }
}
```

The `extends` keyword signals a **upper bounded wildcard**, meaning the type parameter can be instantiated only with a type argument of class `Animal` or any Animal subclass (direct or indirect). **Lower bounded wildcards**, on the other hand, admit superclasses. For example, the type constraint `Set<? super Sheep>` can be satisfied with objects of type `Set<Sheep>`, `Set<Animal>`, or even `Set<Object>`. Bounded wildcards appear frequently in generic function types, with lower bounds for return types and upper bounds for parameter types.

Wildcards cannot be used to instantiate objects; however, **bounded type parameters** exist for this purpose. Bounded type parameters can only have upper bounds. In the following class, we require two fields to share the *same* type, yet be of a class that implements the interface `Comparable`.

```
public class MinMax<C extends Comparable> {
    private C min;
    private C max;

    public MinMax(C x, C y) {
        min = x.compareTo(y) < 0 ? x : y;
        max = x.compareTo(y) > 0 ? x : y;
    }

    public C min() {return min;}
    public C max() {return max;}
}
```

Neither a wildcard nor simply using `Comparable` can express this constraint.

One final note: Generics are part of Java's *static* type system, *not* its dynamic system. While run-time expressions such as `x instanceof List` do work, the type *parameters* do not exist at run time. In other words, `List<Integer>` is a static type only, while `List` is both a static and dynamic type. You'll sometimes hear this situation called **type erasure**.

In this section, we've barely scratched the surface of the world of Java generics. We've not had space to cover **unbounded wildcards**, reification, or wildcard capture. To continue exploring this rich area of the Java language, visit the Generics Trail in the Java Tutorial [81] or Angelika Langer's excellent Generics FAQ [64].

7.6 TYPE VARIANCE

In the previous section, we showed that Java used `List<? extends Animal>` to accept lists of sheep or lists of cows, etc. Why not just `List<Animal>`? There's a good reason. If the latter were acceptable, type errors could occur. Given:

```
List<Horse> horses = List.of(new Horse());
List <Cow> cows = List.of(new Cow());
List <Animal> animals;
```

the assignment `animals = horses;` is illegal. Why? If it were legal, then the (obviously statically correct) call `animals.add(new Cow())` would add a cow to a list of horses. Prohibiting the assignment of a list-of-horses to a list-of-animals, i.e., stating that `List<Horse>` *is not* subtype of `List<Animal>`, prevents this run-time type error. Formally:

- If `List<Horse>` were a subtype of `List<Animal>`, lists would be **covariant**
- If `List<Animal>` were a subtype of `List<Horse>`, lists would be **contravariant**
- If neither holds, lists would be **invariant**
- If both hold, lists would be **bivariant**

Lists, then, are *invariant*, as Java aspires to do mostly static type checking. However, in opposition to this aspiration, Java arrays are *covariant*! Run-time type checks, then, are often required for array elements:

```
public class CovariantArrayExample {
    public static void main(String[] args) {
        Animal[] animals = new Sheep[10];       // Covariance, sadly
        try {
            animals[0] = new Cow("Bessie");
        } catch (ArrayStoreException e) {
            System.out.println("Runtime type checking!");
        }
    }
}
```

You may recall from Chapter 4 that TypeScript has covariant arrays as well. This implies that static type systems, like those of TypeScript and Java, do not catch *all* type errors at compile time, just most of them. Programmers need to be aware of which type errors must be detected at run time. For Java, these include nulls and covariant arrays.

7.7 STREAMS

Like most modern programming languages, Java provides expressive ways to manipulate sequential data. A **stream** in Java provides access to an underlying data *source*, such an array, list, generator function, or I/O channel. The **intermediate operations** on streams transform the data, and the **terminal operation** at the end of stream's execution assembles a result.

Streams can be created in a variety of ways, including but not limited to:

```
Stream.of("dog", "rat", "bat", "cat")
Arrays.stream(a)
myList.stream()
IntStream.range(1, 100)          // 1 through 99
"abc123".codePoints()
Files.lines(fileName)            // lines of a file given the filename
bufferedReader.lines()           // see java.io.BufferedReader docs
Files.list(folderName)
pattern.splitAsStream()
Stream.iterate(8, i -> i * 10)   // infinite! 8, 80, 800, 8000, ...
Stream.generate(Math::random)    // infinite! random vals from [0.0,1.0)
new Random().ints()              // infinite! random ints
```

Once built, streams can be transformed by mapping, filtering, and sorting. You can take only the first few items that meet a condition or skip the first few that meet a condition. You can even specify the exact number of elements to take or skip—limiting the stream size is a must if you created an infinite stream! To "complete" stream processing, you can create an array from the elements, apply a reducer function, or simply report its minimum, maximum, count, or the first element matching a desired condition. Many more possibilities exist. To give you a feel for what's available, Table 7.2 lists just a handful of intermediate and terminal operations. See the Java documentation for details.

Intermediate	Terminal
`filter(`*predicate*`)`	`forEach(`*consumer*`)`
`map(`*fun*`)`	`toArray()`
`flatMap(`*fun*`)`	`reduce(`*accumulator*`)`
`takeWhile(`*predicate*`)`	`collect(`*collector*`)`
`dropWhile(`*predicate*`)`	`min(`*comparator*`)`
`distinct()`	`max(`*comparator*`)`
`sorted()`	`count()`
`sorted(`*comparator*`)`	`anyMatch(`*predicate*`)`
`limit(`*n*`)`	`allMatch(`*predicate*`)`
`skip(`*n*`)`	`findFirst()`
`peek(`*consumer*`)`	`findAny()`

Table 7.2 A few Java stream operations

Of the various ways to terminate a stream, the `collect` method is the most flexible. This method accepts a collector object, many types of which are available in the standard library. Collectors gather stream elements into a list (`toList`) or set (`toSet`) or map (`toMap`), form groups or partitions, compute averages, and join elements into a string. As a small but practical example of using streams, here is a Java command line application that reads text from standard input, breaks it up into words (that contain letters and apostrophes), lowercases of each word, counts them, and reports to standard output each word with its frequency, in alphabetical order (since `TreeMap`s are automatically sorted by key):

```
import java.util.TreeMap;
import java.util.regex.Pattern;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.function.Function;
import java.util.stream.Collectors;

public class WordCountApp {
    public static void main(String[] args) {
        Pattern nonWord = Pattern.compile("[^a-z']+");
        new BufferedReader(new InputStreamReader(System.in))
            .lines()
            .flatMap(line ->
                nonWord.splitAsStream(line.toLowerCase()))
            .filter(word -> !word.isEmpty())
            .collect(Collectors.groupingBy(Function.identity(),
                TreeMap::new, Collectors.counting()))
            .forEach((word, count) ->
                System.out.printf("%s %d%n", word, count));
    }
}
```

Two concepts are helpful to keep in mind when learning Java streams. First, streams don't own any data—the data is held in an underlying collection object or is generated on demand. Second, the intermediate operations don't happen "right away." These operations are lazy: they set up operations that occur when the terminal operation is invoked. This allows the implementation to prepare an efficient execution plan, perhaps performing operations in parallel if possible.

7.8 THREADS

Concurrency refers to the modeling and coordination of independent, distinct computing activities whose execution spans may overlap in time [90]. When we design a system to be concurrent, we focus on these independent activities and the interactions between them. One classic example is a word processor: in a concurrent design, we define separate agents for input device listening, spell checking, hyphenation, line justification, page breaking, and so on, rather than define a monolithic function that scatters bits of each activity in different regions of the code. Note that while the lifetime of the activities overlap, it is not required that concurrent activities truly execute in **parallel**, i.e., at the same time on separate physical *execution units*—processors or cores.

Programs run on one or more "lines of execution," each of which performs operations sequentially. These lines are known as **threads** or sometimes **threads of control**. As threads have their own independent execution lines, they provide a natural way to express concurrent activities. Coroutines, which we encountered in previous chapters, also express concurrency well. A set of coroutines generally will run within a single thread as they explicitly yield control to each other. Threads don't have to yield—in fact, they may be **preempted** by a scheduler when not enough physical execution units exist for the available threads.[7]

Let's look at how to use threads in Java. We'll start 100 threads t_i, for $i \in 1..100$, each of which will determine whether i is prime, and print the value if so:

```
public class PrimePrinterApp {
    public static void main(String[] args) {
        for (var i = 1; i <= 100; i++) {
            final var candidate = i;
            new Thread(() -> {
                for (var divisor = 2; divisor <= candidate; divisor++) {
                    if (divisor == candidate) {
                        synchronized (System.out) {
                            System.out.print(" " + candidate);
                        }
                    } else if (candidate % divisor == 0) {
                        break;
                    }
                }
            }).start();
        }
    }
}
```

The **main thread** enters a loop creating 100 instances of the class `java.lang.Thread`. We pass an instance of the functional interface `Runnable` to the thread's constructor. Calling `start` on the thread invokes its runnable on a new thread.[8] Because each thread writes to a **shared resource**—in this case `System.out`—we use a `synchronized` statement to ensure only one thread can call its methods at a time, thus preventing multiple threads from interleaving their outputs.

Threading naturally introduces **nondeterminacy** into an application. One run of the script above produced: `3 7 5 2 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 83 79 89 97`; another produced `2 71 73 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 79 83 89 97`; still others gave slightly different orderings.

A Java thread t has a threadId, a name, a priority, a state (new, runnable, blocked, waiting, timed waiting, or terminated), and several status fields (alive, daemon, interrupted). You can wait indefinitely for a thread to finish (via t`.join()`) or for a maximum period of time (t`.join(`*millis*`,`*nanos*`)`). A Java thread can be either a (1) **platform thread**, wrapping

[7] Whether or not threads may be forcibly preempted at any time is up to an implementation. A scheduler may bump a thread from an execution unit only at specific points, such as when the thread (voluntarily) waits for an event, a resource to become available, or a timer to expire.

[8] In our example, the runnable is a closure. Java has a rule that closed-over variables must be `final`, so we had to introduce a new variable, `candidate` instead of using `i` directly.

a real operating system thread, or (2) a **virtual thread**. Operating systems threads are rather heavy and somewhat expensive to use, and are limited in number: creating too many of these will exhaust system resources rather quickly. Use platform threads for complex or computationally-intensive ("CPU Bound") tasks for which only a few are needed. Virtual threads, on the other hand, are managed entirely within the Java runtime and assigned to, and unassigned from, actual platform threads (called **carrier threads**) at runtime. Many thousands of virtual threads can comfortably run in a single program. Use virtual threads these for small-ish tasks that spend most of their time waiting for file operations, network operations, and timers (so-called "I/O Bound" activities).

As an illustration of virtual threads, the following application makes a number of concurrent network requests to the free and public Poke API [41] to fetch data for a collection of Pokemons. Each thread makes a single network call and prints a portion of the data that it receives. The threads are launched by an **executor service**, which, because it was initialized in a try-block, will wait for all threads to finish and clean up resources at the end of the block. Every HTTP request is a synchronous call: there are no callbacks, no `await` keyword, no promises or futures being returned. Of course, the calls do not pause the entire application: the runtime *knows* the network request is a blocking call and thus preempts the thread, allowing others to run.

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.concurrent.Executors;
import org.json.JSONObject;

public class PokeApiApp {
    static String baseUrl = "https://pokeapi.co/api/v2/pokemon/";
    static HttpClient httpClient = HttpClient.newHttpClient();
    static String[] pokemonNames = {
            "ditto", "pikachu", "mew", "weedle", "eevee" };

    public static void main(String[] args) {
        try (var service = Executors.newVirtualThreadPerTaskExecutor()) {
            for (var name : pokemonNames) {
                service.submit(() -> {
                    fetchPokemon(name);
                });
            }
        }
    }

    private static void fetchPokemon(String name) {
        var url = URI.create(baseUrl + name);
        var request = HttpRequest.newBuilder(url).build();
        try {
            var response = httpClient.send(
                    request, HttpResponse.BodyHandlers.ofString());
```

```
            if (response.statusCode() >= 400) {
                throw new RuntimeException(
                        "HTTP status code " + response.statusCode());
            }
            var jsonObject = new JSONObject(response.body());
            synchronized (System.out) {
                System.out.printf(
                        "Name: %s, Height: %d, Weight: %d%n",
                        jsonObject.getString("name"),
                        jsonObject.getInt("height"),
                        jsonObject.getInt("weight"));
            }
        } catch (Exception e) {
            System.err.println("Failed to fetch " + name + ": "
                    + e.getLocalizedMessage());
        }
    }
}
```

Writing multithreaded code operating on shared, mutable data brings the possibility of starvation, livelock, deadlock, memory consistency errors, and race conditions. This book is not a treatise on programming with threads, so we won't attempt to define these terms; instead, we simply wish to highlight some of the language and library support for concurrency in Java. We'll outline the important concepts, then finish this chapter with a comprehensive example:

- **Monitors**. Every Java object has its own intrinsic lock, known as a **monitor**. A thread executing the statement `synchronized(o) {...}` acquires the lock for *o* if not already acquired, or blocks indefinitely until a different thread holding the lock releases it.
- **Condition synchronization**. Sometimes you need to execute code of the form "if this condition holds, acquire a lock." However, the condition might change from true to false between the test and the lock acquisition, causing enormous trouble. Java defines the very special `wait`, `notifyAll`, and `notify` methods on every object to allow programming styles to safely, and efficiently, acquire resources conditionally.
- **Executor services**. Executor services manage the creation, execution, and termination of **tasks**. Tasks are objects implementing one of the functional interfaces `Callable` (value-returning) or `Runnable` (non-value-returning). Use a service to either *submit* a single task (asynchronously) or *invoke* a collection of tasks, blocking until all (`invokeAll`) or one (`invokeAny`) of the tasks has completed, or, optionally, until a timer has expired. Submission and invocation both return `Future` objects used to check the pending status of a task, or to wait on the task for completion. Executor services are auto-closable, so you will generally use them in a try-block. You can, however, perform a `shutdown` or `shudownNow` operation to prevent it from accepting new tasks and have its associated resources cleaned up.
- **Thread Pools**. A common type of executor service is a **thread pool**, which allows you to multiplex a large number of complex tasks onto a smaller number of platform threads. Virtual threads should not use a thread pool; they are automatically efficiently managed.

- **Explicit locks and synchronization objects**. The standard library provides a number of different objects for handling various synchronization use cases, including:
  - Locks, to coordinate reading and writing;
  - Count down latches, allowing a number of threads to wait until a counter goes down to zero;
  - Cyclic barriers, used when a fixed number of threads need to meet (block) at some point, then all proceed after the last thread arrives.
  - Phasers, barriers with a variable number of threads that can be waiting;
  - Exchangers, defining a point in which two threads can exchange objects; and
  - Semaphores, allowing a maximum number of threads to use a resource at a time.
- **Atomics**. Objects that take advantage of hardware instructions to perform ultra-fast lock-free atomic read and update operations.
- **Concurrent data structures**. Java's `BlockingQueue` and `ConcurrentMap` implementations simplify writing some concurrent applications. In a blocking queue, for example, use `put` to block until the queue has free space before inserting, and `offer` to attempt an insertion either immediately or within a given duration (returning `true` on insertion and `false` if capacity was not available). When removing, `take` blocks until the queue is nonempty, and `poll` expects removal to be possible immediately or within a time frame, returning whether an element was able to be removed.

The classic examples of concurrent programming include Readers and Writers, the Producer-Consumer Bounded Buffer, and Dijkstra's *Dining Philisophers* [48]. We'll break with tradition and build a little simulation, with food as in Dijkstra's problem, but without shared utensils.

Fifteen customers spend their time placing orders with three cooks. The orders are placed in a queue holding only five orders at a time. A customer will wait up to 7 seconds to get an order into the queue, going home for a bit if waiting too long. The cooks prepare the order then deliver the finished meal to the customer, blocking until the customer takes the meal. The simulation runs until each customer has eaten 10 meals.

Our simulation runner provides an abstract base class for our active threads with a couple methods to allow reporting to standard output: `log` to log a message, and `act` which logs a message before sleeping for a random number of milliseconds (bounded above) to simulate an action (such as cooking or eating). The runner's thread—the application's main thread—creates 18 new threads to simulate the cooks and customers.

```
public class Restaurant {

    static class Agent extends Thread {
        Agent(String name) {
            super(name);
        }

        void log(String action) {
            synchronized (System.out) {
                System.out.println(getName() + " " + action);
```

```
            }
        }

        void act(String action, int time) throws InterruptedException {
            log(action);
            Thread.sleep((int) (Math.random() * time / 2 + time / 2));
        }
    }

    public static void main(String[] args) {
        for (var i = 0; i < 3; i++) {
            new Cook("Cook-" + i).start();
        }
        for (var i = 0; i < 15; i++) {
            new Customer("Customer-" + i).start();
        }
    }
}
```

A Java application finishes when all **nondaemon** threads have finished; **daemon** threads, therefore, are those threads intended to run indefinitely, performing services as needed. Cooks are naturally daemon threads, grabbing orders, cooking, and serving customers. As we'll see in a minute, fetching an order and serving the customer are **synchronous** operations; cooks will block until an order can be fetched, and also block until the customer accepts the handoff of the cooked order.

```
class Cook extends Restaurant.Agent {

    Cook(String name) {
        super(name);
        this.setDaemon(true);
    }

    @Override public void run() {
        try {
            while (true) {
                var order = Order.begin();
                act("cooking " + order, 12000);
                order.serve();
            }
        } catch (InterruptedException e) {
            log("has quit");
            Thread.currentThread().interrupt();
        }
    }
}
```

All blocking operations in Java may throw an `InterruptedException`; this exception must be (1) handled in any method that may throw it, or (2) declared to be throwable. In our simulation, interrupting a cook terminates (pun intended) the employee (thread).

Customers eat ten meals then go home. They have to order each meal. If the order is placed within 7 seconds, the customer will wait for it to be cooked, accept the meal from the chef, then eat. Otherwise, the impatient goes shopping and returns later. Interrupting a customer causes banishment from the restaurant.

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;

public class Customer extends Restaurant.Agent {
    private BlockingQueue<Order> meal = new SynchronousQueue<>();

    Customer(String name) {
        super(name);
    }

    void serve(Order order) throws InterruptedException {
        meal.put(order);
    }

    @Override
    public void run() {
        try {
            var mealsEaten = 0;
            while (mealsEaten < 10) {
                if (Order.place(this, "pancakes", 7000)) {
                    var order = meal.take();
                    act("eating " + order, 10000);
                    mealsEaten++;
                } else {
                    act("can't place order, may return later", 5000);
                }
            }
            log("going home");
        } catch (InterruptedException e) {
            log("has stopped coming to this restaurant");
            Thread.currentThread().interrupt();
        }
    }
}
```

The handoff of the cooked meal between the cook and the customer employs a **synchronous queue**. A synchronous queue has no capacity: its "head" is the object the cook is inserting; items are thus "removed" only when an attempted insert is in progress. The methods `put` and `take` both block indefinitely, a less-than-ideal situation were one party to disappear while the other was waiting.

Finally we come to the order class. We maintain a single queue to hold all (uncooked) order objects. We've made the constructor private so that customers may create orders only via the static method `place`, which may time out. Cooks grab an order from the queue in a blocking fashion (via `take`), and when finished cooking, take credit for the work by updating the order object and delivering it personally to the customer. We've chosen the size

of this queue to be small, relative to the number of customers, so in running the simulation, you'll be likely to see customers getting frustrated (threads will time out) and leaving the restaurant temporarily.

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;

public class Order {
    private static BlockingQueue<Order> orders =
            new ArrayBlockingQueue<>(5);

    private String content;
    private Customer customer;
    private Cook cook;

    private Order(Customer customer, String content) {
        this.customer = customer;
        this.content = content;
    }

    static boolean place(Customer customer, String content, int maxWait)
            throws InterruptedException {
        var order = new Order(customer, content);
        return orders.offer(order, maxWait, TimeUnit.MILLISECONDS);
    }

    static Order begin() throws InterruptedException {
        return orders.take();
    }

    void serve() throws InterruptedException {
        cook = (Cook)Thread.currentThread();
        customer.serve(this);
    }

    @Override public String toString() {
        return content + (cook == null ? " for " + customer.getName()
                : " cooked by " + cook.getName());
    }
}
```

7.9 METAPROGRAMMING

We'll finish our tour of Java with a very brief stop at its metaprogramming facilities. We've seen Ruby metaprogramming in the previous chapters, so the basic ideas by now should be familiar. We would like to treat language constructs *of* our program as objects *in* the program, for both introspection and code generation. On the introspective side, Java provides

ample means of inspecting objects and classes. Using our animal classes from earlier in the chapter:

```
public class IntrospectionExample {
    public static void main(String[] args) throws Exception {

        // Check types at runtime
        Animal bessie = new Cow("Bessie");
        assert bessie instanceof Cow;
        assert bessie.getClass() == Cow.class;

        // Get a class object via its name!
        assert Class.forName("Cow") == Cow.class;

        // Inspect a class
        assert Cow.class.getName() == "Cow";
        assert Cow.class.getSuperclass() == Animal.class;
        assert Animal.class.getDeclaredMethods().length == 2;
    }
}
```

Given a class object, we can obtain its superclass, package, and (arrays of) its constructors, fields, methods, implemented interfaces, and other structural properties. We can use constructor objects to create new instances, field objects to read and set object properties, and method objects to invoke methods whose names are not available until runtime. The following contrived example illustrates instance creation and method invocation:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Class<?> c = Class.forName("Horse");
        Constructor<?> ctor = c.getConstructor(String.class);
        Animal h = Animal.class.cast(ctor.newInstance("CJ"));
        Method m = h.getClass().getMethod("speak");
        assert m.invoke(h).equals("CJ says neigh");
    }
}
```

It's rare for "regular" applications to use metaprogramming facilities, but tools such as GUI builders, and the popular Spring [91] and Hibernate [98] frameworks, perform introspection and code generation, hiding these powerful operations behind easy-to-use annotations. In addition, libraries such as ASM [84] and javassist [56] allow the Java programmer to manipulate Java bytecode, the binary instructions that run on the JVM. These frameworks and libraries are well beyond the scope of this book, but their existence and popularity points out that Java need not be considered only a "static" language.

JAVA WRAP UP

In this chapter, we were introduced to Java. We learned that:

- Programs are made up entirely of classes (which include regular classes, interfaces, records, and emums).
- Java requires that all variables be type-constrained, does most of its type checking at compile time, and makes a strong distinction between type and class.
- Like Lua and JavaScript, Java has both value and reference types, though conversion between the two via autoboxing and unboxing is not difficult.
- A Java class can extend only one class but implement multiple interfaces.
- Interfaces can contain only constants or methods. Interface methods can be abstract, static, private, or default. Language rules prevent confusion or ambiguity surrounding multiple inheritance.
- Interestingly, Java arrays are covariant but collections (from the standard library) are invariant.
- Generic types and methods allowing parameterization by type. Type parameters sometimes require wildcards to enable certain operations to work on objects that can be parameterized by any type, or any type family.
- Java specifies that `null` belongs to every static reference type, so programmers often use optionals from the standard library to get around the "billion dollar mistake."
- Java streams provide an excellent abstraction for working with lazy sequential data from a variety of sources.
- Java uses threads for concurrency, and hence provides a large set of both in-language and standard library support for synchronization and thread management. The heavyweight platform threads wrap OS threads and should be few in number and used for CPU-bound tasks. Virtual threads are very lightweight, essentially unbounded in number, and excel at IO-bound tasks.
- Java's rich metaprogramming facilities include support for both introspection and code generation. While indispensable in many frameworks and libraries, typical applications use these techniques only rarely.

For further study in Java, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Java, see the Java page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/java.

EXERCISES

Now it's your turn. Continue exploring Java with the activities and research questions below, and feel free to branch out on your own.

7.1 Install a Java development environment and execute the opening three scripts in the chapter. Locate and practice with Java's REPL, JShell.

7.2 Why might Java's designers have decided not to allow type inference on fields, parameters, and method return types?

7.3 Write the smallest possible example script that highlights the difference between the stream methods `map` and `flatMap`.

7.4 We glossed over the distinction between a UTF-16 code unit and a Unicode code point. Research the difference.

7.5 Research the primitive type `char`. Can a `char` be automatically converted to an `int`? To a `byte`? How exactly in Java do we represent the character U+1F415 DOG?

7.6 The value `null` belongs to every reference type, but not to any primitive type. What happens if you try to unbox `null`?

7.7 Research Java's instance initializers and static initializers, and write an example class that uses each.

7.8 Find out the difference between nested classes, member classes, local classes, and anonymous classes, and give examples of each.

7.9 In the functional interface example in this chapter, what is the class of the expression `x->x*2`? Use `getClass` to find out.

7.10 What exactly is a method reference? Rewrite the expression `reduce(Math::max)` two ways: (1) using the arrow notation, and (2) by instantiating an instance of a class implementing `java.util.function.BinaryOperator`.

7.11 Write a script to find the first Fibonacci number greater than 10^50 by generating an infinite stream of Fibonacci values and using the `findFirst` method on the `Stream` class. You will have to learn about the `BigInteger` class to do this exercise.

7.12 The following definition of static typing appears at http://c2.com/cgi/wiki?StaticTyping: "[Static typing] means that a reference value is manifestly (which is not the same as at CompileTime) constrained with respect to the type of the value it can denote, and that the language implementation, whether it is a compiler or an interpreter, both enforces and uses these constraints as much as possible." Break down and explain the rationale for this definition and show whether it differs from the common understanding of static typing as validating type constraints prior to execution. You will of course need to research the precise definition of reference value.

7.13 Some bloggers have considered the term "unityped" a perjorative and have written in defense of such languages. What are their arguments?

7.14 How is Java's static typing similar to, or different from Python's allowance of type constraints?

7.15 The class `Class` is parameterized. What is the purpose of this type parameter?

7.16 In the `AnimalChorusExample` from this chapter, how would the `chorus` method be different if its parameter were constrained with `List<Animal>` instead of `List<? extends Animal>`.

7.17 In the source code for the standard library, in the class `java.util.Objects`, you will find:

```
public static boolean isNull(Object obj) {
    return obj == null;
}
```

When would a programmer likely find this method preferable to the simple expression it invokes?

7.18 We spent a lot of time discussing optionals, almost as if they were encouraged to be used everywhere as a solution to the billion dollar mistake. However, their implementation in Java is somewhat controversial. Read http://blog.codefx.org/java/dev/design-optional/ and summarize the main points against optionals. What do you think?

7.19 Explain the rationale behind the wildcard parameters in the methods of the `Optional` class listed in this chapter. In particular, why were the lower bounded vs. upper bounded wildcard choices made where they were? Can you think of a general rule-of-thumb for using `extends` vs. `super` based on your analysis?

7.20 In this chapter, we listed the six states for Java threads (new, runnable, blocked, waiting, timed waiting, and terminated). Research the meaning of each, and how threads transition between states.

7.21 Determine if it is possible, prohibited, or required, for Java platform threads to be mapped to real operating system threads.

7.22 In addition to threads, the Java standard library supports *processes*. Research how processes are created and managed in a Java application.

7.23 Enhance the Pokémon script from this chapter to request information from 100 Pokémons. Write an alternative implementation using platform threads instead of virtual threads. Did you notice any difference? (If not, that is okay.)

7.24 Enhance the restaurant example from this chapter to log a message when customer places order.

7.25 We may wish to finish our restaurant simulation with the message `"The restaurant is closing down"` after all of the customers have gone home. Implement this feature in two ways: (a) using `join`, and (b) using a `CountDownLatch`.

7.26 In this chapter, we saw examples of threads responding to being interrupted, but we never explained how to interrupt a thread! Find out how this is done. Enhance the restaurant simulation so that after launching each of the threads, the main thread interrupts a few customers and one cook after a random amount of time has passed.

7.27 We took a little shortcut in our `ReflectionExample` script in this chapter, by declaring that our `main` method could throw any exception. In the Java culture, this is considered bad practice. Rewrite the script so that the throws clause properly lists the individual exceptions that may be thrown.

7.28 Find out about the concept of **type erasure** in Java. Why exactly did Java's designers make this a feature?

CHAPTER 8

Kotlin

Kotlin is a modern programming language originally targeted to the Java Virtual Machine, advertised as being concise, cross-platform, and fun.

First appeared	2011
Created at	JetBrains
Recognized for	Modernizing Java
Notable uses	Enterprise computing, Android applications
Tags	Architecture-neutral, Multithreaded, High-performance
Six words or less	"Modern, safe, concise Java successor"

Java's mix of massive popularity and being somewhat slow to modernize has given rise to a number of successors capable of interoperating with existing Java code, most notably Groovy, Scala, Ceylon, and Kotlin. Each address many of Java's shortcomings (discussed in the last chapter) in various ways, though thanks to Google anointing it the preferred language for Android applications [39], Kotlin, named after the island the Baltic Sea, is the most successful.

This modern language has much to offer not only programmers that *must* work in the Java ecosystem, but to anyone interested in modern language design. Kotlin supports multiple programming paradigms (including procedural, functional, and object-oriented) and sports many features programmers have come to expect: null safety, pattern matching, type inference, extensibility, data classes, coroutines, and anonymous functions. Kotlin programs can compile into Java bytecode for execution on the Java Virtual Machine (alongside existing Java code) or can compile into code that runs directly within a web browser. The standard Kotlin distribution comes with a compiler targeting LLVM, making the language automatically suitable for native applications running on a wide variety of hardware platforms.

As Java interoperability was a design goal of the language, we'll spend some time on comparisons of Kotlin with Java. We'll investigate how Kotlin was or was not able to fully address the noted shortcomings of Java outlined in the previous chapter, while not neglecting the expressive and fun features of this popular and well-loved [53] language.

8.1 HELLO KOTLIN

Here's our first teaser script:

```
for (c in 1..40) {
    for (b in 1..<c) {
        for (a in 1..<b) {
            if (a * a + b * b == c * c) {
                println("$a, $b, $c");
            }
        }
    }
}
```

While Kotlin applications normally have a function called `main` as their starting point, we'll be using Kotlin Scripting to reduce some boilerplate code. Our script has nothing too different from previous languages on our tour, except perhaps the nice operator `..<` for ranges with exclusive upper bounds. String literals require double quotes, and string interpolation shares JavaScript's `${}` notation, though when interpolating a *single* variable we may elide the curly braces. Now to the second teaser:

```
fun printPermutations(a: Array<String>, n: Int) {
    if (n <= 0) {
        println(a.joinToString(separator = "\t"))
    } else {
        for (i in 0 ..< n) {
            printPermutations(a, n-1)
            val j = if (n % 2 == 0) {0} else {i}
            a[n] = a[j].also { a[j] = a[n] }
        }
        printPermutations(a, n-1)
    }
}

printPermutations(args, args.lastIndex)
```

While concise, there are a few language features worth calling out in this script. First, the type annotations on the function parameters are required. Second, you'll notice that named arguments are permitted directly in the call, as we previously saw in Python and Ruby. Third, there's no conditional operator (such as `_?_:_`). Instead, the `if` construct can be used both as a statement *and* as an expression. When used in an expression context, the two arms produce the expression's value. Finally, we used a rather interesting idiom for swapping array elements. The expression x`.also` *block* first evaluates the receiver x, then executes the block, then produces the previously evaluated value of the receiver. In our script, this means: (1) the current value of $a[j]$ is evaluated; (2) within the block, array slot j is overwritten with $a[n]$; and (3) since the `also` method returns the original value of $a[j]$, the value of $a[n]$ is properly updated.

The third teaser reads from standard input, made easy by the `readLine()` and Kotlin's expressive sequence operations:

```
data class Player(
    val team: String,
    val name: String,
    val games: Int,
    val ppg: Double
) {
    companion object {
        fun fromLine(line: List<String>): Player {
            val points = line[3].toDouble()
            val games = line[2].toInt()
            return Player(line[0], line[1], games, points / games)
        }
    }

    fun reportLine(): String {
        return String.format("%-22s%-4s%8.2f", name, team, ppg)
    }
}

generateSequence { readLine() }
    .map { it.split(",") }
    .map { Player.fromLine(it) }
    .filter { it.games >= 15 }
    .sortedByDescending { it.ppg }
    .take(10)
    .map { it.reportLine() }
    .forEach { println(it) }
```

This script is made from a single class declaration and a call to `generateSequence`. The sequence sources its values by repeatedly calling `readLine()`, which produces the next line from standard input, or `null` when no more input remains (terminating the sequence). Each line is split and turned into a player object; the sequence is filtered to keep only the players that have played in 15 games, is sorted and clipped to its top ten; then each remaining player is reported with a nice output format. The anonymous functions used here all have a single parameter and can hence use a shorthand syntax where the implicit parameter name is `it`.

You may notice a similarity between the player **data class** in this example with the Java player `record` in the previous chapter. The obvious syntactic difference is that Kotlin has no "static methods." Instead, a Kotlin class can be given one (and only one) companion object, whose methods have access to that of the class itself. Data classes, companion objects, and sequences are just a few of the many interesting Kotlin features we will now explore.

8.2 THE BASICS

Kotlin programs are built from a collection of source files, each of which consists of a sequence of declarations (variables, functions, classes, objects, etc.) and statements. Variables marked `val` are read-only (can only be assigned to once), while those marked `var` may be reassigned. Type constraints can usually be inferred if given an initializer.

```
val message = "Hello World!"  // Type inferred
var found: Boolean    // Type required when no initializer
// println(found)     // Will not compile! 'found' not initialized
found = true
```

Kotlin's basic numeric types include the 8, 16, 32, and 64 bit signed integers `Byte`, `Short`, `Int`, and `Long`; the corresponding unsigned integer types `UByte`, `UShort`, `UInt`, and `ULong`; and the IEEE binary32 and binary64 types `Float` and `Double`. The boolean type is `Boolean` and the `String` type consists of elements of type `Char`. Because chars are, like Java and JavaScript, UTF-16 character codes, you'll have to get code points via a library method, namely `codePoints`:

```
val message = "←\uD83C\uDF53→"
// LEFTWARDS ARROW, STRAWBERRY, RIGHTWARDS ARROW
for (codePoint in message.codePoints()) {
    println(Character.getName(codePoint))
}
```

Kotlin's arrays are fixed-size and always mutable, so programmers will generally opt for lists instead, as they can grow and shrink, and can even be made read-only. In addition to lists, the standard library features a rich set of other collections, including sets and maps:

```
val berries = listOf("blueberry", "blackberry", "strawberry")
val team = mutableListOf<String>()
team.add("Ashleigh")
val smallPrimes = mutableSetOf(2, 3, 5, 7, 11)
var capitals = mapOf(
    "Senegal" to "Dakar",
    "Vietnam" to "Hanoi",
    "France" to "Paris")

assert(berries.size == 3)
assert(berries[0] == "blueberry")
assert(team == listOf("Ashleigh"))
assert(3 in smallPrimes)
assert(capitals["France"] == "Paris")
```

Objects have **properties** and member functions, sometimes called **methods**. Properties may optionally have initializers, getters, and setters. The latter two are useful to maintain relationships between properties:

```
object Boundary {
    var radius = 1.0
    var area: Double
        get() = Math.PI * radius * radius
        set(value) {
            radius = Math.sqrt(value / Math.PI)
        }
    fun draw() = "I can't draw yet"
}
```

```
assert(Boundary.radius == 1.0)
Boundary.radius = 2.0
assert(Boundary.area == 4 * Math.PI)
Boundary.area = 64.0 * Math.PI
assert(Boundary.radius == 8.0)
assert(Boundary.draw() == "I can't draw yet")
```

Function parameters can have defaults (unlike Java), rest parameters are introduced with the keyword `vararg`, and function arguments can be named in a call. The spread operator is a prefix `*`. Type annotations on parameters are required, but if the function body is just a single expression, the return type can be inferred if you use the shorthand definition syntax.

```
// Default arguments
fun log(message: String, level: String = "INFO"): String {
    return "[$level] $message"
}

// Positional and named arguments
assert(log("Starting") == "[INFO] Starting")
assert(log(message = "Starting") == "[INFO] Starting")
assert(log("Out of memory", level = "ERR") == "[ERR] Out of memory")
assert(log(level = "DEBUG", message = "Here") == "[DEBUG] Here")

// Single Expression Function
fun square(x: Int) = x * x
assert(square(5) == 25)

// Varargs
fun weightedSum(vararg numbers: Int, multiplier: Int): Int {
    return numbers.sum() * multiplier
}
assert(weightedSum(1, 2, 3, multiplier = 10) == 60)
val smallPrimes = intArrayOf(2, 3, 5, 7, 11)
assert(weightedSum(*smallPrimes, multiplier = -2) == -56)
```

If a function has no explicit return, it is assumed to return the value `()` of type `Unit`, which you may, but do not have to mention explicitly if your function body is a block. If the function does not return at all, the return type is `Nothing`:

```
fun greet(name: String) { println("Hello, $name") }
fun part(name: String): Unit { println("Bye, $name") }
fun poof(): Nothing { throw Exception("poof") }
```

Function values may be stored in variables, and passed to and returned from other functions. To store or pass a declared function or method, you'll need to use the function reference operator, `::`, to obtain a function value. Function values can also be written as **anonymous functions** or **lambdas**, as illustrated here:

```
fun twice(f: (Int) -> Int, x: Int): Int = f(f(x))

// Function reference operator (::) for declared funs & methods
fun square (x: Int): Int = x * x
assert(twice(::square, 5) == 625)
assert(twice(Int::inc, 3) == 5)

// Anonymous functions: fun without a name
assert(twice(fun(x: Int): Int = x * x, 5) == 625)

// Lambda expressions: expression in curly braces
assert(twice({ x -> x * x }, 5) == 625)
assert(twice({ it * it }, 5) == 625)
```

Note that lambdas with a single parameter do not need the arrow (`->`) but can instead refer to the single argument as `it`.

If the final argument in a call is itself an anonymous function, it can be written *after* the parentheses—a syntactic device known as a **trailing closure**. Not only does this make pipeline-oriented code very readable (as we saw in the third teaser program at the beginning of this chapter), it can give the appearance of extending the language with new "statements":

```
fun unless(condition: Boolean, action: () -> Unit) {
  if (!condition) {
    action()
  }
}

unless (1 == 2) {
  println("1 is not equal to 2")
}
```

Use `if` for simple conditionals and `when` for multiway (like Java's `switch`). Both `if` and `when` can be used in a statement context or an expression context. The `when` branches are tried in order until one is successful. A `when` without an argument is a nice replacement for a chain of ifs and else ifs:

```
// If expression
val latitude = 34.0
val hemisphere = if (latitude > 0) {"North"} else {"South"}

// When expression with no argument
val (x, y) = Pair(8, 9)
val comparison = when {
    x > y -> "Greater"
    x < y -> "Less"
    else -> "Equal"
}
```

```
// When expression with an argument
var secret = "999"
var keys = listOf(21, 8, 55)
val message = when (x) {
    0, 1 -> "Zero or one"
    secret.toInt() -> "You found it"
    in keys -> "A key"
    !in 3 until 12 -> "3 through 11"
    else -> "Unexpected"
}

assert(hemisphere == "North")
assert(comparison == "Less")
assert(message == "A key")
```

We've packed a lot into our section on Kotlin basics, but there is of course much more to cover. While we can't look at the entire language, there are a number of language elements worthy of a deeper look. We'll start with classes and make our way to concurrency.

8.3 CLASSES

As in Java, a class defines a kind of factory for creating objects. The **primary constructor** appears in the class header (along with any fields you want to initialize with the constructor), while the class body holds properties, methods, secondary constructors, initializer blocks, companion objects, and inner classes.

```
class Circle(
    val radius: Double,
    private var x: Double,
    private var y: Double,
) {
    val area = Math.PI * radius * radius
    val circumference = 2 * Math.PI * radius

    constructor(descriptor: String): this(
        descriptor.split(":").map { it.trim().toDouble() }
    )
    private constructor(a: List<Double>) : this(a[0], a[1], a[2])
    init { println("Circle created!") }

    fun move(dx: Double, dy: Double) {
        x += dx
        y += dy
    }
    override fun toString(): String {
        return "Circle with radius $radius at ($x,$y)"
    }
}
```

```
val c = Circle(2.0, 0.0, 0.0)      // use primary constructor
c.move(13.0, 21.0)                 // invoke a method
println(c)                         // indirectly calls toString()
val newC = Circle("3.0:1.0:2.0")   // use secondary constructor

assert(c.radius == 2.0)
assert(c.area == 4.0 * Math.PI)
assert(newC.circumference == 6.0 * Math.PI)
```

The circle class above has a primary constructor taking three doubles—the radius, center x, and center y respectively. The x and y properties are mutable, but their visibility is set to `private`, meaning they cannot be seen outside the class. The `radius` property happens is immutable (due to `val`), but it has the default visibility of `public`. The properties `area` and `circumference` are initialized but have no getters and setters, as their values cannot change. The `move` method is able to mutate x and y, since code inside the class can access private members.

There are two secondary constructors for enabling the creation of circles from a descriptor string. In Kotlin, all secondary constructors must delegate to the primary constructor via the keyword `this`. Our class features one initialization block. These blocks are executed whenever an instance is created.

Every class in Kotlin is a descendant of the built-in class `Any`, with three methods `equals`, `hashCode`, and `toString`. When defining a new class, method implementations may override ancestor implementations; such methods are marked `override`. Classes may not be subclassed by default; they must be marked `open` to allow this. Similarly, methods may not be overridden unless marked `open`:

```
open class User(name: String) {
  open val minPasswordLength = 12
}

class Admin(name: String): User(name) {
    override val minPasswordLength = 18
}
```

Abstract classes and interfaces are automatically open and do not require the `open` keyword. Similarly, abstract methods of such classes and interfaces need not be marked `open` to be overridden. Our recurring animals example provides an illustration:

```
abstract class Animal(val name: String) {
    fun speak() = "$name says ${sound()}"
    abstract fun sound(): String
}

class Cow(name: String): Animal(name) {
    override fun sound() = "moooo"
}
```

```
class Horse(name: String): Animal(name) {
    override fun sound() = "neigh"
}

class Sheep(name: String): Animal(name) {
    override fun sound() = "baaaa"
}

val h = Horse("CJ")
assert(h.speak() == "CJ says neigh")
val c = Cow("Bessie")
assert(c.speak() == "Bessie says moooo")
assert(Sheep("Little Lamb").speak() == "Little Lamb says baaaa")
```

In order to limit the number of subclasses to a fixed set (known at compile time), mark the class `sealed`. Sealed classes, too, are automatically open. The following script limits the number of tree subclasses to two. The script will also show that Kotlin supports generic types, which we covered previously in TypeScript and Java.

```
sealed class Tree<T> {
    class Empty<T> : Tree<T>()
    class Node<T>(
        val left: Tree<T>, val value: T, val right: Tree<T>
    ) : Tree<T>()

    fun size(): Int = when (this) {
        is Empty -> 0
        is Node -> left.size() + 1 + right.size()
    }
}

assert(Tree.Empty<Int>().size() == 0)

val t: Tree<Int> = Tree.Node(
    Tree.Node(Tree.Empty(), 3, Tree.Empty()),
    5,
    Tree.Node(Tree.Empty(), 8, Tree.Empty()))
assert(t.size() == 3)
```

Application designers often distinguish **active** and **passive** classes. Instances of active classes act as agents with rich behavior, initiating and carrying out complex application functionality on their own. Instances of passive classes simply hold data. Kotlin provides the **data class** for these simple containers.

You can mark a class as a data class provided the class is defined at the top level (not nested), its primary constructor has at least one parameter, all constructor parameters are marked `val` or `var`, and the class is also not open, sealed, nor abstract. For your data class, Kotlin will automatically generate the methods `equals`, `hashCode`, `toString`, `copy`, and numbered component methods. Let's explore:

```
// A one liner is all you need
data class User(val id: Int, val email: String)

// Three users
val friend = User(133, "alice@example.com")
val colleague = User(133, "alice@example.com")
var guest = User(205, "bob@example.com")

// Autogenerated toString
assert(friend.toString() == "User(id=133, email=alice@example.com)")

// Autogenerated equals
assert(friend == colleague)
assert(friend.equals(colleague))

// Autogenerated hashCode
assert(friend.hashCode() == colleague.hashCode())

// Autogenerated copy
val newFriend = guest.copy()
assert(newFriend.id == 205)
assert(newFriend.email == "bob@example.com")
assert(newFriend == guest)
assert(newFriend !== guest) // different instances

// Autogenerated component methods
assert(friend.component1() == 133)
assert(friend.component2() == "alice@example.com")

// Destructuring
val (id, email) = friend
assert(id == 133)
assert(email == "alice@example.com")
```

Autogenerating methods not only cuts down on the number of lines of code necessary to implement a class, but frees the programmer from having to remember to update all the boilerplate methods whenever the structure of a class changes.

We'll close our look at Kotlin classes with something fun: member functions with a single argument can be made to be written as infix operators:

```
class Employee(val name: String) {
    private var manager: Employee = this
    constructor(name: String, manager: Employee) : this(name) {
        this.manager = manager
    }
    infix fun manages(e: Employee) = e.manager == this
}
```

```
val alice = Employee("Alice")
val bob = Employee("Bob", manager = alice)
assert(alice manages alice)
assert(alice manages bob)
```

There are quite a few standard (nonsymbolic) infix operators, including `to` for making pairs; `until`, `downTo`, and `verb` for making ranges; string functions `matches`, `compareTo`, and `zip`; and the bitwise operators `and`, `or`, `xor`, `shl`, `shr`, and `ushr`. All named operators, whether built-in or programmer-defined, exist at the *same* precedence level[1] and are naturally left-associative. After all, `x or y and z` simply sugars `x.or(y).and(z)`.[2]

8.4 EXTENSIONS

In several of the languages visited so far, methods can easily be added to classes at runtime. In Lua, you add functions to the metatable; in JavaScript and TypeScript, you add to the prototype. Kotlin's approach is interesting: you don't really add properties or classes, but you get the syntactic convenience of doing so with extension properties and methods.

Extensions can be defined for classes you write, someone else writes, or even those from the standard library. Here are a few examples:

```
fun Int.factorial(): Int {
    return (1..this).fold(1) { x, y -> x * y }
}

assert(0.factorial() == 1)
assert(10.factorial() == 3628800)

val String.scalarCount: Int
    get() = this.codePointCount(0, this.length)

assert("".scalarCount == 0)
assert("←\uD83C\uDF53→".scalarCount == 3)

fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    this[index1] = this[index2].also { this[index2] = this[index1] }
}

val numbers = mutableListOf(34, 8, 21, 5, 1)
numbers.swap(1, 3)
assert(numbers == mutableListOf(34, 5, 21, 8, 1))
```

Though extension functions are not strictly methods, they may be written infix if they take a single parameter. This makes sense since extension functions, like infix operators, are essentially syntactic sugar.

[1] See [59] for the complete operator precedence chart for Kotlin.

[2] Other languages, notably Haskell and Swift, allow programmers to define their own infix operators and specify their own precedence levels.

8.5 COMPANION OBJECTS

There are a variety of approaches to the problem of defining properties and objects which logically "belong with" a class but are not tied to any *instance* of the class. We've seen Java's static fields and methods, Python's `@classmethod` and `staticmethod` decorators, and JavaScript's ability to define properties directly on constructor functions rather than in prototypes. Rust (which we'll see in a future chapter) has the idea of *associated functions*. Kotlin's approach is to attach properties and functions to a **companion object**.

In the following exploration we have a `Point` data class with a property for *the* origin and a function to compute the midpoint between two points. how members of the companion object use the class name as prefix.

```
import kotlin.math.hypot
import kotlin.math.sqrt

data class Point (val x: Double, val y: Double) {
    companion object {
        val ORIGIN = Point(0.0, 0.0)
        fun midpoint(p: Point, q: Point) = Point(
            (p.x + q.x) / 2.0, (p.y + q.y) / 2.0)
    }
    fun distanceFromOrigin() = ORIGIN distanceTo this
    fun mirror() = Point(-x, -y)
    infix fun distanceTo(p: Point) = hypot(x - p.x, y - p.y)
}

val oneOne = Point(1.0, 1.0)
val halfHalf = Point(0.5, 0.5)
assert(Point.midpoint(Point.ORIGIN, oneOne) == halfHalf)
assert(Point(4.0, -3.0).distanceFromOrigin() == 5.0)
assert(Point(1.0, 1.0) distanceTo Point(0.0, 0.0) == sqrt(2.0))
assert(oneOne.mirror() == Point(-1.0, -1.0))
```

An advantage of the companion object approach is that it allows programmers to think of *all* methods as instance methods, rather than mentally distinguishing class vs. instance methods. Functions defined in a companion object *are* instance methods whose receiver is the companion object. In the end-of-chapter exercises, you will explore whether the same mental model applies to other languages.

8.6 NULL SAFETY

One of Java's more interesting design decisions was the inclusion of the value `null` into every reference type in the static type system. For example, a Java variable *s* declared with type `String` may be assigned the value `null`, and the compiler will not issue an error. The invocation `s.toLowerCase()` is also not a compile-time error (the phrase is—surprisingly—*statically* type correct!), though a good IDE or linter may warn you of impending run-time doom. Indeed, if the method call is executed with `null` as the receiver, a run-time error occurs, as `null` is not part of the *dynamic* string type.

Nearly every other language featured in this book with a static type system keeps `nulls` out of other types unless the programmer explicitly creates a union type including it. Kotlin is no exception. Kotlin calls such a union a **nullable type**, and comes with a nice syntax. Variables constrained to type `String` may *only* hold strings; those constrained to `String?` may hold strings or `null`.

```
// Name is required, manager is optional
class Employee(val name: String, val manager: Employee? = null) {
    infix fun manages(e: Employee) = e.manager == this
}

val alice = Employee("Alice") // Alice has no manager
val bob = Employee("Bob", manager = alice)

// Compiler will not allow using an Employee? as an Employee
// println(alice.supervisor.name) // Will not compile
// println(bob.supervisor.name) // Will not compile either!
```

To "unwrap" the optional, you can, like in TypeScript make use of narrowing, or use the safe call operator `?.`, which produces `null` if the receiver is null or the property value otherwise. Suffixing this operator with `let` will unwrap, and if not null, pass the unwrapped value to a closure:

```
// You can always use the "safe call" operator
assert(alice.manager?.name == null)
assert(bob.manager?.name == "Alice")

// Use the safe call operator with let
bob.manager?.let {
    // Only gets here if supervisor is not null
    assert(it.name == "Alice")
}
```

Kotlin's **Elvis operator**[3] coalesces nulls, that is, x`?:`y produces x if x is non null and y otherwise. You'll see this operator used frequently in functions that wish to exit early on errors, as Kotlin's designers used `throw` and `return` for expressions rather than statements:

```
// Demonstrate ?: (Elvis operator)
val supervisor = alice.manager?.name ?: "YOUR NAME HERE"
assert(supervisor == "YOUR NAME HERE")

fun reportingChain(e: Employee?, from: List<String>): List<String> {
    val name = e?.name ?: return from
    return reportingChain(e?.manager, from + name)
}
assert(reportingChain(bob, listOf()) == listOf("Bob", "Alice"))
```

There's a convenient method to remove nulls from sequences:

[3] As with Python's walrus operator, this one gets its name from looking at it sideways. You should see a pair of eyes and wavy hair.

```
// Demonstrate filterNotNull
val employees = listOf(alice, bob, null, bob)
val managers = employees.filterNotNull().map { it.name }
assert(managers == listOf("Alice", "Bob", "Bob"))
```

And if you are *really* sure you won't have a null, you can use the dangerous `!!` operator which unwraps and throws a *runtime* exception for `nulls`:

```
// The not-null assertion operator, !!, use sparingly if at all
println(bob.manager!!.name) // Will print "Alice", lucky duck
try {
    println(alice.manager!!.name) // Please no
} catch (e: NullPointerException) {
    println("Caught NullPointerException")
}
```

8.7 COROUTINES

If interoperating with Java, the Kotlin programmer has access to Java's `Thread` class and `Runnable` interface, and can thus write multithreaded code with Java platform threads—albeit with a lighter syntax and a convenient `thread()` function for launching. However, Kotlin provides its own lightweight concurrency primitive, the **coroutine**. Like those in other languages we've seen, Kotlin coroutines can suspend and resume and communicate and wait for each other; they are useful for generators, cooperative multitasking, and general asynchronous activities. You'll notice similarities with Java's virtual threads, as they are extremely lightweight and efficient. We'll provide only a simple example here, referring you to the *Kotlin Coroutines Guide* [58] for a more in-depth look.

Here is a Kotlin version of the Java program in the last chapter for fetching Pokémon information from an API, augmented with helpful log messages:

```
@file:DependsOn(
    "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1",
    "com.squareup.okhttp3:okhttp:4.11.0",
    "org.json:json:20230618",
)

import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import org.json.JSONObject

val baseUrl = "https://pokeapi.co/api/v2/pokemon/"
val pokemonNames = listOf("ditto", "pikachu", "mew", "weedle", "eevee")

data class PokeInfo(val name: String, val height: Int, val weight: Int)
```

```
suspend fun fetchPokemon(name: String) {
    println("Fetching $name on ${Thread.currentThread().name}")
    val request = Request.Builder().url("$baseUrl$name").build()
    val response = OkHttpClient().newCall(request).execute()
    if (!response.isSuccessful) {
        println("Cannot fetch $name: HTTP ${response.code}")
        return
    }
    val jsonObject = JSONObject(response.body?.string() ?: return)
    println(PokeInfo(
        jsonObject.getString("name"),
        jsonObject.getInt("height"),
        jsonObject.getInt("weight")))
}

runBlocking {
    for (name in pokemonNames) {
        launch(Dispatchers.IO) {
            fetchPokemon(name)
        }
    }
    println("The fetches are underway")
}
println("The fetches are done")
```

In this script, `runBlocking` creates a coroutine and blocks its thread—the main thread, since it is called at the top level—until completion. The function `launch` is also a coroutine builder: it simply starts a new coroutine running without waiting for it to finish. Use `launch` for "fire-and-forget" scenarios where the launched coroutine does not return anything. If you do need something produced by a job, use the `async` coroutine builder to launch a coroutine and immediately return a `Deferred` value (similar to a future or promise in other languages). The deferred value has an `await` method which does a nonblocking wait for the deferred value to become present.[4] There's an `awaitAll` method on sequences of deferred objects, too.

The script launches a few coroutines, one for each Pokémon we wish to fetch from an API. The coroutines invoke the **suspendable function** `fetchPokemon`. The first argument to `launch` is the *IO Dispatcher*. Kotlin's dispatchers determine which platform threads coroutines run on. The built-in IO dispatcher has a thread pool specially designed to run coroutines that spend a lot of time blocked on IO requests (e.g., waiting for file, database, and network operations). You may notice that we never explicitly invoked a "join" operation to wait for any of the launched coroutines to complete. Instead, we made use of Kotlin's structured concurrency feature: by defining the Pokemon-fetching coroutines *inside* an outer coroutine, the outer one will automatically wait for its inner ones to finish.

One run of this program produced:

[4] Note how `async` and `await` are not Kotlin keywords. The former is a function and the latter is a method of the `Deferred` type.

```
Fetching ditto on DefaultDispatcher-worker-1
Fetching pikachu on DefaultDispatcher-worker-3
Fetching mew on DefaultDispatcher-worker-2
Fetching weedle on DefaultDispatcher-worker-5
The fetches are underway
Fetching eevee on DefaultDispatcher-worker-4
PokeInfo(name=ditto, height=3, weight=40)
PokeInfo(name=weedle, height=3, weight=32)
PokeInfo(name=eevee, height=3, weight=65)
PokeInfo(name=pikachu, height=4, weight=60)
PokeInfo(name=mew, height=4, weight=40)
The fetches are done
```

As was our custom in this chapter, we've used Kotlin scripting, rather than fully-featured Kotlin applications in order to illustrate language concepts. One obvious downside of Kotlin scripting is apparent at the top of our script: hardcoded dependencies on external libraries. See the Kotlin documentation for instructions on building applications with systems such as Maven and Gradle that take care of dependency management and similar tasks.

KOTLIN WRAP UP

In this chapter, we were introduced to Kotlin. We learned that:

- Kotlin was influenced by Java and interoperates well with Java, but today runs on a variety of platforms other than Java's JVM.
- The language has a rich set of static types, including both signed and unsigned integers, floats, booleans, characters, and a variety of array, function, and object types. In fact, numbers, booleans, and characters are objects, in the sense that methods may be called on them.
- It is possible to declare an object without first declaring a class. Objects have properties and methods. Properties may have getters and setters.
- Functions are quite flexible: defaults, named arguments, and varargs are all supported. To use functions and methods as values, the function reference operator (`::`) is needed. Kotlin distinguishes anonymous functions from lambdas.
- Trailing closures allow for interesting expression forms.
- The class syntax is lighter than Java's, with the primary constructor appearing in the class header.
- Members of a class are public by default; the other modifiers are private, protected, and internal.
- Classes must be marked open to be extended. Abstract and sealed classes are automatically open. (Interfaces are as well, though these were not covered in this chapter.)
- All classes are descendants of the class `Any`, which defines three methods that classes can, and often do, override. Data classes automatically override them, and provide autogenerated copy and component access methods as well.
- Extensions provide nice syntactic sugar.

- Companion objects provide functionality that most languages tend to use "static" or "type-level" members for.
- Kotlin pays a lot of attention to protecting you from nulls. To be able to use the value `null`, you will need a nullable type. Operators like `?.` and `?:` make optional (nullable) data relatively easy to work with.
- Coroutines are the preferred mechanism for concurrency. However, we covered a vanishingly small amount of what Kotlin offers in this area.

For further study in Kotlin, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Kotlin, see the Kotlin page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/kotlin.

EXERCISES

Now it's your turn. Continue exploring Kotlin with the activities and research questions below, and feel free to branch out on your own.

8.1 Locate the Kotlin home page and make your way to the docs page at https://kotlinlang.org/docs/home.html. Once there, find and go through the Tour, the Overview, and the Concepts sections. Make notes of language features not encountered in this chapter.

8.2 Run the teaser program for generating Pythagorean triples on the official Kotlin playground at https://play.kotlinlang.org/.

8.3 Install Kotlin on your local machine and run several of the scripts from this chapter.

8.4 Find out how to iterate through the UTF-8 bytes of a string.

8.5 Kotlin's `==` operator on lists does a shallow comparison of each list element. How do you compare the references to each list for sameness? How do you perform deep copies?

8.6 In a Kotlin program, variables defined at the top level are called properties, the same word used for certain members of objects. Why is this? Can top-level properties also have getters and setters? Experiment with a small script of your own design.

8.7 Is the compiler happy with an attempt to assign a `UInt` to an `Int`-constrained variable? An `Int` to a `UInt`? A `Int` to a `Double`?

8.8 Research the `lateinit` keyword and construct an example showing its use.

8.9 In this chapter, we saw an example of the spread operator used when passing arguments to a `varargs` parameter. Can the operator be used to construct lists, as it can in JavaScript, Python, and Ruby?

8.10 Can named arguments appear before positional arguments in a call?

8.11 Can the function `fun f() {println("ok")}` be called in an expression? If so, what value does it produce? If not, what error does the compiler report?

8.12 Find as many differences as you can between an anonymous function and a lambda in Kotlin. Ask an AI chatbot to construct some interesting examples.

8.13 Define a function called `loop` which takes one argument, a lambda, and executes it repeatedly, forever. Show how calling your `loop` function with a trailing closure is indistinguishable from adding a new loop statement to the language.

8.14 Research how and when functions are best defined with the `inline` keyword.

8.15 Research the degree to which arms of a `when` construct can be used to deconstruct patterns, if at all.

8.16 In the `Circle` class example in this chapter, we were able to make `area` and `circumference` properties. If we modify the class to have a mutable radius property, would would have to rewrite these properties to be methods? Would *any* modifications be required at all?

8.17 Add a few interesting methods to the `Tree` class from this chapter.

8.18 Extend the `Tree` class of this chapter to a binary search tree. Research will be required, as you'll need to learn Kotlin's way of bounding the generic type to be comparable.

8.19 Write extensions for `Int` to (1) determine if the integer is prime and (2) produce the string representation of the representation in any base between 2 and 36, inclusive.

8.20 Create a Kotlin class for two-dimensional vectors, similar to those we saw in the chapters on Lua and Python, but add additional functionality to increase your practice. Note that Kotlin, like Lua and Python, treats operator invocations as sugar for method class, so your vector addition operation will look like this:

```
operator fun plus(v: Vector): Vector {
    return Vector(i + v.i, j + v.j)
}
```

Also try to use companion objects and infix methods where sensible.

8.21 Recall that the type T`?` contains all and only the values of T plus the value `null`, while the type `Nothing` contains no values at all. How exactly can you characterize the type `Nothing?`? In particular, how many values belong to this type?

8.22 Port the restaurant simulation from the Java chapter to Kotlin. Do some research to see in Kotlin's channels are helpful in your task.

8.23 Learn how to build Kotlin command line applications with Gradle. Rewrite a handful of scripts from this chapter as command line applications. Make sure to include the Pokémon API example in your set of rewritten programs. Use the most up-to-date dependencies.

CHAPTER 9

Julia

Julia is a modern language for scientific and technical computing.

First appeared	2012
Creator	Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
Recognized for	Multi-dispatch, Macros, Multicore support
Notable uses	Data Science, Statistics, Numerical Analysis
Tags	High-performance, Dynamic, Multi-dispatch
Six words or less	A dynamic language for technical computing

Julia is targeted to high-performance numeric and scientific computing, where it competes with Matlab, Python, R, and Fortran—languages with well-established and mature libraries. While relatively new, Julia brings an impressive set of features not only of interest to the statistics and data science communities (such as its math-friendly syntax, and support for vectors and matrices), but also to those looking for an efficient and powerful general-purpose language.

In this chapter, we'll tour this new language, beginning with our standard three introductory programs, then dive right into Julia's type system. We'll see that Julia shares much in common with TypeScript, having built-in support for **abstract types**, **parametric types**, sum and product types, and subtyping. We'll take the opportunity to review covariance, contravariance, and invariance, which we previously discussed in our chapter on Java. As a language designed with statistical computing, linear algebra, and data science applications in mind, we will necessarily focus on Julia's expressive and powerful array operations.

Next, we will introduce **multiple dispatch**, an important aspect of Julia's support for object-oriented programming. Previous languages on our tour select the appropriate method to execute based on the type of a single *receiver* at run time, but Julia goes farther, selecting based on the types of all of the method's *arguments*. We'll then look at metaprogramming facilities, which go well beyond what we've seen in the previous chapters. Julia can introspect not only objects and types, but also expressions and the internal structure of the program itself. We'll see **hygienic macros** for the first time. We'll close with a look at the rather elegant language support for parallel and distributed computing.

9.1 HELLO JULIA

We'll start with the usual right triangle script:

```
for c = 1:40, b = 1:c-1, a = 1:b-1
    if a * a + b * b == c * c
        println("$a, $b, $c")
    end
end
```

Julia's *ranges* (of the form `x:y`) have inclusive bounds, and string interpolation uses `$`. The `for`-statement has a neat compact form that avoids the usual requirement of nested loops. Heap's algorithm follows:

```
function printpermutations(a, n)
    if n <= 0
        println(join(a, "\t"))
    else
        for i = 1:n-1
            printpermutations(a, n-1)
            local j = iseven(n) ? i : 1
            a[j], a[n] = a[n], a[j]
        end
        printpermutations(a, n-1)
    end
end

printpermutations(ARGS, length(ARGS))
```

Julia prefers `length(a)` to `a.length`; even `join` and `split` look more like top-level functions than methods on arrays and strings—though they really *are* methods as we will soon see. The familiar ternary conditional operator is present, and swaps are easy to write. And, like Lua but unlike most other languages, the lower bound of arrays is 1.

Next is our data science teaser:

```
using Printf

struct Player
    team::String
    name::String
    games::Int
    ppg::Float64
end

function Player(line::Vector{SubString{String}})
    points = parse(Float64, line[4])
    games = parse(Int, line[3])
    return Player(line[1], line[2], games, points / games)
end
```

```
report(p::Player) = @sprintf("%-22s%-4s%8.2f", p.name, p.team, p.ppg)

players = Vector{Player}()
for line in eachline(stdin)
    player = Player(split(line, ","))
    if player.games >= 15
        push!(players, player)
    end
end
sort!(players, by=player -> player.ppg, rev=true)
for player in players[1:10]
    println(report(player))
end
```

The expression `eachline(stdin)` gives us a line-by-line iterator over standard input. We split each line into a vector of substrings which we use to construct a player object. Julia does not define class constructs with nested data fields and methods. Instead, a `struct` defines fields and an autogenerated constructor that takes one parameter per field. Additional constructors are just functions with the name of the struct. "Methods" are simply functions that take an object of the type as a parameter. We've defined `report` with the shorthand syntax. The script also highlights Julia's syntax for range-based indexing, named arguments in calls, and the convention for using `!` as the final character of a mutating function.

9.2 THE BASICS

From our introductory programs, you may have noticed Julia looks a little like Lua. Structured statements are keyword terminated (e.g., `while`...`end`, `for`...`end`, `if`...`elseif`...`end`, `try`...`catch`...`end`), indexes start at 1 instead of 0, local variables are introduced with the `local` keyword, and there is no `class` keyword. Julia's coroutines are similar to Lua's, too. But Julia adds some nice syntactic forms. Multiplication can sometimes use juxtaposition, functions can be anonymous (`->`), composed (∘) and piped (`|>`), and operators can be **broadcast** by prefixing with a dot (`.`):

```
@assert (5√9, 17 / 5, 17 ÷ 5) == (15, 3.4, 3)
@assert iseven(sign(log2(exponent(8)))) == false
@assert 8 |> exponent |> log2 |> sign |> iseven == false
@assert [1,2,3] |> c -> map(x -> 3x+1, c) == [4, 7, 10]

@assert 5 .+ [2, 8, -2] == [7, 13, 3]
@assert [8, 13, 5] .^ 2 == [64, 169, 25]
@assert .√([9, 16, 25]) == [3, 4, 5]
@assert (1:9 .|> x -> x^2) |> filter(iseven) |> sum == 120
```

In addition to `function`...`end`, we have two alternate ways to define functions:

```
f(x, y) = 2x + y            # call-like form on left-hand side
g = x -> x * x              # arrow notation
```

Functions can have optional and keyword parameters. You can omit the keyword `return` when returning the last expression in a function (as in Ruby):

```
function f(a, b=1 ; c=2, d=3)
    [a, b, c, d]
end

@assert f(100) == [100, 1, 2, 3]
@assert f(100, 200) == [100, 200, 2, 3]
@assert f(100, d=10) == [100, 1, 2, 10]
@assert f(100, 200, d=400, c=300) == [100, 200, 300, 400]
```

A semicolon separates the positional and keyword parameters, and keyword parameters must be given default values (in other words, all keyword parameters are also optional). Julia uses three dots (`...`) for packing and unpacking. Positional parameters pack and unpack into iterable collections of values; keyword parameters pack and unpack into arrays of (*symbol*, *value*) pairs:

```
f(x, y...) = y
@assert f(1, 2, 3) == (2, 3)

g(x; y...) = collect(y)
@assert g(1, a=2, b=5) == [:a => 2, :b => 5]
```

Rather than having only global or function scopes, Julia's scoping regions coincide with the following constructs: function bodies, `while` and `for` loops, and blocks introduced with `try`, `catch`, `finally`, `let`, and `type`. Variables introduced in a scope are visible to nested scopes, and shadowing is permitted. We introduce new variables into the current scope with `local` or `const`; function parameters are automatically made part of the function body scope. If you assign to a variable without marking it local, a new local is introduced in the current scope *unless* it has been marked `local` or `global` in an enclosing scope. Marking a variable `global` is required to allow writing to a global variable in an "inner" (nonglobal) scope.

```
a, b, c = 1, 2, 3        # three globals
(function ()
    a = 10               # introduces local and shadows
    global b = 20        # overwrites global
    d = 10c + 10         # new local, reading global
    local e = 5
    while true
        e = 50           # outer e because marked local
        f = 60           # local to while loop!
        break
    end
    @assert (a,b,c,d,e) == (10,20,3,40,50)
    @assert (try f catch e sqrt(100) end) == 10
end)()

@assert (a,b,c) == (1,20,3)
```

All of Julia's types are available at run time, and are fully first-class: you can assign type values to variables and pass them to and return them from functions. There's even a type called `Type`). Types are organized into a hierarchy (though single-inheritance only) with `Any` as the top type. As Julia is designed for scientific computing, rational and complex numeric types are built in. Optionals are "split" into separate types for `Some` and `Nothing`. A part of the hierarchy of Julia's built-in types is shown in Figure 9.1.

```
Any
├─ Number
│  ├─ Real
│  │  ├─ Integer
│  │  │  ├─ Bool
│  │  │  ├─ Signed
│  │  │  │  ├─ Int8
│  │  │  │  ├─ Int16
│  │  │  │  ├─ Int32
│  │  │  │  ├─ Int64
│  │  │  │  ├─ Int128
│  │  │  │  └─ BigInt
│  │  │  └─ Unsigned
│  │  │     ├─ UInt8
│  │  │     ├─ UInt16
│  │  │     ├─ UInt32
│  │  │     ├─ UInt64
│  │  │     └─ UInt128
│  │  ├─ Rational{T<:Integer}
│  │  ├─ AbstractFloat
│  │  │  ├─ Float16
│  │  │  ├─ Float32
│  │  │  ├─ Float64
│  │  │  └─ BigFloat
│  │  └─ AbstractIrrational
│  │     └─ Irrational{sym}
│  └─ Complex{T<:Real}
├─ Symbol
├─ AbstractChar
│  └─ Char
├─ AbstractString
│  └─ String
├─ AbstractPattern
│  └─ Regex
├─ Pair{A, B}
├─ AbstractArray{T,N}
│  ├─ DenseArray{T,N}
│  │  ├─ Array{T,N}
│  │  └─ BitArray{T,N}
│  └─ Range{T}
│     ├─ OrdinalRange{T}
│     └─ AbstractUnitRange{T}
│        └─ UnitRange{T}
├─ AbstractDict{K, V}
│  └─ Dict{K, V}
├─ AbstractSet
│  └─ Set{T}
│  └─ BitSet{T}
├─ Type{T}
│  ├─ DataType
│  ├─ Union
│  ├─ UnionAll
│  └─ Core.TypeofBottom
└─ Expr
```

```
Some{T}
Nothing
Missing
Ref
Task
Module
```

Figure 9.1 A few of the built-in Julia types

Julia does not require type annotations, but you are free to provide them. All assignments of a value to a variable with a type annotation will be type-converted to the annotated type if possible. (In Julia, a value v is converted to type T by calling `convert(T,v)`.)

```julia
x::Int64 = 0x02
y = 0x02
@assert typeof(x) == Int64  # The UInt8 was converted
@assert typeof(y) == UInt8  # No annotations, no coversion

# x = "Oh no" # ERROR: Cannot convert String to Int64
```

Type annotations may increase performance: a compiler can use knowledge that a variable has a given type to create an efficient memory layout and run-time access code.

Every type in Julia is either **abstract** or **concrete**. Every value has a single concrete type; for example, `2.54` has the type `Float64`. Let's explore a few more:

```
@assert typeof(3) == Int64
@assert typeof(0xFA31) == UInt16
@assert typeof(false) == Bool
@assert typeof('\u263a') == Char
@assert typeof("Hello") == String
@assert typeof(BigInt(72)^3897) == BigInt
@assert typeof(:hello) == Symbol
@assert typeof(nothing) == Nothing
@assert typeof(Float64) == DataType
@assert typeof(DataType) == DataType
```

Abstract types do nothing more than *generalize* other types; for example, the abstract type `AbstractFloat` generalizes the four concrete types `Float16`, `Float32`, `Float64`, and `BigFloat`. Calling `typeof` will never produce an abstract type, but you can use `isa` to check "membership" in an abstract type:

```
@assert typeof(88) == Int64
for t in [Int64, Integer, Signed, Real, Number, Any]
    @assert isa(88, t)
end
```

In Julia, *only* abstract types can have subtypes; in fact, in Figure 9.1, all leaf types are concrete and all nonleaf types are abstract.[1] We can use the language itself to explore type relationships, with `supertype` producing *the* supertype of a type, and the `<:` operator, which determines whether one type is a descendant of another.

```
@assert supertype(Int32) == Signed
@assert supertype(Symbol) == Any
@assert supertype(DataType) == Type
@assert Float64 <: Real
@assert BigInt <: Number
```

Let's implement our own types. We'll do so by directly translating our little animals script from earlier chapters. We create an abstract type `Animal`, and three concrete subtypes. As before, all animals `speak` the same way, but each makes its own `sound`:

```
abstract type Animal end
speak(a::Animal) = "$(a.name) says $(sound(a))"

struct Horse <: Animal
    name
end
sound(h::Horse) = "neigh"

struct Cow <: Animal
    name
end
sound(c::Cow) = "moooo"
```

[1] This is *not* the case in every language. Plenty of other languages allow concrete types to have subtypes.

```
struct Sheep <: Animal
    name
end
sound(s::Sheep) = "baaaa"

s = Horse("CJ")
@assert speak(s) == "CJ says neigh"
c = Cow("Bessie")
@assert speak(c) == "Bessie says moooo"
@assert speak(Sheep("Little Lamb")) == "Little Lamb says baaaa"
```

Notice the lack of class constructs with nested `speak` and `sound` definitions. The distinct `sound` implementations are tied to a type via type annotations on the parameter. In Julia's terminology, `sound` is a **generic function** with three distinct **methods**. The function `methods` reports the methods for a generic function, available to us in the REPL:

```
julia> sound
sound (generic function with 3 methods)

julia> methods(sound)
# 3 methods for generic function "sound":
sound(h::Horse)
sound(c::Cow)
sound(s::Sheep)
```

Julia types can be parameterized by other types and certain other values. We can get a feel for these types by studying a few examples:

```
@assert typeof(1:10) == UnitRange{Int64}
@assert typeof(0x5 // 0x22) == Rational{UInt8}
@assert typeof(5 // 34) == Rational{Int64}
@assert typeof(8.75im) == Complex{Float64}
@assert typeof([5,3]) == Array{Int64,1}
@assert typeof([3, "abc"]) == Array{Any, 1}
@assert typeof([]) == Array{Any, 1}
@assert typeof([1 0; 0 1]) == Array{Int64, 2}
@assert typeof(Set(4)) == Set{Int64}
@assert typeof(Set(['3', '$'])) == Set{Char}
```

To distinguish sets of integers from sets of characters, Julia defines a **parametric type** called `Set{T}` whose parameter T can be any type. Type parameters can have restrictions as well. Rational numbers, for example, can have `UInt8` or `Int64` components but not string components, so Julia provides the type `Rational{T<:Integer}`, restricting T to a descendant type of `Integer`. You'll notice several families of parametric types in Julia, including those for sets, arrays, ranges, and complex numbers.

Generally, nonparameterized types are instances of `DataType`; parameterized types are instances of `UnionAll`. `UnionAll` and `DataType` are nonparameterized and instances of `DataType`. They are also subtypes of `Type{T}`, which is parametric and thus an instance of `UnionAll`:

```
@assert typeof(Set) == UnionAll
@assert typeof(Set{String}) == DataType

@assert typeof(DataType) == DataType
@assert typeof(UnionAll) == DataType

@assert typeof(Type) == UnionAll
@assert typeof(Type{UInt128}) == DataType
```

Like TypeScript, Julia features unions and tuples. For example, the type `Union{UInt8, Bool}` contains the values `0`, `1`, ... , `254`, `255`, `false`, `true`; the type `Tuple{UInt8, Bool}` contains the values `(0,false)`, `(0,true)`, `(1,false)`, `(1,true)`, ... `(255,false)`, `(255,true)`.

```
u = Union{UInt8, Bool}          # A new type
@assert typeof(u) == Union
@assert isa(u, Type)            # A union type is a type

@assert isa(0x08, u)            # UInt8 values belong
@assert isa(false, u)           # Boolean values belong
@assert !isa(256, u)            # Not a UInt8, does not belong
```

```
t = Tuple{UInt8, Bool}          # a tuple type
@assert typeof(t) == DataType   # tuple types aren't special
@assert isa(t, Type)            # tuple types are types

@assert isa((0x7A, true), t)    # tuples of the right type belong
@assert !isa((3, "wrong"), t)   # tuples of the wrong type do not
@assert !isa(0x25, t)           # non-tuples do not belong
```

Unions and tuples are not limited to two constituent types:

```
# 3 component types
@assert typeof((4, false, [])) == Tuple{Int64, Bool, Array{Any, 1}}
@assert isa(false, Union{Int64, Bool, Array{Any, 1}})

# 1 component type
@assert typeof((5,)) == Tuple{Int64}
@assert isa(false, Union{Bool})
@assert Bool != Tuple{Bool}           # Because false != (false,)
@assert Bool == Union{Bool}           # Do you see why?

# No component types
@assert typeof(()) == Tuple{}
```

Union types are also called **sum types** and tuples may be known as **product types**. These terms can be remembered by noting that from the 256 values in `UInt8` and 2 in `Bool`, the union (sum) type has $256 + 2 = 258$ values while the tuple (product) type has $256 \times 2 = 512$.

The type `Union{}` has no instances while `Tuple{}` has exactly one, namely `()`, the zero-element tuple. In type theory, the type of no elements is called a `bottom type`. A `top type`, conversely, contains all possible values. Julia's top type is `Any`.

9.3 MULTIDIMENSIONAL ARRAYS

Julia's array type is parameterized not only by the type of its elements but by its **dimension**. So the type of one-dimensional integer arrays differs from the type of two-dimensional integer arrays. Here are some examples of arrays:

$$a = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \end{bmatrix} \quad b = \begin{bmatrix} 5 & 3 & 7 \end{bmatrix} \quad c = \begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 6 \end{bmatrix}$$

Here a is written in Julia as `[10,20,30,40]` and is a *one*-dimensional array, or **vector**, of type `Array{Int64,1}`. Array b is written `[5 3 7]` (note spaces instead of commas), and is a 1×3 (one row, three columns) *two*-dimensional array, or **matrix**, of type `Array{Int64,2}`. Array c is a 2×3 array, also with type `Array{Int64,2}`, and is written `[1 0 9; 0 1 6]`. Higher dimensional arrays have no special syntax, though they can be constructed with comprehensions, e.g., `[i+j+k for i=1:4, j=1:3, k=1:5]`.

In an n-dimensional array, the expression $a[e_1, e_2, ...e_n]$ will select a single element from the array when each of the indexes e_i are integers. However, the index expressions can also be ranges or vectors, allowing some very interesting **slices** to be computed:

```
a = [11 12 13 14; 21 22 23 24; 31 32 33 34]

@assert a[2,3] == 23                      # element in row 2, col 3
@assert a[2,2:4] == [22,23,24]            # row 2, cols 2 through 4
@assert a[1:3,3] == [13; 23; 33]          # rows 1 through 3 of col 3
@assert a[3,1:end] == [31,32,33,34]       # row 3, all elements
@assert a[3,:] == [31,32,33,34]           # row 3, all elements
@assert a[2,[1;3;4]] == [21,23,24]        # row 2, cols 1, 3, 4
@assert a[[1;3],[1;4]] == [11 14; 31 34] # very disjointed subarray
@assert a[7] == 13                        # enumerates by columns
```

Julia has *hundreds* of built-in methods for processing arrays, including `length(A)` for the number of elements in A, `ndims(A)` for the number of dimensions, and `size(A)` for a tuple of its dimensions. Some construct new arrays:

```
@assert zeros(Int64, 3) == [0, 0, 0]
@assert zeros(Int64, 2, 2) == [0 0; 0 0]
@assert ones(Int64, 3, 2) == [1 1; 1 1; 1 1]
@assert fill(5, 1, 4) == [5 5 5 5]
@assert transpose([1 3; 2 4]) == [1 2; 3 4]
@assert [1 3; 2 4]' == [1 2; 3 4]             # postfix ' transposes
@assert fill(10, 3, 1) == [10 10 10]'         # 2d column array!
@assert [2x for x in 1:5] == [2,4,6,8,10]
```

Many of Julia's unary and binary operators can be "vectorized" to `broadcast` to each element of an array (of any dimension).

```
a = [1 2 3; 4 5 6; 7 8 9]
b = [1 0 0; 0 2 0; 9 9 9]

@assert iseven.(a) == [
    false true false; true false true; false true false]
successor(n) = n + 1
@assert successor.(a) == [2 3 4; 5 6 7; 8 9 10]

@assert a .+ b == [2 2 3; 4 7 6; 16 17 18]
@assert a .* b == [1 0 0; 0 10 0; 63 72 81]
@assert a .^ b == [1 1 1; 1 25 1; 40353607 134217728 387420489]
@assert a .* 3 == [3 6 9; 12 15 18; 21 24 27]
@assert a .- 1 == [0 1 2; 3 4 5; 6 7 8]
@assert a ./ 2 == [0.5 1.0 1.5; 2.0 2.5 3.0; 3.5 4.0 4.5]
@assert a .^ 2 == [1 4 9; 16 25 36; 49 64 81]
@assert 1 .- b == [0 1 1; 1 -1 1; -8 -8 -8]

@assert 3 .* a .^ 2 .+ successor.(a) == [
    5 15 31; 53 81 115; 155 201 253]
```

In some cases, operators have been overloaded to perform a broadcast; however, in cases where the operator is conventionally understood to have a different meaning when applied to matrices, the conventional meaning is used. Matrix multiplication is a good example:

```
a = [1 2 3; 4 5 6; 7 8 9]
b = [1 0 0; 0 2 0; 9 9 9]

# Addition has been overloaded to be elementwise
@assert a + b == [2 2 3; 4 7 6; 16 17 18]

# Multiplication, however, is matrix multiplication
@assert a * b == [28 31 27; 58 64 54; 88 97 81]
```

Julia has a vast number of built-in functions and standard library functions useful in the field of linear algebra, such as `tr` (trace), `det` (determinant), and `inv` (inverse), as well as cross product, eigenvalue computation, Hessenberg and singular value decompositions, Givens rotations, transpositions, triangles, diagonalizations, and a large number of factorizations. Dozens more methods are included in additional standard and third-party packages. The following is but of taste of what is available:

```
using LinearAlgebra

a = [2 -1 0; 1 1 1; 0 1 2]
b = [2 1 0; -1 1 -1; 0 -1 2]
u = [1, 2, 3]
v = [1, 0, 0]
```

```
@assert tr(a) == 5
@assert det(a) == 4.0
@assert cross(u, v) == [0, 3, -2]
@assert UnitUpperTriangular(a) == [1 -1 0; 0 1 1; 0 0 1]
@assert UpperHessenberg(a) == [2 -1 0; 1 1 1; 0 1 2]
@assert dot(a, b) == 5

rotate!(u, v, 3, 8)              # ! indicates mutation of u and v
reflect!(v, u, 1, 10)            # ! indicates mutation of u and v
@assert u == [-61, -166, -249]
@assert v == [105, 44, 66]
```

9.4 MULTIPLE DISPATCH

Earlier in this chapter we built a little type hierarchy of animals, horses, sheep, and cows, and showed how Julia's method of dispatching—selecting the proper function or method to invoke among a collection of operations with the same name—was not via a specially marked receiver object, but rather on the type of argument of the callee. Does this idea extend to multiple arguments? For example, could we construct a function `times` such that:

- If x and y are numbers, the function multiplies, e.g., `times(8,3)` ⇒ `24`
- If x is a string and y is an unsigned integer, we have repetition, e.g., `times("ho",3)` ⇒ `"hohoho"`
- If x is a vector and y is a number, we have element-wise multiplication, e.g., `times([1,2,3],3)` ⇒ `[3,6,9]`
- If x is a number and y is a vector, we also have element-wise multiplication, e.g., `times(5,[1,2,3])` ⇒ `[5,10,15]`

Doing so would certainly be better than writing a single function that checks the types of its arguments in a multi-way `if`-statement, an approach that would be difficult to read, mix of disparate concerns, and be impossible of extensibility without modifying existing source code. The ability to dispatch on multiple arguments, known as **multiple dispatch** would also be superior to writing four functions with distinct names. It would also feel rather nice, especially since previous languages with `single dispatch` force us to choose which type a method "belongs to." and this is not always obvious.

Julia, in fact, does allow multiple dispatch. In fact all arguments in a call are considered. So we can write:

```
times(x::Number, y::Number) = x * y
times(x::AbstractArray{T,1}, n::Number) where {T<:Number} = x .* n
times(n::Number, x::AbstractArray{T,1}) where {T<:Number} = x .* n
times(s::AbstractString, n::Unsigned) = repeat(s, n)

@assert times(5, -2.5) == -12.5
@assert times([1.0, 5, 2//3], 3) == [3.0, 15, 2]
@assert times(3, [1.0, 5, 0]) == [3.0, 15, 0]
@assert times("ho", UInt64(3)) == "hohoho"
```

Here `times` is a **generic function** with four **methods**.

The first call in our example dispatches to the first method, because the argument types `(Int64, Float64)` can be passed to the parameter tuple `(Number, Number)`. But notice the definition of the second method. It is tempting to define its first parameter as an `AbstractArray{Number, 1}`, but this will not work! Passing an array of `Float64` objects to an array of numbers fails because, unlike tuples, Julia arrays are not covariant on their element type.[2] Julia's **parametric methods** rescue us here. We introduce a type parameter T and define the method's first parameter as an array of elements whose type is `Number` or any descendant type of `Number`.

9.5 METAPROGRAMMING

We've seen a few introspective capabilities in previous chapters, such as asking a Python or Ruby class for a list of its methods, or asking a JavaScript function for the number of arguments it accepts. Julia has dozens of such functions, including `supertypes`, operator `<:`, `typeof`, and `isa` (seen before), as well as `fieldnames` (of a datatype), `names` (of a module), and `methods` (to list the methods of a generic function). The surprising functions `code_llvm` and `code_native` will show you the intermediate and target assembly language code for a method.

But Julia goes even farther: individual *expressions*, in their *unevaluated* form, can be accessed and manipulated as objects within the program itself. There are several ways to create expression objects:

```
e1 = Meta.parse("x * sqrt(9) - 5")
e2 = :(x * sqrt(9) - 5)
e3 = Expr(:call, :-, Expr(:call, :*, :x, Expr(:call, :sqrt, 9)), 5)

@assert e1 == e2 && e2 == e3 && typeof(e1) == Expr
```

Objects of type `Expr` have three fields: `head`, `args`, and `typ`. The internal structure of the expression in the previous example (without the `typ` fields) is shown in Figure 9.2.

Unevaluated expressions (`Expr` objects) are also known as **quoted** expressions, named after the way we use quotes in prose to distinguish a *mention* of a word from its *use*: we say that ice is frozen water but "ice" has three letters. To make things interesting, Julia allows evaluation during the construction of a quoted expression, via **splicing** (intentionally using the same syntax for string interpolation we saw earlier):

```
y = 10
e = :(x + $(y / 2))      # y/2 evaluated here
@assert e == :(x + 5)
```

Access to unevaluated expressions allows us to rewrite code before evaluation, sometimes to optimize, sometimes to enhance (by adding, for example, timing, logging, or persistence), and sometimes to extend the language with new syntactic forms. While it is possible in theory to write functions that accept expressions as arguments and return expressions, Julia provides **macros** to more elegantly manipulate expressions. Macros are evaluated at

[2] Covariance was discussed in previous chapters on TypeScript and Java, languages that both feature covariant arrays

Figure 9.2 Internal structure of the Julia Expression `:(x*sqrt(9)-5)`

the time the script is parsed, and return expressions that are then compiled into the context in which they are called. For example, the following script

```
x = 2

macro tenTimesX()
    println("hello");    # executed at parse time
    :(x * 10)            # expression returned at parse time
end

function f()
    @tenTimesX()         # replaced with x*10 at parse time
end                      # when called, f executes x=2; x*10
```

prints a message, even though the function f is never called! While parsing the script, Julia executes the macro call within f (printing a message). The returned expression, `:(x * 10)`, can then be compiled into the body of f. This rewrites the function to:

```
function f()
  x = 2
  x * 10
end
```

Julia takes care that writable variables in a macro's result expression do not modify variables of the same name in the caller's context. In technical terms, Julia macros are **hygienic**. In the following example, the macro's own local y does *not* overwrite the caller's y:

```
x = 2

macro tenTimesX()
    :(y = 10; x * y)
end

(function ()
    y = 3                       # Will this y be changed?
    z = @tenTimesX()            # Changed to 10 perhaps?
    @assert y == 3 && z == 20   # No, y will not change
end)()
```

We can use the function `macroexpand` in the REPL to see why no change occurs:

```
julia> macro tenTimesX()
         :(y = 10; x * y)
       end

julia> macroexpand(:((x,y) = (3,2); @tenTimesX()))
quote
    (x,y) = (3,2)
    begin
        #1#y = 10
        x * #1#y
    end
end
```

During macroexpansion, a new symbol, `#1#y`, was used in place of the variable y, avoiding a name clash. Julia will generate new names for variables that are (1) explicitly marked local, (2) assigned to but not marked global, or (3) used a function argument. Other variables are resolved in the context of the macro definition, so the macro's use of well-known globals will not be overwritten by names the caller might have inadvertently redefined.

There are times, however, when we *do* wish a macro to manipulate variables from the caller's context. This is convenient when making use of macros' ability to "extend" the language with what appear to be new syntactic forms—a benefit of macros' ability to take arguments as unevaluated expressions, which, for convenience, are never quoted. Let's create and test an `unless` statement. We want the body of our generated expression to evaluate variables in the caller's context, so we suppress the hygienic generation of new variables with `esc`:

```
macro unless(condition, body)
    :(if !$condition
        $(esc(body))
    end)
end

x, y = 0, 1
@unless (x == 5) begin
    y = 3                   # should execute
end
```

```
@unless (x == 0) begin
    y = 5                    # should not execute
end

@assert y == 3
```

This macro rewrites the code fragment

`@unless (x < 5) begin y = 2 end`

into

`if !(x < 5); y = 2; end`

Without `esc` the macro would have rewritten our code fragment into:

`if !(x < 5); #2#y = 2; end`

The `esc` function explicitly bypasses hygiene: Escaped expressions are compiled directly into the output without any new symbols being generated.

9.6 PARALLEL COMPUTING

To be successful as a language for scientific computing, a language must support efficient execution over large datasets; it should take advantage of processors with multiple cores or multiple computers in a cluster. Running related computations on multiple processing units simultaneously is called **parallel programming**. When multiple independent processes communicate exclusively via message passing (as opposed to shared memory), we have **distributed computing**. Julia provides both primitives and high-level constructs for computing across multiple independent processes.

A Julia **remote call** invokes a function asynchronously on a specific process, returning immediately with a **remote reference**, or **future**. Calling `fetch` on the remote reference will wait for the function to complete on the remote process, and produce the function's result. The following script creates three new processes,[3] and instructs process #4 to factor a large number:[4]

```
using Distributed
addprocs(3)
@assert nprocs() == 4

@everywhere import Primes
using Primes

future = remotecall(factor, 2, 21883298135690819)
@assert isa(future, Future)
```

[3]Calling `addprocs` with an integer argument creates processors on the local machine to take advantage of multiple cores. There are other variants of `addprocs` to add processors on remote machines either via user-host-port strings or cluster managers; we will not cover these alternatives in this text.

[4]The `factor` function resides in the package `Primes`. You must first install this package by executing `Pkg.add("Primes")` in an interactive shell, or `julia -e 'Pkg.add("Primes")'` from the command line.

```
factors = fetch(future)
@assert collect(factors) == [93234719=>1,234711901=>1]
```

The `@spawn` macro is often used in place of `remotecall`; it not only takes advantage of a clearer syntax (since macros take unevaluated expressions as arguments), but it chooses a process for you:

```
using Distributed
addprocs(3)
@everywhere import Primes

ref = @spawn Primes.factor(21883298135690819)
factors = fetch(ref)
@assert collect(factors) == [93234719=>1, 234711901=>1]
```

Between the remote call and the fetching of the result, the calling process is free to do other work.

A common example of parallel programming across multiple cores computes an approximation of π by generating random points in the square $(0,0)...(1,1)$. Counting the number of random points within the inscribed arc (the points (x, y) for which $x^2+y^2 \leq 1$ in Figure 9.3) divided by the total number of points within the square approximates the value $\frac{\pi}{4}$. Let's generate a million random points in parallel on three cores:

Figure 9.3 Approximation of $\frac{\pi}{4}$

```
using Distributed
addprocs(3)

@everywhere function approximate_pi(trials::Int64)
    hits = 0
    for i in 1:trials
        if sum(rand(2).^2) < 1
            hits += 1
        end
    end
```

```
    hits/trials * 4
end

references = [@spawn approximate_pi(1000000) for i=1:3]
approximation = sum([fetch(ref) for ref in references])/3
@assert 3.1 < approximation < 3.2
```

Here the `@everywhere` macro ensures our function is available to each process. We spawn three executions on each of the three cores, then fetch and average the results. But we can do a bit better, letting Julia do most of the work with its `@parallel` macro:

```
using Distributed
addprocs(3)

trials = 4000000
hits = @distributed (+) for i = 1:trials
    Int(sum(rand(2).^2) < 1)
end

approximation = hits/trials * 4
@assert(3.1 < approximation < 3.2)
```

The parallel for-loop partitions the range over all of the available workers (all processes except the "main" process) and applies the reduction function (here we've used `+`) locally within each process. The calling process applies the function again to each process's result.

In addition to support for parallel and distributed execution, Julia offers other concurrent features that we won't cover here, including coroutines and channels. We've seen coroutines before, and we'll visit channels later in the book when we visit Rust and Go.

JULIA WRAP UP

In this chapter, we were introduced to Julia. We learned that:

- Julia was designed for applications in data science, numerical analysis, and scientific computing.
- Julia features inclusive ranges and array bounds starting at 1, rather than 0.
- In function declarations, positional parameters come first; keyword parameters, if any, are separated from the positional parameters with a semicolon. Positional parameters *may* have defaults; keyword arguments *must* have defaults.
- Scopes coincide with function bodies, `while` and `for` loops, and blocks introduced with `try`, `catch`, `finally`, `let`, and `type`. The construct `begin`...`end` in particular, is not a scope. Variables introduced in a comprehension are scoped to each iteration.
- All Julia types reside in a single supertype-subtype hierarchy with `Any` at the top. Each type can have at most one supertype. Types can be abstract or concrete. Only abstract types can have subtypes. Types are first-class objects, and the type system is easy to introspect via methods such as `super`, `subtypes`, `isa`, `typeof`, and the operator `<:`.

- Types can be parameterized. Parameterized collection types in Julia are invariant.
- Julia features sum (`Union`) and product (`Tuple`) types. Tuples are covariant.
- Type annotations provide the foundation for multiple-dispatch, in which methods are selected based on the types of all of a function's arguments. Annotations are also frequently employed to improve performance.
- Julia expressions can be manipulated and modified before evaluation, enabling sophisticated forms of metaprogramming.
- Macros are executed when code is parsed, that is, before it is run. Macros return expressions which are compiled into the context of the call. Macros take their arguments in unevaluated form, without the need to explicitly quote them.
- Julia provides many features for parallel and concurrent programming. Processes can be created to take advantage of multiple cores on a processor or multiple processors in a cluster.
- The low-level constructs for parallel programming are remote calls and remote references. Various macros, such as `@spawn`, `@fetch`, and `@parallel` provide a higher-level programming model.

For further study in Julia, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Julia, see the Julia page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/julia.

EXERCISES

Now it's your turn. Continue exploring Julia with the activities and research questions below, and feel free to branch out on your own.

9.1 Find the Julia home page. Download the latest distribution and experiment with the REPL.

9.2 Browse the standard library documentation at http://julia-demo.readthedocs.org/en/latest/stdlib/base.html.

9.3 Research and explain, via interesting examples, the differences between `==`, `===`, and `is`. Does the `==` operator perform a shallow or deep equality test?

9.4 What operator does Julia use for string concatenation?

9.5 Compare and contrast Python and Julia's approaches to positional, optional, and keyword parameters, and their corresponding arguments. Consider questions for each language such as: Can positional and keywords be optional? Can positional and keyword parameters and arguments appear in any order?

9.6 Define a function called `fib` to compute the nth Fibonacci number, using tail recursion and optional parameters. Calling `fib(7)` should return 13, via the following calls `fib(7)` ⇒ `fib(6,0,1)` ⇒ `fib(5,1,1)` ⇒ `fib(4,1,2)` ⇒ `fib(3,2,3)` ⇒ `fib(2,3,5)` ⇒ `fib(1,5,8)` ⇒ `fib(0,8,13)` ⇒ 13.

9.7 If you were to browse Julia source code in both the online documentation and in online public repositories, you may notice that the `local` keyword is somewhat rare.

Why do you think this is? In what situations would you consider the `local` keyword to be necessary?

9.8 Do the Julia expressions `[1 2]` and `[1 2 3]` have the same type? If so, what is that type?

9.9 Given the two expressions `[1 [2 3]]` and `[1;[2;3]]`, (a) what are the types of each? (b) draw pictures of the arrays (or matrices) that the expressions represent.

9.10 The expression `isa(e,Irrational) && isa(`π`,Irrational)` evaulates to true, yet `typeof(e)==typeof(`π`)` evaulates to false. Why, exactly?

9.11 Why is the Julia type `Type{T}` a parameterized type? What exactly, is the type `Type{Int64}`? When, if ever, would the types `Type` are `Type{Type}` be useful?

9.12 The type `Any` is sometimes called a "top" type. Find out what this means. Does Julia have a "bottom" type? If so, what is it?

9.13 Evaluate each of the following expressions and explain each result:

```
a. isa([0, 1], Array)
b. isa([0, 1], Array{Integer})
c. isa([0, 1], Array{Int64})
d. isa([0, 1], Array{Int64,1})
e. isa([0, 1], Array{Int64,2})
```

9.14 What is the type of the expression `[i+j+k/2 for i=1:4, j=1:3, k=1:5]`? How does Julia infer this type?

9.15 In your version of Julia, how many distinct methods exist for the following generic functions: `+`, `convert`, `length`, and `&`.

9.16 Does Julia use contravariance anywhere? If so, where? If not, find one other language in which it is used and give an example.

9.17 Research advantages and disadvantages of multiple dispatch over single dispatch. Why did Julia adopt multiple dispatch?

9.18 Are type annotations checked before the program is run or during? Write a program that proves your answer.

9.19 We covered three techniques for denoting `Expr` objects in this chapter. There is a fourth way, namely the `quote`...`end` construct. How does it differ from the other three? As it is more complicated, when is it ever advantageous to use it?

9.20 Are macros necessary? If you answer yes, give an example of a (useful) macro which cannot be easily simulated by regular "run-time" code that may, of course, use unevaluated expression objects. If you answer no, show how to duplicate the functionality of each of the macros defined in this chapter.

9.21 The variables generated during Julia's hygienic macro expressions start with the hash (`#`) character. Why is this not problematic, given that the hash character begins comments in Julia source code?

9.22 Design a macro for an "`until` statement," similar to the "`unless` statement" presented in this chapter.

9.23 We did not cover coroutines (called tasks in Julia) in this chapter, so here is your chance to learn about them. Create two tasks, one that produces a million random integers in the range 0..99, and another that consumes these values by histogramming them into buckets for the values 0–9, 10–19, 20–29, and so on.

9.24 What are the possible status values of a Julia task? How do these compare to the statuses of Lua's coroutines?

9.25 In the REPL, the return value of a remote call will display something like `RemoteRef{Channel{Any}}(4,1,13)`. Explain exactly what is being presented here.

9.26 If you are on a Unix or Unix-like system with multiple cores, run a π-approximation script from this chapter under the Unix `time` command. You should notice that the `real` time is less than the `user` time. Experiment with different values of arguments to `addprocs`. Which value on your system gives the most "speedup"?

9.27 Research Julia's shared arrays and create an example that illustrates their usefulness.

CHAPTER 10

Clojure

Clojure is a "dynamic ... general-purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming." [45]

First appeared	2007
Creator	Rich Hickey
Recognized for	Macros, Lisp-ness, Concurrency support, Java interoperability
Notable uses	Data mining, Artificial intelligence
Tags	Functional, Dynamic, Concurrent, Homoiconic
Six words or less	A Lisp for the JVM

In the 1950s John McCarthy created Lisp. In doing so, he was said to have "[done] for computing something like what Euclid [had done] for geometry," [37] written the "Maxwell's Equations of software," [27] and given the world "the greatest single programming language ever designed." [105] Lisp, remarked Edsger Dijkstra, has "assisted a number of our most gifted fellow humans in thinking previously impossible thoughts." [23]

This chapter introduces Clojure. Clojure is a Lisp—a modern Lisp. It has a richer set of data structures than the original Lisp. It does a better job than most (if not all) other Lisp dialects in isolating side effects and implementing concurrent features. Clojure interoperates with Java, so your Clojure program has access to the entire Java platform. Your Clojure code can call Java libraries. Your Java code can execute Clojure-written code.

After presenting our usual three introductory scripts, we will cover the basic data types of Clojure (there are quite a few of these) and devote some time to **persistent data structures**, and the use of **transients** to isolate mutable operations. We'll then look at Clojure's various approaches to concurrency (one of the language's great strengths) visiting refs, agents, and software transactional memory.

We'll close our tour with a look at the macro system, which takes advantage of Clojure's representation of code as data, and is considered by many to be the essential ingredient of any Lisp dialect.

10.1 HELLO CLOJURE

We begin with our usual three programs. Let's generate right triangle measurements:

```
(doseq [c (range 1 41) b (range 1 c) a (range 1 b)]
  (if (= (+ (* b b) (* a a)) (* c c))
    (printf "%d, %d, %d\n" a b c)))
```

Clojure's ranges are inclusive at the lower bound and exclusive at the upper bound. Like Julia, we can write nested loops with a flattened syntax. The equality operator is (perhaps refreshingly) `=`, not `==` or `===`. Clojure's `printf` works like Java's `printf`.

How about that syntax! We can't help but notice the use of parentheses (and brackets) for structure. This design choice has some advantages over the control structure syntaxes we've seen up to this point: it is more compact than the terminal `end` of Lua, Ruby, and Julia; avoids the curly-brace style debates plaguing JavaScript, Java, and friends; and does not impose any particular indentation requirements on the programmer, as does Python. Clojure (like all Lisps) takes the world of nested parentheticals even further, applying it regularly to all code—control flow and expressions—throughout the entire language, even to the point of placing arithmetic operators in an unfamiliar prefix position. So where we might traditionally see:

```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

we have, in Clojure:

```
(/ (+ (- b) (Math/sqrt (- (* b b) (* 4 a c)))) (* 2 a))
```

The Clojure text is a direct translation from the abstract syntax tree (see Figure 10.1). This syntax simplifies a couple aspects of language design: operators and named functions are treated exactly the same, and there is no need to define operator precedence or associativity.

Figure 10.1 An abstract syntax tree

Now let's turn to our second teaser script: Heap's algorithm. Although this algorithm is defined in terms of successive mutations on a single, global array, we will embrace the Clojure culture where everything (within reason) is immutable:

```
(defn swap [a i j]
  (assoc a j (a i) i (a j)))

(defn print-permutations [v n]
  (if (zero? n)
    (do (println (clojure.string/join "\t" v)) v)
    (loop [i 0 a v]
      (if (< i n)
        (let [a (print-permutations a (dec n))]
          (recur (inc i) (swap a (if (even? n) 0 i) n)))
        (print-permutations a (dec n))))))

(print-permutations
  (vec *command-line-args*)
  (dec (count *command-line-args*)))
```

We may take a slight performance hit: the invocation `(swap a i j)` returns a *new* vector like a but with the values at indexes i and j swapped. We'll talk about efficiency later. For now, let's focus instead on the many interesting aspects of the script:

- Functions are defined with `defn` followed by the function name, then a **vector** containing the parameter names, then the function body.
- Clojure uses the syntax `(f x y)` for function calls (such as `inc`, `dec`, and `count`), **special forms** (such as `if`, `let`, and `loop`), and macros (which we'll see later). Clojure evaluates arguments prior to a function call, but does not do so for macros and special forms. You will need to memorize which symbols refer to functions and which do not; there is no marker, such as the `@` in Julia, to distinguish the various kinds of entities.
- Several Clojure special forms, including `let`, `loop`, and `doseq`, create bindings local to the form. For example `(let [x 5 y 2] (* x y))` evaluates to 10, with x and y bound only within the form.
- The special form `(if` x y z`)` evaluates x and, if truthy, evaluates and returns y; otherwise it evaluates and returns z.
- The special form `(do` $e_1 ... e_n$`)` evaluates a sequence of expressions, presumably for their side effects, and returns the result of evaluating the last expression. We used `do` above in order to print a permutation but also return it from the current invocation in a chain of recursive function calls. Clojure tends to favor a style of programming where mutating-while-looping is replaced with a chain of function calls, each returning the next computation state.
- The `loop` special form uses `recur` to advance the loop, rebinding its locals to the "arguments" sent by the `recur` form. Our loop starts with i bound to 0 and a bound to our initial vector. Each recurrence of the loop binds i to the next index position $(i + 1)$ and a to the next permutation. The loop ends at $i = n$, since no `recur` form is evaluated at this point.[1]

[1] The `loop`/`recur` form takes the place of tail-recursive functions in similar languages, and exists because Clojure does not wish to trust that its target virtual machine is smart enough to avoid the pile of memory-consuming frames generated during a chain of recursive calls.

Our third teaser is once again the data science application reading lines from standard input and producing the desired reoprt:

```
(defrecord Player [team name games ppg])

(defn from-line [[team name games points]]
  (let [points (parse-double points)
        games (parse-long games)]
    (->Player team name games (/ points games))))

(defn player-report [player]
  (format "%-22s%-4s%8.2f"
          (:name player) (:team player) (:ppg player)))

(dorun
  (->>
    (line-seq (java.io.BufferedReader. *in*))
    (map #(clojure.string/split % #","))
    (map from-line)
    (filter #(>= (:games %) 15))
    (sort-by :ppg >)
    (take 10)
    (map player-report)
    (map println)))
```

This is rather dense script, but instructive to study, as it manages to pack in a number of interesting aspects of the language. There are basically two parts: (1) the definition of a "type" for players, and (2) the loading, transformation, and reporting of the data.

There are few ways in Clojure to create custom data types; in this script the `record` is best. Using `defrecord` we get a type for objects that have autogenerated keyword accessors for fields (e.g., `:name`, `:team`, etc.) value-based equality, and more. In our case, associated functions for creating a player from a string and printing a player in the desired format are written as regular functions. The parameter to `from-line` illustrates destructuring, and the body illustrates multiple bindings in a `let`-form as well as the use of the autogenerated player constructor (`->Player`).

The main portion of the script executes, via `dorun`, a *threaded* sequence of forms. The "thread-last" operator `->>` evaluates its first form and passes the result as the last "argument" of the next form, and so on. The sequence begins with generating, from the standard input stream `*in*`, a **lazy** sequence of lines with `line-seq`. A lazy sequence is one that produces its elements only when needed, so we need not worry about slurping a large file into memory. The sequence is passed to the function which maps the "splitting of a string on the comma character" to each line. This resulting sequence is processed by a mapper creating player objects, then filtered to preserve only players appearing in 15 or more games, then sorted by points-per-game descending, of which the first ten are taken, and finally each line is formatted and printed.

We should mention that because the sequence is lazy, nothing happens until the sequence itself is somehow "consumed." In other words, all of the map and filter and sort and take

operations don't actually *do* anything on their own—they in effect build up a set of instructions to eventually be applied. The `dorun` form causes these instructions to execute.

10.2 THE BASICS

Understanding Clojure begins with understanding its execution model, defined in terms of the **reader** and the **evaluator**. The reader reads source code from a stream, parsing the text into **forms**. Each form will subsequently be passed to the evaluator for evaluation. Forms known to the reader, and the kinds of objects they represent, include:

- **Integers**: `98`, `98325679988888112356249870 03`, `0x3F`. All bases between 2 and 36 are supported: we can write 512 as `30rh2` or `7r1331`. The integer types are `Byte`, `Short`, `Integer`, and `Long` from the package `java.lang`; and `BigInt` from `clojure.lang`. Literals have the type `Long` if in the range $-2^{-63}...2^{63}-1$ and `BigInt` otherwise. You can force the type BigInt with the suffix `N` (e.g., `200N`). The functions `byte`, `short`, `int`, `long`, and `bigint` create numbers of a desired type.
- **Doubles**: `2.54`, `3E5`, `5.0`, `-99.73`. The double types are `Float` and `Double` from `java.lang`. Create objects of a desired type with the `float` and `double` functions.
- **Ratios**: `22/7`, `3/10`. These have the type `clojure.lang.Ratio`.
- **BigDecimals**: `3M`, `78.2E900M`. These values have type `java.math.BigDecimal`. BigDecimals can be created from other numbers with the `bigdec` function.
- **Strings**: `"Hello"`. Enclosed in double quotes and may span multiple lines.
- **Characters**: `\z`, `\newline`, `\tab`, `\u263A`. Identical to Java characters in meaning, if not in syntax.
- **Booleans**: `true`, `false`.
- **Nil**: Use the value `nil` as you would `null` in Java. Note, also, as in Lua and Ruby, `nil` and `false` are the only falsy values.
- **Keywords**: `:dog`, `:rat`. A keyword in Clojure evaluates to itself. Always.
- **Symbols**: `factorial`, `+`, `even?`. Symbols refer to global variables, function parameters, and similar entities.
- **Lists**: `(:a 2 (6 8 \q) "ok" 'then)`. Whitespace-separated forms, delimited by parentheses.
- **Vectors**: `[2 5.5 \a [(list 1 2) 3]]`. Whitespace-delimited forms, delimited by square brackets.
- **Sets**: `#{6.3 [8 :k] "hello"}`.
- **Maps**: `{:jo "Amman" :ws "Apia" :ee "Tallinn"}`. Key-value pairs. The keys needn't be keywords, though they tend to be in practice.
- **Functions**: `#(+ (* 2 %1 %2)`. Arguments are `%1`, `%2`, and so on.

The reader also handles comments, metadata maps, syntax quoting and unquoting, anonymous functions, regular expressions, etc. It also deals with syntax sugaring; for example, `@x` desugars to `(deref x)` and `'x` to `(quote x)`. We'll see several of these forms and constructs later in the chapter.

Simple forms other than symbols evaluate to themselves, while vector, set, and map forms

evaluate their elements and produce collection objects. Symbols get resolved to a value depending on what the symbol currently refers to. Evaluation of the list $(f_1, f_2, ..., f_n)$ first evaluates the form f_1. If f_1 evaluates to a macro or special form, it operates on the *unevaluated* remaining forms. Otherwise, Clojure casts f_1 to a function and applies it to the *evaluations* of $f_2, ..., f_n$. Quoting suppresses the symbol and list evaluations: `'x` evaluates to the symbol itself, and `'(a b c)` evaluates to a three-element list rather than applying a to b and c.

Clojure uses dynamic typing, with a number of symbols predefined to stand for types:

```
(use 'clojure.test)
(is (= (type 3) Long))
(is (= (type 5.0) Double))
(is (= (type 4/7) clojure.lang.Ratio))
(is (= (type 5N) clojure.lang.BigInt))
(is (= (type 5.88M) BigDecimal))
(is (= (type "Hello
  World") String))
(is (= (type \z) Character))
(is (= (type false) Boolean))
(is (= (type nil) nil))
(is (= (type 'dog) clojure.lang.Symbol))
(is (= (type :dog) clojure.lang.Keyword))
(is (= (type Long) java.lang.Class))
(is (= (type java.lang.Class) java.lang.Class))
```

Because symbols need not contain only alphanumeric characters, addition, subtraction, multiplication and friends are not special operators, but rather regular functions. Thanks to Clojure's syntax, they each accept a variable number of arguments:

```
(use 'clojure.test)
(is (= (+ 8 7 3 2 8) 28))        ; sum a lot of numbers
(is (= (+ 8) 8))                 ; yes we can sum just one
(is (= (+) 0))                   ; 0 is the identity for +
(is (< 3 5 7 9 12))              ; a convenient sort test
(is (not (<= 8 9 10 22 20 8)))
(is (= (max 8 7 14 2 8) 14))
```

The division operator produces longs, doubles, or ratios intelligently:

```
(use 'clojure.test)
(is (= (type (/ 3 9)) clojure.lang.Ratio))
(is (= (type (/ 8 4)) Long))
(is (= (type (/ 3.0 9)) Double))
```

The functions `+`, `-`, `*`, `inc`, and `dec` on longs may overflow, throwing an `ArithmeticException`; use `+'`, `-'`, `*'`, `inc'`, and `dec'` to auto-promote to `BigInt`:

```
(use 'clojure.test)
(let [x (long 1E15) max-long Long/MAX_VALUE]
```

```
  (is (thrown? ArithmeticException (* x x)))
  (is (= (*' x x) (bigint 1E30)))
  (is (thrown? ArithmeticException (inc max-long)))
  (is (= (inc' max-long) (+' 1 max-long))))
```

Collections are immutable and implement equality with value—not reference—semantics. Many common operations apply to all collections, such as `count`, `empty?`, `not-empty`, and `conj`, short for **conjoin**. Conjoining a collection *c* and an element *e* returns a *new* collection with all the elements *c* together with *e*.

```
(use 'clojure.test)
(let [a-list '(1 2 3) a-vec [1 2 3] a-set #{1 2 3} a-map {1 2 3 4}]
  (is (= (empty? a-set)) false)
  (is (not-empty a-vec))
  (is (= (map count [a-list a-vec a-set]) [3 3 3]))
  (is (= (count a-map) 2))
  (is (= (conj a-list 4) '(4 1 2 3)))        ; list: conj to front
  (is (= (conj a-vec 4) [1 2 3 4]))          ; vec: conj adds to end
  (is (= (conj a-set 4) #{3 2 1 4}))         ; order is irrelevant
  (is (= (conj a-map [5 6]) {1 2 5 6 3 4})) ; adds a new pair
```

These functions, and dozens more like them, operate by internally turning the collection into a **sequence**, a logical list of elements, usually (but not always) lazy. (Recall that a lazy sequence produces its elements on demand, while a nonlazy sequence stores each element in memory). You can produce a sequence directly from a collection *c* by evaluating `(seq` *c*`)`. Let's look at a few more sequence functions:

```
(use 'clojure.test)
(let [a '(10 20 30) b #{5 15 25 35}]
  (is (some #(> % 30) b))
  (is (some (partial > 30) b))
  (is (count (filter (partial >= 20) a)) 2)
  (is (reduce * a) 6000)
  (is (= (map inc a) (seq [11 21 31])))
  (is (take 2 a) (seq [10 20]))
  (is (= (interleave a (sort b)) (seq '(10 5 20 15 30 25)))))
```

The previous script illustrated function values. The expression `#(> % 30)` evaluates to a function of a single argument and returns whether its value is greater than 30. As functions are first-class values, we can associate a symbol with them through a **local binding**.

We can also write function values with the special form `fn`:

```
(use 'clojure.test)
(let
  [average (fn [x y] (/ (+ x y) 2))]
  (is (= (average 9 5) 7)))
(let
  [average #(/ (+ %1 %2) 2)]
  (is (= (average 9 5) 7)))
```

Functions, by the way, support a variable number of arguments. The parameter after the `&` becomes a sequence containing the additional arguments in the call:

```
(use 'clojure.test)
(let [show (fn [x y & z] [x y z])]
  (is (= (show 1 2) [1 2 nil]))
  (is (= (show 1 2 3) [1 2 '(3)]))
  (is (= (show 1 2 3 4) [1 2 '(3 4)])))
```

This brief overview should give a feel for what Clojure code looks like. Now we're ready to move on to some very interesting, and even sophisticated, corners of the language. We'll begin with namespaces.

10.3 NAMESPACES

Namespaces contain mappings of symbols to **vars** and **classes**. Vars refer to mutable storage locations. The special form `def` stores, or **interns**, a var into a namespace. We'll create a second var using `defn`, which gives us a pleasant syntax for interning a var containing a function value (using `def` behind the scenes).

```
(ns library)            ; Going to work in this ns

(def x 5)               ; Map symbol x to a Var
(defn average [x y]     ; Map symbol average to a Var
  (/ (+ x y) 2))        ; Local x and y are not Vars
(def x 8)               ; Vars are mutable!

(ns application)        ; Switch to new namespace

(use 'clojure.test)     ; No need to say clojure.test/is
(is (= library/x 8))
(is (= (library/average 10 20) 15))

(use 'library)          ; No need to say library/x now
(is (= x 8))            ; this is x from library
```

We begin this script by entering a (presumably new) namespace named `library`, and interning two vars. We then modify the contents of (mutate) the var. The symbol `x` *remains mapped to the same var*; only the contents of the var changes. See Figure 10.2. We then switch to a new namespace named `application`; to access the objects in the other namespace we use a qualified symbol. After invoking `use`, however, the symbol mappings within the library namespace become available without qualification.

Vars are Clojure objects. We denote the var object mapped to the symbol x by `#'x`, or equivalently, `(var x)`. Evaluating the symbol produces the value held in the mapped var. To get this value from the var object itself, we use `deref`, or the shorthand notation with the `@` prefix. Study the following script carefully:

Figure 10.2 Symbols, vars, and values

```
(use 'clojure.test)
(def x 100)                          ; create a Var, map x to it
(def y #'x)                          ; the actual Var that x maps to
(is (= (type y) clojure.lang.Var))   ; the Var is an object with a type
(is (= (deref y) 100))               ; deref is the value in the Var
(is (= @y 100))                      ; @y is shorthand for (deref y)
(def x 200)                          ; Update the var through the symbol
(is (= (deref y) 200))               ; See the change directly
(is (identical? #'x (var x)))        ; #'x is shorthand for (var x)
```

Prior to encountering vars, all of the Clojure objects we've seen, including collections, were immutable. Immutability is certainly the norm in Clojure; the *only* mutable objects are vars, refs, agents, and Java object fields (Clojure "interops" with Java quite nicely). Immutability makes threaded programming easier by removing race conditions and other concurrent problems arising from shared state. However, threads do need to cooperate via the changing state of a system. Fortunately, vars, refs, and agents have been carefully designed to manage this cooperation quite well. We'll get to refs and agents shortly, but first, we have an important question to answer. If collections are immutable, with add, remove, and update operations generating new collection objects, how can such operations possibly be efficient?

10.4 PERSISTENT DATA STRUCTURES

Consider a 1000-element list, referred to by the symbol a. The expression `(conj a x)` creates a brand new 1001-element list, whose first element is the value of x. Should this new list contain copies of each of the elements of a, or can the two lists share structure, as in Figure 10.3? While conjoining by sharing is as efficient as possible, the downside is that changes to one list would (surprisingly) affect the other. However, as long as the lists are immutable, sharing works! No surprises are even possible.

Figure 10.3 Conjoining to a Persistent List

We can verify that Clojure does indeed share:

```
(def a (list (repeat 1000 :dog)))   ; 1000-dog list
(def b (conj a :cat))               ; new list, cat at front
```

```
(use 'clojure.test)
(is (identical? a (rest b)))
(is (= (type a) clojure.lang.PersistentList))
```

The expression `(rest b)` refers to the sequence of all elements except the first element of b, which our script tells us is indeed `identical` to (the same internal object as) a. Since the list a remains, and was not destroyed by modification, and still remains accessible, we call it a **persistent** list.

Adding to the front of a list poses no efficiency concerns, but how can we add to the rear, as vector conjoining requires? Copying each of the vector elements appears to be necessary, but we can get by with a logarithmic number of copies if we index the vector elements using a tree structure. In Figure 10.4 our vector v has the value `[1 2 3 4 5 6]`. Elements reside in the leaf nodes from left to right. To compute `(conj v 7)`, we walk down the path to the node at the end of the vector, copying internal nodes as we go, sharing structure where we can. This technique of **path copying** applies also to functions like `assoc`, which we encountered in our anagrams example at the beginning of the chapter.

Figure 10.4 Conjoining to a Persistent Vector

We've drastically simplified things here. The actual persistent vector implementation used by Clojure is far more sophisticated: internal tree nodes have up to 32 children, using bit vectors to manage the tree structure. We're not concerned in this book with specific implementation details; our goal is simply to point out the existence of sublinear algorithms, so that language designers may embrace immutability and developers may gain a sense of the cost of using these objects. See Okasaki [80] for a comprehensive treatment of persistent data structures.

10.5 TRANSIENTS

Persistent structures pay for the advantages of immutability (thread safety, sharing) by sacrificing performance. However, while *constructing* an object—before it ever needs to be seen by the rest of the program—you can take advantage of internal, mutable operations that can speed things up significantly. Clojure's **transient data structures** operate this way; they are built up in an efficient, mutable fashion, then turned into a persistent structure when ready to use. The following example constructs a (nonlazy) vector of Fibonacci numbers:

```
(defn fib [n]
  (loop [a 0 b 1 result (transient [0])]
    (if (> b n)
      (persistent! result)
      (recur b (+' a b) (conj! result b)))))

(use 'clojure.test)
(is (= (fib 0) [0]))
(is (= (fib 1) [0 1 1]))
(is (= (fib 5) [0 1 1 2 3 5]))
(is (= (fib 144) [0 1 1 2 3 5 8 13 21 34 55 89 144]))
(is (= (fib 200) [0 1 1 2 3 5 8 13 21 34 55 89 144]))
(println (fib (bigint 1E200)))
```

We begin our loop binding a to 0, b to 1, and *result* to `(transient [0])`, a transient vector containing the single element 0. We add new values via `conj!`, not `conj`; the bang (`!`) alerts the reader that something unusual is taking place, in this case mutation. After the vector has been filled, we create, and return, a new persistent vector.

Clojure will not prevent you from misusing transients by accessing them from multiple threads at the same time. Mutability demands responsibility in a multithreaded environment. However, Clojure offers several facilities for making concurrency safe. We'll turn to these now.

10.6 SOFTWARE TRANSACTIONAL MEMORY

When running on the JVM platform, Clojure threads are Java threads. The translation of Java's `new Thread(r)` for runnable r to Clojure is `(Thread. r)` (the trailing dot is part of the name of the symbol denoting the constructor). And the Java expression `t.start()` becomes `(.start t)` in Clojure—method symbols have a leading dot. The following script creates 10 threads, each of which prints a list of asterisks, in a completely unsynchronized, free-for-all fashion:

```
(dotimes [i 10]
  (.start (Thread. (fn [] (println (repeat i "*"))))))
```

This script's output may vary each time you run it; one output was:

```
((((((**  * ** *  *(* ) **)*

((* **  ** *  * **)
(*)*
 * ** )*
 **))
* * **  ***  * * )

* * * ** )*
 * *)
```

We can apply a simple fix, using the `locking` macro to print only while the current thread holds the intrinsic lock for the standard output object:

```
(dotimes [i 10]
  (.start (Thread. (fn []
    (locking *out* (println (repeat i "*")))))))
```

This produces, on one run:

```
()
(* * * * * * * * *)
(* * * * * * * *)
(* * * * * * *)
(* * * * * *)
(* * * * *)
(* * * *)
(* * *)
(* *)
(*)
```

But Clojure offers sophisticated concurrency support that makes explicit use of threads and locks very rare. In this section, we'll explore **software transactional memory**, or STM, and cover agents in the next section. Transactional memory is implemented in Clojure with **refs**, mutable storage locations that can only be updated inside a **transaction**, specified in Clojure with `dosync`. Transactions prevent concurrent updates of references from other threads.

STM solves one of the classic problems in concurrency—transferring money between accounts—quite nicely. First, let's note that transferring, say, 100 units of currency from a savings account to a checking account generally requires at least 6 steps:

1. Read the balance of the savings account (suppose it is 10000), into a machine register
2. Decrement the register by 100
3. Write the decremented value (99900) back to memory
4. Read the balance of the checking account (suppose it is 500), into a machine register
5. Increment the register by 100
6. Write the incremented value (600) back to memory

If another thread were able to access and mutate either of these account balances during the sequence of transfer instructions, new money could be gained or lost. Suppose thread t_1 is currently executing this six-statement sequence. Further suppose thread t_2, deposits lottery winnings into the savings account *between* t_1's steps 1 and 3. Step 3 (on t_1) overwrites the huge deposit. Placing the entire transfer in a transaction prevents such situations from happening:

```
(use 'clojure.test)

(def savings (ref 10000))
(def checking (ref 500))

(is (and (= @savings 10000) (= @checking 500)))

(dosync
  (let [amount 100]
```

```
    (alter savings - amount)      ; alter savings by subtracting amount
    (alter checking + amount)))  ; alter checking by adding amount

(is (and (= @savings 9900) (= @checking 600)))
```

Refs work well when you need to coordinate access to multiple objects in a synchronized fashion; the language mandates that any attempt to alter refs outside of a transaction throw an exception. For a different scenario—safe access to a *single* object in an *asynchronous* fashion—Clojure offers **agents**.

10.7 AGENTS

Like vars and refs, agents store values which you retrieve via `deref` (or `@`). To change the value managed by the agent, you dispatch an **action**. An action is simply a function call that, provided the optional validation test passes, produces the new value for the agent. The function executes asynchronously on a thread from thread pools managed by Clojure; the programmer never starts any threads or performs any locking. The threads are nondaemon threads, however, so the programmer must explicitly shutdown the pool. The following trivial example sends 50,000 increment actions to a counter agent with a validation function ensuring the counter never becomes negative. It prints the value while, presumably, the queued actions are still being executed, then after a two-second pause giving the queued actions more time to execute, prints the value again. Finally, the script shuts down the agent system, terminating any as yet unprocessed actions.

```
(def counter (agent 0 :validator #(>= % 0)))

(dotimes [i 50000] (send counter + 1))

(println @counter)   ; Will likely be < 50000
(Thread/sleep 2000)
(println @counter)   ; Will likely be 50000
(shutdown-agents)
```

Note how Clojure's agent-based programming exhibits strong philosophical differences from "object-oriented" programming. In OOP, we focus on objects with internal state and behavior defined for that object in particular or objects of a certain type. Agents, on the other hand, have state but no behavior! No restrictions are placed on the functions you can send to the agent. Indeed, the actions are the center of attention.

Emphasizing functions over objects does not mean we can't model iconic object-oriented tasks. In the next section, we take a look at what Clojure offers.

10.8 THE EXPRESSION PROBLEM

In previous chapters we've seen various implementations of cows, horses, and sheep, each making their own type-specific sound. Generally, the animals were defined with classes, with a common superclass `Animal` specifying that each animal should have a name and speak by returning a message saying that the animal (noted by its name) makes its sound. In the following script, we create these animal classes in Clojure, and, rather than inheriting

from a superclass, we define a **protocol** for sound-making that each class will implement. Unlike Java, however, but like Python, each "method" requires an explicit parameter for the receiver. As protocols specify only behavior and no state, methods that would ordinarily belong a superclass are, in Clojure, just functions:

```
(defprotocol Sounder
  (sound [sound-maker]))

(defrecord Horse [name] Sounder
  (sound [this] "neigh"))
(defrecord Cow [name] Sounder
  (sound [this] "moooo"))
(defrecord Sheep [name] Sounder
  (sound [this] "baaaa"))

(defn speak [animal]
  (str (.name animal) " says " (sound animal)))
```

The `defrecord` macro accepts a class name, fields, protocols that it should implement, and the bodies of any methods it needs. Assuming a Clojure installation running on a JVM, Clojure records are real Java classes (instances of `java.lang.Class`), obeying the usual Java interop syntax for construction, e.g. `(Horse. "CJ")`, and for field access, e.g. `(.name h)`. Let's verify this:

```
(use 'clojure.test)
(def h (Horse. "CJ"))
(is (= (speak h) "CJ says neigh"))
(def c (Cow. "Bessie"))
(is (= (speak c) "Bessie says moooo"))
(is (= (speak (Sheep. "Little Lamb")) "Little Lamb says baaaa"))

(is (= (type h) user.Horse))
(is (= (type Horse) java.lang.Class))
```

Our animal system now has three classes and one class-specific method. But we can imagine extending our project with both new kinds of animals—dogs, perhaps—or new kinds of functionality, such as producing an emoji character. See Figure 10.5. The **expression problem** refers to the situation in some languages that extending a system in one direction is easy but extending in the other requires touching existing code. Languages with classes clearly have no trouble extending a system with new classes; Clojure is no exception:

```
; Add a new class
(defrecord Dog [name] Sounder
  (sound [this] "woof"))
```

Adding new functionality to existing classes (without modifying the source code for existing classes) requires that the language has a mechanism for **type extension**. The `extend-type` macro solves the expression problem in Clojure:

```
; Add new functionality
(defprotocol EmojiDescribable (emoji [this]))
(extend-type Horse EmojiDescribable (emoji [this] "\uD83D\uDC0E"))
(extend-type Cow EmojiDescribable (emoji [this] "\uD83D\uDC04"))
(extend-type Sheep EmojiDescribable (emoji [this] "\uD83D\uDC11"))
(extend-type Dog EmojiDescribable (emoji [this] "\uD83D\uDC15"))

(println (emoji h))
(println (emoji c))
(println (emoji (Sheep. "Woolie")))
(println (emoji (Dog. "Spike")))
```

Protocols enable the creation of objects with shared behaviors in a Ruby-esque or Java-esque fashion. But Clojure *also* supports a style similar to, but not quite the same as, Julia's generic functions. A Clojure **multimethod** specifies a dispatching function and one or more actual methods that dispatch based on the result of applying the dispatching function to the method arguments. Here is our animals example, implemented with a multimethod `sound`, dispatching based on the result of applying `:Animal` to the object we'd like to hear a sound from. Note that in this example, we have *not* introduced any new types. The animal objects are just plain maps!

```
(defmulti sound :Animal)
(defn speak [animal]
  (str (:name animal) " says " (sound animal)))

(defn horse [name] {:Animal :horse :name name})
(defmethod sound :horse [h] "neigh")

(defn cow [name] {:Animal :cow :name name})
(defmethod sound :cow [c] "moooo")

(defn sheep [name] {:Animal :sheep :name name})
(defmethod sound :sheep [s] "baaaa")
```

Figure 10.5 The expression problem

```
(use 'clojure.test)
(def h (horse "CJ"))
(is (= (speak h) "CJ says neigh"))
(def c (cow "Bessie"))
(is (= (speak c) "Bessie says moooo"))
(is (= (speak (sheep "Little Lamb")) "Little Lamb says baaaa"))
```

Notice we have no expression problem with multimethods either:

```
; Add a new kind of animal
(defn dog [name] {:Animal :dog :name name})

; Add a new method
(defmulti emoji :Animal)
(defmethod emoji :horse [h] "\uD83D\uDC0E")
(defmethod emoji :cow [c] "\uD83D\uDC04")
(defmethod emoji :sheep [s] "\uD83D\uDC11")
(defmethod emoji :dog [d] "\uD83D\uDC15")
(doseq [a [h c (sheep "Wooly") (dog "Spike")]] (println (emoji a)))
```

Though our animals example did not take advantage of this capability, multimethods allow dispatch on (any or all of its) arguments, while protocol-based dispatch considers (only) the *receiver*. We saw this difference before, when contrasting Ruby and Julia. Clojure and Julia implement multiple dispatch a little differently, however; Julia considers only the types of the method arguments, while Clojure executes a user-defined function on the argument values to determine the method to dispatch. You'll explore this highly flexible feature in the end-of-chapter exercises.

10.9 MACROS

We've saved the best of Clojure for the last section.

You've undoubtedly noticed that programs (code) in Clojure are just lists (data). Programs can easily output lists and then evaluate them. This makes metaprogramming feel just right: In most other languages we generate code as strings (which need compilation) or byte code (which is unreadable for average humans), Clojure simply generates lists. Julia is a bit like Clojure in that it generates expression objects, but Clojure's generated code *looks* exactly like the code you'd write yourself. Clojure is a **homoiconic** language.

As in Julia, a macro accepts unevaluated expressions, performs some computations, and evaluates the result of these computations. Three classic examples come to mind here. The first "evaluates" an expression written in infix form:

```
(defmacro eval-infix [form]
  (list (second form) (first form) (second (rest form))))

(use 'clojure.test)
(is (= (eval-infix (20 * 7)) 140))
```

```
(is (= (macroexpand '(eval-infix (20 * 7)))
    '(* 20 7)))
```

Because macro invocations do *not* evaluate arguments before the call, invoking `(eval-infix (20 * 7))` passes a 3-element list to the macro. The macro returns a new three-element list, specifically `(* 20 7)`, which is immediately evaluated to produce 140. The `macroexpand` function will show you the forms produced by the macro without evaluation.

The second popular example highlights the use of macros to "extend the syntax" of the language. We define a new form, called `unless`:

```
(defmacro unless [condition this that]
  (list 'if (list 'not condition) this that))

(use 'clojure.test)
(is (= (unless (< 1 2) 10 7) 7))
(is (= (unless (< 2 1) 10 7) 10))
(is (= (macroexpand '(unless (< 1 2) 10 7))
    '(if (not (< 1 2)) 10 7)))
```

We can write this macro another way, using the **syntax quote** (`):

```
(defmacro unless [condition this that]
  `(if (not ~condition) ~this ~that))
```

The syntax quote is similar to the regular quote except that (1) symbols inside syntax quotes are fully qualified, e.g. `(unless 1 2 3)` expands into `(if (clojure.core/not 1) 2 3)`, and (2) we can *unquote* within syntax quoted forms. Unquoting simply tells Clojure to evaluate certain forms inside of a quoted form. The use of syntax quoting tends to produce more readable macros, constructing the resulting forms from patterns rather than from complex expressions piecing together lists.

The third popular example comes from the Clojure core library itself—the `or` macro. Clojure's `or` evaluates arguments from left to right, returning the first truthy value encountered, or where no truthy operands exist, returning the last argument—or `nil` if no arguments. This means the expression `(or false nil "hello" (/ 1 0))` produces `"hello"`, without ever evaluating the fourth argument.[2] Here's the code:

```
(defmacro or
  ([] nil)
  ([x] x)
  ([x & next]
    `(let [or# ~x]
      (if or# or# (or ~@next)))))
```

We're seeing a definition by cases for the first time; we specify what the macro should produce when given zero arguments, one argument, or more than one argument. Let's look at the third case closely. The macroexpansion

[2]Rght here we have one of several answers to "Why do we have macros?" We can't easily write short-circuit-or as a function since functions evaluate their arguments before a call. Granted we could *explicitly* pass in quoted forms and invoke `eval`, but such a mechanism does not integrate nicely with the rest of the language.

```
(macroexpand '(or alpha beta gamma delta))
```

yields

```
(let* [or__4469__auto__ alpha]
  (if or__4469__auto__
    or__4469__auto__
    (clojure.core/or beta gamma delta)))
```

Suffixing an identifier with a hash character causes the macroexpansion to generate a new symbol guaranteed not to conflict with any other; this prevents the macro from accidentally shadowing an existing binding. Also note the use of **unquote splicing** (`~@`). Had we simply written `~next` rather than `~@next`, our expansion would have ended with `(clojure.core/or (beta gamma delta))`. Splicing helps a great deal in keeping macros easy to read.

Macros are not arcane features, but very common in Clojure. A large number of the definitions in the standard library are macros, including `if-not`, `when`, `delay`, `and`, `or`, `locking`, `->`, `->>`, `defmulti`, `defmethod`, `dotimes`, and `doseq` and literally dozens more. You can find the source code for these macros at [46].

Because macros provide a means to transform code prior to execution, they give us the ability to create powerful language extensions in a way that is qualitatively different from adding functions to a standard library. There are no limits to the number of macros you can create. However, as in any high-level language, you will see some forms that are so low-level, so wired into the inner workings of Clojure itself, that they could not be (possibly or conveniently) written even as macros. There are a few, and are known in Clojure as special forms. We've listed for reference, a few of these, in Table 10.1.

`def`	Sets the root binding of a variable, creating it if necessary.
`if` e_1 e_2 e_3	Evaluates e_1. If truthy, evaluates and produces e_2. Otherwise evaluates and produces e_3.
`do` $e_1 ... e_n$	Evaluates the expressions in order, producing e_n.
`let`	Evaluates e in the context of local bindings.
`quote` *form*	Produces *form* without evaluating it.
`var` *sym*	Produces the var bound to *sym*, not the value in the *var*.
`fn`	Defines a function.
`loop`	Like `let`, but sets up a **recursion point**.
`recur`	Transfers control to the recursion point with bindings rebound to the arguments of the `recur` form.
`throw` e	Similar to Java's `throw`.
`try`	Similar to Java's `try-catch-finally`.

Table 10.1 Some Clojure special forms

Clojure, like all Lisps, uses special forms where other languages would use syntactic structures such as `if` and `while` statements, function declarations, and all kinds of punctuation. In Clojure, there really is no surface syntax; all is abstract syntax, where function calls, macro calls, and special form invocations all look the same. This does make macro writing a bit less cumbersome than say, in Julia, where the programmer must mindfully construct the abstract syntax tree of the code being generated. It will always, perhaps, remain a design choice that is both very popular and very unpopular. But there's something compelling,

perhaps profound, about this integration of code and data, the homoiconicity, and the ease of metaprogramming in the family of Lisp languages. Something worthwhile, too, as Eric S. Raymond writes:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

CLOJURE WRAP UP

In this chapter, we were introduced to Clojure. We learned that:

- Clojure is a Lisp, with powerful metaprogramming facilities gained from code-as-data, such as macros.
- Clojure programs are executed with a reader and evaluator. The reader reads source text and creates forms, which can be symbols, keywords, literals, or a variety of interesting data structure representations. The evaluator evaluates forms. Symbols and lists are evaluated specially. Quoting suppresses evaluation, allowing symbols and lists to be referred to directly.
- The `let` special form creates local bindings, and for convenience, so do a few other forms.
- Instead of tail recursive functions, the language provides the special forms `loop` and `recur`.
- Clojure favors dynamic typing.
- Code appears as (nested) lists, with the head treated as the operator (function, macro, or special form), and the tail (the rest of the list elements) its operands.
- The collections—lists, vectors, sets, and maps—are all immutable, persistent, and have value semantics for equality. They can each be treated as lazy sequences, upon which dozens of predefined operations can be performed.
- Namespaces are used to organize large programs. Namespaces hold vars and classes.
- Vars are one of the few mutable objects in Clojure.
- Transients speed up the creation of certain persistent structures, taking advantage of mutability that programmers are expected to isolate.
- Refs are mutable but can only be updated within a transaction, providing synchronous coordinated access to multiple values. Agents allow multiple threads to access a single value asynchronously. Both agents and refs, like vars, wrap a value which the programmer accesses via `deref`.
- Protocols and multimethods solve the expression problem, allowing both new types and new operations to be added to a system without modifying existing code.
- A powerful macro system together with homoiconicity characterizes Lisp dialects, including Clojure, like no other feature. The interplay between code and data is considered by some to be an enlightening experience once fully understood. Taking a more pragmatic view, Clojure's macro system allows for relatively clean language extensions, especially when employing syntax quoting, unquoting, and splicing.

For further study in Clojure, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Clojure, see the Clojure page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/clojure.

EXERCISES

Now it's your turn. Continue exploring Clojure with the activities and research questions below, and feel free to branch out on your own.

10.1 Locate the Clojure home page and the documentation section. Read the language Rationale page. Clojure's creator, Rich Hickey, has written an excellent piece on state versus identity. Read this essay. Give a concrete example of Java, for example, conflating identity and state.

10.2 Experiment with the Clojure REPL.

10.3 Find the class of each of the following expressions. Invoke `(class` e`)` in the REPL to find out.

```
=> (* 3 875628774836565478347653846 5M)
=> (/ 9 3)
=> (do (def x 0) #'x)
=> #(* %2 %1 %3)
=> #"abc"
=> *in*
=> (quote quote)
```

10.4 In this chapter we encountered the `->>` macro. A similar macro, `->` also exists. How, exactly do they differ? Give an example in which `->` must be used over `->>`.

10.5 The word count program in the first section of this chapter was described without too much detail. In particular, neither `merge-with` nor `into` were explained. Research these forms and use them in an example script of your own.

10.6 Extend the word count script from this chapter to consider the full range of Unicode letters as words.

10.7 We've seen the special syntax for vector (`[`$v_1 \ldots v_n$`]`), map (`{`$k_1\, v_1 \ldots k_n\, v_n$`}`), and set (`#{`$v_1 \ldots v_n$`}`) literals. We can't use literals for collections with thousands of items, however. Give Clojure expressions for: (a) the vector of the first thousand primes, (b) the set of the first thousand multiples of 3, starting at 0, and (c) a mapping from the characters `\a`...`\m` to `\n`...`\z` and `\n`...`\z` to `\a`...`\m`.

10.8 We've seen that `(+)` evaluates to 0 because the identity of addition is 0. Similarly `(*)` produces 1, the identity for multiplication. What are the identities of `max` and `min`, if any? If none, how does Clojure evaluate `(max)` and `(min)`?

10.9 Experiment with the application of the less-than function `<` to operands of differing types. Do numeric types freely cast? How do values like `nil` compare with numbers and strings? Can strings be compared with vectors or lists? Are booleans comparable?

10.10 Compare Clojure's equality function (`=`) with that of Java. Does it operate more like Java's `==` operator or its `equals` method?

10.11 How does `interleave` work with sequences of different sizes?

10.12 What is the difference, if any, between `(partial < 7)` and `#(< % 7)`? In which situations would `partial` be preferred to a function expression?

10.13 What function is this?

```
(defn mystery [n]
  (loop [a 0 b 1 i n]
    (if (zero? i) a (recur b (+' a b) (dec i)))))
```

10.14 This is not a factorial function:

```
(defn fact-maybe [n]
  (let [f (fn [x a] (if (<= x 1) a (f (dec x) (*' a x))))]
    (f n 1)))
```

What exactly does the function do? Make a small adjustment to make the function compute the factorial of its argument.

10.15 Use a transient to build up a vector of the first 1000 primes.

10.16 The following script:

```
(def x 0)
(let [
  t1 (Thread. #(dotimes [_ 10000] (def x (inc x))))
  t2 (Thread. #(dotimes [_ 10000] (def x (dec x))))]
  (.start t1)
  (.start t2)
  (.join t1)
  (.join t2)
  (println x))
```

need not print 0. Rewrite the script so that increments and decrements are done in a transaction, and verify that multiple runs of your program print 0.

10.17 Redo the previous problem using agents.

10.18 What problems could occur if a mutable value was stored in a ref?

10.19 Extend the expression problem example from Section 10.8 using protocols to both (a) add a type for pigs, and (b) add a function to produce the binomial name (the Latin, or scientific, name consisting of the genus and species) of the animal.

10.20 Repeat the previous problem to extend the animal example using multimethods.

10.21 Learn about the library `core.async`. Develop a simple script that defines an unbuffered channel in which one process writes a message and a second process reads and prints the message from the channel.

10.22 Find out how your Clojure implementation autogenerates symbols in macros, or if

perhaps the language definition itself specifies a particular format. Knowing this, can you or can you not guarantee that you would never get unlucky and have a name clash with a generated symbol?

10.23 Are Clojure macros hygienic? If so, show how they fit a commonly accepted definition of the term. If not, are they (1) not hygienic at all, or (2) perhaps partially hygienic? In which situations do they fall short of being fully hygienic?

10.24 Is it possible to determine at runtime, whether a symbol refers to a function, a macro, or a special form?

10.25 Run the following script on your Clojure installation (we've taken it straight from the Clojure documentation, with slight modifications, so you might have seen it before):

```
(defn len [x] (.length x))

(defn fast-len [^String x] (.length x))

(doseq [f [len fast-len]]
  (time (reduce + (map f (repeat 1000000 "dogs")))))
```

The parameter of `fast-len` is annotated with a **type hint**. How exactly do type hints explain the large performance difference between the two functions?

10.26 Read about Clojure transducers, and give an example of their use.

CHAPTER 11

Haskell

Haskell is the most widely known of the so-called pure functional languages.

First appeared	1990
Recognized for	Pure functions, nonstrict semantics, type classes, monads
Notable uses	Academia, Software validation and verification, Modeling, Compilers
Tags	Nonstrict, Lazy, Functional
Six words or less	"Pure functions meet laziness and typeclasses"

The 1980s saw an explosion in research into functional programming, with some twenty or so such languages coming to life. Haskell arose from an effort to bring together various ideas and contributions from these efforts, and has since gained many fans both in academia and industry. Like many of its direct ancestors and cousins in the so-called ML family of languages, it sports features such as higher-order functions, static **Hindley-Milner typing**, tuple types, pattern matching, and modules. But Haskell makes its mark by successfully bringing together three big ideas: it has a **nonstrict semantics**, categorizes types via **typeclasses**, and meets enough of the requirements to be known as a **pure** functional language.

Haskell is a rich language with much to offer both programmers and those interested in the study of programming languages, primarily due to design decisions arising from the attempt to reach a high degree of functional purity throughout the language. A **pure function** always produces the same output for a given input, so questions regarding the handling of state, side-effects, input/output, and randomness naturally arise. Haskell's approach to these issues give the language a characteristic feel with interesting problem-solving paradigms that are indeed powerful yet unfamiliar to many who know exclusively the more mainstream imperative languages.

We'll begin our exploration of Haskell with our three common introductory programs. We'll then look at language elements including modules, functions, and types. We'll cover Haskell's approach to type inference (inherited from its ancestors in the ML family). We'll look at algebraic datatypes, pattern-matching, and parallelism. We'll see how typeclasses are related to the generics of previous languages on our tour. We'll close with an exploration of **effects**, and Haskell's use of category-theoretic mathematical machinery to describe them.

11.1 HELLO HASKELL

Let's get right to the traditional first program:

```
main =
    mapM_ putStrLn
        [ show a ++ ", " ++ show b ++ ", " ++ show c
        | c <- [1..40], b <- [1..c], a <- [1..b]
        , a^2 + b^2 == c^2
        ]
```

Here a **list comprehension** generates a list of strings as it iterates through triples (a, b, c) meeting the right-triangle condition. Each triple description is manufactured with `show`, producing strings from numbers, and `++`, performing concatenation. The interestingly-named `mapM_` applies `putStrLn` ("put string line") to each list member to ultimately write them one-per-line to standard output. Technically, `putStrLn` takes a string and produces an **IO action**, which, when eventually performed, will print to standard output. The function `mapM_` not only maps `putStrLn` over the list elements, but collects all the resulting IO actions into a single action. Binding this action to `main` carries out the compound action of printing every line.

Our second teaser generates command line argument permutations. We get permutation generation for free from the standard library's module `Data.List`:

```
import System.Environment (getArgs)
import Data.List (permutations, intercalate)

main =
    getArgs >>=
        mapM_ (putStrLn . intercalate "\t") . permutations
```

Command line arguments are obtained from `getArgs`, but as these arguments come from outside the program, Haskell considers them **tainted**, and stores them not in a plain list, but rather in a list wrapped in an IO action, specifically, an IO action for *reading*. The `>>=` operator takes the list out of the wrapper (it "reads" the argument list) and passes it to a function. The function receiving the argument list applies `permutations` to this list, resulting in a list of lists, each of which gets intercalated (this is just Haskell's word for what every other language seems to just call `join`) with tabs and printed to standard output. The dot operator in Haskell (`.`) composes functions: `(f.g)` applies g then f, i.e., `(f.g)(x)` is exactly `f(g(x))`.

Now for the third teaser.

```
import System.IO (getContents)
import Data.Text (pack, unpack, splitOn)
import Data.List (sortOn)
import Text.Printf (printf)

splitOnComma :: String -> [String]
splitOnComma = map unpack . splitOn (pack ",") . pack
```

```haskell
frequent :: [String] -> Bool
frequent (_:_:games:_) = ((read games)::Int) >= 15

data Player =
    Player { name :: String, team :: String, ppg :: Float }

toPlayer :: [String] -> Player
toPlayer (team:name:games:points:_) =
    let ppg = ((read points)::Float) / ((read games)::Float) in
        Player { name = name, team = team, ppg = ppg }

toString :: Player -> String
toString player =
    printf "%-22s%-4s%8.2g" (name player) (team player) (ppg player)

main =
    getContents >>=
        mapM_ putStrLn . map toString . take 10 . reverse .
        sortOn ppg .  map toPlayer . filter frequent .
        map splitOnComma . lines
```

This example illustrates quite a bit of Haskell, including function import from standard library modules, type signatures, custom datatypes, and pattern matching. Outside of any syntax you might find unfamiliar at this point, the body of `main` does express the computation rather directly:

1. `getContents` fetches the contents of standard input as, you may have guessed, a string wrapped in an IO action. We use `>>=` to extract the string and pass it to a processing pipeline.

2. The file contents are processed by a function composed of many smaller functions. Reading from right to left, we see that the file content (stored initially as a string) is first broken up into a list of strings, each representing a line of player data. Each line is then split on the `","` delimiter to extract the player's team, name, games played, and total points accumulated. We keep only the lines with players that played in 15 or more games, construct player objects containing the important points-per-game statistic, sort the list of players by this metric, reverse the list so the higher points-per-game players are first, take the top 10 players, and finally produce a nicely formatted description of each player.

3. Our friend `mapM_ putStrLn` arranges for the printing to take place upon binding to `main`.

As is our custom, we've skipped a number of details as our introductory examples exist only to give brief glimpses of the language in action. We'll pick up missing pieces in the upcoming language overview, especially the theory behind how Haskell goes to great lengths to "control" effects such as input/output. Many programmers find Haskell unfamiliar at first, but often agree that there are a number of fascinating features making the language a very worthwhile study.

11.2 THE BASICS

Haskell's core language is quite small. It has no need for statements with fancy syntax as it relies on function composition for control flow. It features only six basic types: `Bool` (with instances `True` and `False`), `Int` (bounded integers), `Integer` (unbounded integers), `Float` (single-precision integers), `Double` (double-precision integers), and `Char` (consisting of all Unicode characters). It also contains three mechanisms for defining new types:

- **Tuples**. $(T_1, T_2, \ldots T_n)$ is the Haskell product type; for example, the value `(7, "abc")` has type `(Float, String)`. Tuple types can have any number of constituent types, including zero. The empty tuple type, `()`, is called the **unit type**, and it (necessarily) has only one instance, namely `()`.[1]
- **Functions**. $T_1 \rightarrow T_2$ is the type of functions from T_1 to T_2. Haskell functions do not have "multiple parameters" or "multiple return values," since accepting and returning tuple values provides the same functionality! Here's an example: the function `\x -> (not x, x)` has type `Bool -> (Bool, Bool)`.
- **Algebraic Data Types**. All types other than the basic six types, tuples, and functions, are defined "algebraically"—interestingly, as we'll see, as sums of products. These include lists, strings, dictionaries, enumerations, records, and even types comprising a class hierarchy in the so-called "object-oriented languages." We will cover these types in Section 11.6.

The types of expressions are fully resolved at compile-time and generally not available at run-time.[2] You can, however, inspect types within a Haskell REPL such as `ghci`:

```
$ ghci
Prelude> :t (True, 'u')
(Bool, Char)
Prelude> :t \(x, y) -> (y::Int) - x
(Int, Int) -> Int
Prelude> data PrimaryColor = Red | Blue | Green
Prelude> :t Blue
PrimaryColor
```

Programs are sequences of type and function definitions, optionally grouped into modules. Program logic lives in expressions with basic values, operators, and function calls. The only syntactic conveniences are the bracketing of lists with `[` and `]`, the bracketing of strings with double quotes, and forms such as `let`, `where`, `if`, `case`, and `do`—a very small syntax indeed! Expression structure is determined from two rather simple rules:

- Function application associates to the left: `f x y z` is the same as `((f x) y) z`.
- Unary operators *always* take precedence over binary operators: `sqrt x + y` is the same as `(sqrt x) + y`.

The frequently used operator `$` reduces the need for parentheses: `f $ x` is just `f x` but right-associative. Thus `f $ g $ h $ x` is the same as `f (g (h x))`. The operator has the lowest precedence of any binary operator, thus `sqrt $ x + y` is the same as `sqrt (x + y)`.

[1] It may appear confusing to say that `()` has type `()`, but it makes sense as a natural extension of saying `(1, true)` has type `(Int, Bool)`.

[2] We say "generally" as component types of an algebraic type can be exploited in run-time pattern-matching constructs.

Here is a quick peek at some of the basic types and associated operators. We'll be using the `guard` function for assertions so our scripts self-test. We'll explain the `do` syntax in the upcoming section on monads:

```
import Control.Monad (guard)
import Data.Char (ord, chr)

main = do
    guard $ 3 * 8 + 5 ^ 2 == 49
    guard $ 19 / 5 == 3.8
    guard $ div 19 5 == 3
    guard $ mod 19 5 == 4
    guard $ 2 ^ 128 == 340282366920938463463374607431768211456
    guard $ 2 ** 128 == 3.402823669209385e38
    guard $ ord '¢' == 162 && chr 162 == '¢'          -- codepoint
    guard $ (False && (1 < 5) || even 100) == True    -- logical ops
    guard $ (let x = 3 in x + 10) == 13               -- let expression
    guard $ (if 1 < 2 then 21 else 55) == 21          -- if expression
```

Here `div` and `mod` appear to take two arguments, but this is just an illusion—all Haskell functions accept only one argument. The expression `div 19` produces *a function that divides its argument into 19*. Perhaps writing `(div 19)(5)` or even `let divNineteen = div 19 in divNineteen 5` would make the internal behavior more explicit, but the idiomatic `div 19 5` is generally preferred. Functions returning functions for the purpose of simulating multiple arguments are known as **curried** functions, after the mathematician Haskell Curry. Functions accepting a single tuple to simulate multiple arguments are **uncurried**.

Definitions in Haskell can be, and usually are, prefixed with a **type signature**:

```
import Control.Monad (guard)

dozen :: Int
dozen = 12

isMultipleOfFive :: Int -> Bool
isMultipleOfFive n = mod n 5 == 0

-- Input is a tuple, output is a value (uncurried)
average :: (Double, Double) -> Double
average (x, y) = (x + y) / 2.0

-- Input is a value, output is a function (curried)
mean :: Double -> (Double -> Double)
mean x y = (x + y) / 2.0

main = do
    guard $ 5 * dozen == 60
    guard $ isMultipleOfFive 100
    guard $ average (3.5, 10) == 6.75
    guard $ mean 3.5 10 == 6.75
```

Haskell lists have a special syntactic form, and a wealth of pre-defined operators and functions. A few appear in the script below, along with an illustration of Haskell's ranges, list comprehensions, and anonymous functions:

```
import Control.Monad (guard)
import Data.List (intersperse, intercalate)

main = do
    guard $ null []                          -- is empty
    guard $ length [3,5,8,2,1] == 5
    guard $ head [3,5,8,2,1] == 3            -- first element
    guard $ tail [3,5,8,2,1] == [5,8,2,1]    -- all but first
    guard $ 3 : [5,8,2,1] == [3,5,8,2,1]     -- cons (fast)
    guard $ [3,5,8] ++ [2] == [3,5,8,2]      -- append (slow)
    guard $ take 2 [3,5,8,2,1] == [3,5]      -- first n elements
    guard $ drop 2 [3,5,8,2,1] == [8,2,1]    -- all but the first n
    guard $ init [3,5,8,2,1] == [3,5,8,2]    -- all but last
    guard $ last [3,5,8,2,1] == 1

    -- Ranges and comprehensions
    guard $ [2..5] == [2,3,4,5]
    guard $ [x | x <- [3..8], even x] == [4, 6, 8]
    guard $ [x + y | x <- [1,2], y <- [9,0,2]] == [10,1,3,11,2,4]

    -- Splits, joins, and zips
    guard $ splitAt 3 [3,5,8,2,1] == ([3,5,8],[2,1])
    guard $ concat [[3],[5,8],[],[2,1]] == [3,5,8,2,1]
    guard $ intersperse 0 [1,2,3] == [1,0,2,0,3]
    guard $ intercalate [0,0] [[1],[2,3]] == [1,0,0,2,3]
    guard $ zip [1,2,3] ['a','b'] == [(1,'a'),(2,'b')]

    -- Higher order and anonymous functions
    guard $ takeWhile (\x -> x < 7) [3,5,8,2,1] == [3,5]
    guard $ dropWhile (not . even) [3,5,8,2,1] == [8,2,1]
    guard $ filter (\x -> x < 7) [3,5,8,2,1] == [3,5,2,1]
    guard $ filter even [1..10] == [2,4,6,8,10]
    guard $ map succ [3,5,8,2,1] == [4,6,9,3,2]
    guard $ concatMap (\x -> [x,-x]) [1..3] == [1,-1,2,-2,3,-3]
    guard $ all even [3,5,8,2,1] == False
    guard $ any even [3,5,8,2,1] == True
    guard $ sum [1..10] == 55
    guard $ product [1..10] == 3628800
```

Haskell lists are immutable, and like those in Clojure, are implemented with persistent data structures. Therefore, as in Clojure, adding to the front of the list—what Clojure calls `conj` and for which Haskell uses the **cons operator** `:`—is far more efficient than appending two lists—Haskell's `++` operator.

The type `String` is simply an abbreviation for the type `[Char]`; therefore, strings are just lists, though with a convenient syntax.

```
import Control.Monad (guard)
import Data.Char (ord, toUpper, isLetter)
import Data.List (intersperse, intercalate)

main = do
    guard $ "a←b" == ['a', '←', 'b']
    guard $ null ""
    guard $ map ord "a←b" == [97,8592,98]
    guard $ map toUpper "a←b" == "A←B"
    guard $ take 2 "a←b" == "a←"
    guard $ length "a←b" == 3
    guard $ [c | c <- "a←b", isLetter c] == "ab"
    guard $ intersperse ',' "abc" == "a,b,c"
    guard $ intercalate "<->" ["AB","BC","CA"] == "AB<->BC<->CA"
    guard $ all isLetter "a←b" == False
```

Processing strings as lists of characters can be slow, so strings are often packed into text objects with an efficient internal representation, then unpacked if the list representation is needed. You'll explore text objects in the end-of-chapter exercises.

You may have noticed that Haskell has no conventional statements: the body of a function is always an expression.[3] Sequencing computations is done by composing functions, for which `let` and `where` may be used for syntactic sugar.

```
import Control.Monad (guard)

-- These three function bodies are equivalent!
a = let x = 5 in x + 3
b = x + 3 where x = 5
c = (\x -> x + 3) 5

main = do
    guard $ a == 8
    guard $ b == 8
    guard $ c == 8
```

There are additional syntactic conveniences, of course. Rather than long chains of `if`s and `else`s, complex conditional logic is often best expressed with **guards**—boolean expressions that determine "what to return":

```
grade :: Int -> [Char]
grade score
    | score >= 90 = "A"
    | score >= 80 = "B"
    | score >= 70 = "C"
    | score >= 60 = "D"
    | otherwise   = "F"
```

[3]This even applies to the `do`-notation, which is syntactic sugar for an expression, and not a block of statements. We'll describe this form in the section on Monads below.

The guards can use values from a `where` clause:

```
quadrant :: Double -> Double -> [Char]
quadrant x y
    | xPositive && yPositive = "1"
    | xNegative && yPositive = "2"
    | xNegative && yNegative = "3"
    | xPositive && yNegative = "4"
    | xZero && not yZero = "Y axis"
    | not xZero && yZero = "X axis"
    | otherwise = "Origin"
    where (xPositive, xNegative, xZero) = (x > 0, x < 0, x == 0)
          (yPositive, yNegative, yZero) = (y > 0, y < 0, y == 0)
```

Infix operators are curried functions; for example, the type of the division operator (`/`) is `Float -> Float -> Float`. Haskell allows an infix operator to appear in prefix position by enclosing it in parentheses. For example, `(/) 5 8` evaluates to 0.625. The parentheses also allow the operator itself to be bound to an identifier or passed as an argument. There's more: a regular curried function can appear in an infix position by enclosing it in backticks:

```
import Control.Monad (guard)

main = do
    guard $ 3 * 5 == 15
    guard $ (*) 3 5 == 15
    guard $ mod 29 8 == 5
    guard $ 29 `mod` 8 == 5
    guard $ foldl (-) 1 [3,5,2,8] == -17     -- (((1-3)-5)-2)-8
    guard $ foldr (-) 1 [3,5,2,8] == -7      -- 3-(5-(2-(8-1)))
```

Infix operators can be used in **operator sections**, explained in this small script:

```
import Control.Monad (guard)

-- The section (* 5) is the same as \x -> x * 5
-- The section (< 7) is the same as \x -> x < 7
-- The section (1 /) is the same as \x -> 1 / x
main = do
    guard $ map (* 5) [3, 5, 2] == [15, 25, 10]
    guard $ filter (< 7) [3, 5, 8, 2] == [3, 5, 2]
    guard $ map (1 /) [2, 4, 8] == [0.5, 0.25, 0.125]
```

Sections can be used to generate convenient functions; for example, `isZero = (== 0)`, `reciprocal = (1 /)`, and `halfOf = (/ 2)`.

Infix operators are assigned an integer precedence level from 0 (lowest) to 9 (highest), and either left, right, or no associativity. The infix functions from the standard module `Prelude`—the module imported by default into all Haskell code—appear in Table 11.1:

By default, newly defined operators have precedence 9 and are left-associative, but other values can be assigned. Here we define two operators that compute $2x + y$, one with a high

Operator(s)	Prec	Assoc	Description
`!!`	9	L	list-element-at-index
`.`	9	R	composition: $(f\ .\ g)x = f(g(x))$
`^  ^^  **`	8	R	exponentiation

`*  /`
`quot  rem`
`div  mod`
	7	L	multiply, divide, integer division truncated toward 0, remainder, integer division truncated toward $-\infty$, modulo

`+ -`	6	L	add, subtract
`<>`	6	R	monoidal append
`:  ++`	5	L	list-cons, list-append

`<  <=  >  >=`
`=  /=`
`elem  notElem`
	4	None	less, less or equal, greater, greater or equal, equal, not equal, element-of, not-element-of

`<$  <$>` `<*  <*>  *>`	4	None	leftwards application, fmap, left action, applied over, right action
`&&`	3	R	(short-circuit) logical and
`||`	2	R	(short-circuit) logical or
`>>=  >>`	1	L	bind, then
`=<<`	1	R	flipped bind
`$  $!  seq`	0	R	application, strict application, second

Table 11.1 Haskell operators

precedence (8) and one with low precedence (2), illustrated in combination with multiplication (precedence level 7):

```
import Control.Monad (guard)

(<-*->) :: Int -> Int -> Int
(<-*->) x y = 2 * x + y

(>*<) :: Int -> Int -> Int
(>*<) x y = 2 * x + y

infix 8 <-*->     -- binds tighter than multiplication
infix 2 >*<       -- binds looser than multiplication

main = do
    guard $ (8 * 3 <-*-> 5) == 88     -- 8 * (2*3+5)
    guard $ (8 * 3 >*< 5) == 53       -- 2*(8*3) + 5
```

We've chosen to make our new operators nonassociative; to set left or right associativity we would use `infixl` or `infixr`, respectively.

We are now ready to cover some of Haskell's most notable features: lazy evaluation, Hindley-Milner type inference, typeclasses, algebraic datatypes, and its approaches to stateful computation, parallelism, and concurrency.

11.3 LAZY EVALUATION

All expressions in Haskell are evaluated **lazily**, meaning that they are not evaluated until, and unless, they are needed. For example, arguments to a function are not evaluated unless used in the function. The opposite of lazy evaluation is **strict evaluation**. A nonstrict semantics means that expressions can produce a valid value even when subexpressions have no value.[4] Thus we can in Haskell write a function that produces zero regardless of its argument:

```
import Control.Monad (guard)

zero x = 0

main = do
    guard $ zero "ignore me" == 0
    guard $ zero (1 `mod` 0) == 0   -- this is fine!
```

The equivalent program in a language with strict semantics would produce a division by zero error.

A nonstrict semantics allows us to write "infinite structures" as seen in the function definition gracing the language's home page:

```
primes = filterPrime [2..]
  where
    filterPrime (p:xs) =
        p : filterPrime [x | x <- xs, x `mod` p /= 0]

main = print(take 10 primes)
```

Here `primes` is the (infinite) list of all values x greater than or equal to 2, such that x is not divisible by any prime number p. The application of `take` ensures that only the first ten values—the only ones we need to print—are actually computed.

11.4 TYPE INFERENCE

Like several other languages we've encountered so far, Haskell computes the type of every expression at compile time, using type information specified in type signatures where available and performing type inference where information is omitted. While it is not uncommon for many languages that compute static types to require type annotations for, say, function parameters and return types, Haskell employs a **Hindley-Milner type sytem** [121, 76, 19] to deduce the least general type of *any* expression not explicitly annotated. Let's start with a very simple example:

```
mystery x s =
    concat $ replicate (round (sqrt x)) s
```

[4]Loosely, an expression without a value is one whose evaluation would enter an infinite loop or fail. In formal semantics, the meaning assigned to a "valueless" expression is $\perp$, pronounced "bottom." A semantics is **strict** iff the meaning of $f(\perp)$ is $\perp$ for all functions, operators, and constructors f.

None of the identifiers have type annotations, but Haskell can infer them all. How? Roughly, we proceed as follows:

1. Working from left to right, we start by assigning **type variables**: `x` gets type α, `s` gets β, and therefore `mystery` gets $\alpha \to \beta \to \gamma$.
2. We know the type of `repeat` is `Int -> String -> String`, so the type of `round(sqrt x)` must be `Int`.
3. We know the type of `round` is `Float -> Int`, so the type of `sqrt x` must be `Float`.
4. We know the type of `sqrt` is `Float -> Float`, so the type of `x` must be `Float`. Since we had previously assigned the type variable α to `x` we can now **unify** α and `Float`.[5]
5. The second argument of `repeat` (`s`) must be a string, so we unify β with `String`.
6. Because `repeat` produces a string, we unify γ with `String`.
7. Therefore, the type of `mystery` is `Float -> String -> String`.

Let's look at a more interesting case:

```
mystery (x, y) z =
    (y / 2.0, z, [x, x, x])
```

Here, we begin by mapping the type variable α to `x`, β to `y`, γ to `z`, and thus $(\alpha, \beta) \to \gamma \to \delta$ to `mystery`. Because we know the type of float division (`/`) we can infer that y has type `Float` and unify β with `Float`. We can't get any new information for `z` or `x`, but we can determine the return type of the function: $(\texttt{Float}, \gamma, \texttt{List}\,\alpha)$.[6] While it is customary to use Greek letters for type variables when discussing type inference, Haskell sticks to lowercase letters for its type variables, and reports the type of `mystery` as `(a, Float) -> b -> (Float, b, List a)`.

11.5 TYPECLASSES

We've seen how Haskell infers the type of, say, `\(x,y) -> \z -> length("ok" ++ y) > z` to be `(a, String) -> Int -> Bool` and `\x -> x` to have type `a -> a`. But how can we infer the type of

```
\x -> \y -> x < y
```

The inference of `a -> a -> Bool` is incorrect! The type variable `a` cannot stand for *any* type, as functions, for example, cannot be compared with the `<` operator. We must *constrain* the type variable `a` to only those types whose instances can be ordered. Similarly, while computing the type of

```
\x -> \y -> (2 * x, y)
```

we would like to infer `a -> b -> (a, b)` but we must constrain `a` to be a numeric type. Previous languages on our tour have used the term "generic constraints" or "bounded type

[5] In this case, unification appears to be simply instantiation of a type variable with a concrete type. However, in more complex cases, we might discover that two distinct type variables must refer to the same type, and thus need to be "unified" into one variable.

[6] As in several previous languages, Haskell uses type variables for generic types: `List` α is the generic list type, while `List String` and `List Int` are instantiable types.

parameters." Haskell's approach is to have each type belong to zero or more **typeclasses** (not unlike how types implement interfaces in other languages), where each typeclass specifies certain operations. A sampling of Haskell typeclasses is shown in Table 11.2:

Typeclass	Description	Example Members
Eq	Types whose instances can be tested for equality with operators `(==)` and `(/=)`.	Most types except functions and `IO`
Ord	Types whose instances can be ordered with `(<)`. Also supports `(<=)`, `(>=)`, `(>)`, `compare`, `min`, `max`.	Most types except functions and `IO`
Bounded	Types that have a smallest and largest element: `minBound`, `maxBound`.	The six basic types
Enum	Types whose instances have successor and predecessor relationships.	The six basic types
Num	All numbers. Supports `(+)`, `(-)`, `(*)`, `negate`, `abs`.	All numeric types
Integral	Numeric integer types, with integer division: `quot`, `rem`, `div`, `mod`, and combinations thereof.	`Int`, `Integer`
Floating	Floating-point types, supporting operations to access the components of the representation, as well as a variety of functions, including the usual trigonometric and logarithmic operations.	`Float`, `Double`
Fractional	Types whose instances support non-integer division `(/)`.	`Float`, `Double`
Read	Types whose instances can be constructed from strings.	Most types except functions and `IO`
Show	Types whose instances have a string representation.	Most types except functions and `IO`
Traversable	Types whose instances can be traversed, mapped, etc.	Lists, Tuples, `Maybe`
Foldable	Types whose instances can be folded and indexed.	Lists, Tuples, `Maybe`
Monoid	Types whose instances support mapping, concatenating, and appending.	Lists, Tuples, `Maybe`, Functions, `IO`
Functor	Roughly, types providing a context with an operator `<$>` that applies a function to a value-in-context.	Lists, Tuples, `Maybe`, Functions, `IO`
Applicative	Roughly, functors with a `<*>` operator that applies a function-in-context to a value-in-context.	Lists, Tuples, `Maybe`, Functions, `IO`
Monad	Roughly, applicatives with an operator `>>=` that sends a value-in-context to a function that transforms a value to a value-in-context.	Lists, Tuples, `Maybe`, Functions, `IO`

Table 11.2 A few of the Haskell typeclasses

When writing a constrained type expression, the typeclasses of type variables are prefixed to the type with the => arrow:

```
isIncreasingTuple :: Ord a => (a, a) -> Bool
isIncreasingTuple (x, y) = x < y

advance :: (Eq a, Bounded a, Enum a) => a -> a
advance x = if x == maxBound then minBound else succ x

thirdOf :: Fractional a => a -> a
thirdOf x = x / 3

readAndShow :: (Read a, Show b) => (String, b) -> (a, String)
readAndShow (x, y) = (read x, show y)

second :: (Foldable t, Eq a, Num a) => t a -> Bool
second sequence = elem 1 sequence
```

In the end-of-chapter exercises, you will compare Haskell's typeclasses with Java's bounded generics.

Typeclasses form a natural inheritance hierarchy. For example, integral types are also enumerable, real types are numeric as well as ordered, and so on. The complete list of predefined types and typeclasses resides at [88, Ch. 9], though for convenience we've shown a slice of this hierarchy in Figure 11.1.

Figure 11.1 A few of Haskell's built-in typeclasses

You can place any class you define yourself into a typeclass; we'll see how in the next section. In the end-of-chapter exercises you will research how to define your own typeclasses.

11.6 ALGEBRAIC DATA TYPES

Haskell provides a single construct, the **algebraic data type**, for defining both sum and product types. In its most general form, these types are written as sums of products.

Let's look first at sums—types whose instances can have varying forms. For example, responses to a query could be successful (with an answer) or erroneous (with an error code); trees can be empty or nonempty; users can be logged in (with credentials) or anonymous; shapes can be circles or rectangles; entities in a program representation could be numbers, identifiers, or operators. Unlike TypeScript, Haskell *requires* the variants to be labeled, or tagged. In literature, you'll see these types called **tagged unions**, **discriminated sums**, or **disjoint sums**. The "tags" conveniently become functions to construct instances of the data type.

We'll explore via examples. First, we will define a new type called `Response` with two variants. We'll add it to the typeclasses `Eq` and `Show`, a process that Haskell makes extremely convenient via the `deriving` keyword. This gives our class default equality and showing operations, something that many of the previous languages in this text can do:

```
import Control.Monad (guard)

data Response a
    = Error String
    | Ok a
    deriving (Show, Eq)

squareRoot x =
    if x < 0 then Error "negative" else Ok (sqrt x)

main = do
    guard $ squareRoot 9 == Ok 3
    guard $ squareRoot 100.0 == Ok 10.0
    guard $ squareRoot (-4) == Error "negative"
```

Next, we'll define a type for trees and show that functions accepting algebraic data types use **pattern matching** on the constructors. Here trees are either (1) empty, or (2) a node containing a value and a list of trees as its children. We will want string representations of our trees, so we'll need to associate them with the `Show` typeclass; however, we want a custom show function, so we can't use `deriving`:

```
import Control.Monad (guard)

data Tree a
    = Empty
    | Node a [Tree a]

size :: Tree a -> Int
size Empty = 0
size (Node _ children) = 1 + (sum $ map size children)
```

```
instance (Show a) => Show (Tree a) where
    show Empty = "()"
    show (Node value children) =
        "(" ++ show value ++ (foldr (++) "" $ map show children) ++ ")"

main =
    let
        t0 = Empty
        t1 = Node "dog" []
        t2 = Node 17 [Node 31 [], Node 53 []]
        t3 = Node 21 [t2, t2, Empty, Node 3 []]
    in do
        guard $ size t0 == 0
        guard $ size t1 == 1
        guard $ size t3 == 8
        guard $ show (t0::Tree Int) == "()"
        guard $ show t1 == "(\"dog\")"
        guard $ show t2 == "(17(31)(53))"
        guard $ show t3 == "(21(17(31)(53))(17(31)(53))()(3))"
```

Note the "sum of products" structure. The tree is either empty *or* a node (a sum), but the node contains a value *and* a list of children (a product). While the sum variants were named, the product members were not—they are accessed solely by pattern matching. This example also highlights the inclusion of a custom type into a typeclass via the `instance` declaration. Adding the clause `deriving (Show)` in the `data` declaration tells Haskell to generate a basic `show` function, which of course, is unlikely to line up with your specific needs.

Haskell's `List` type is algebraic, too! A list is either (1) empty or (2) an element (the head) connected to a list (its tail). It has two constructors: `[]` for the empty list and `(:)` for lists with a head and tail. Here's how we use them:

```
import Control.Monad (guard)

double :: [a] -> [a]
double [] = []
double (x : xs) = x : x : double xs

main = do
    guard $ (double []::[Int]) == []
    guard $ double [5, 5] == [5, 5, 5, 5]
    guard $ double [2, 5, 3] == [2, 2, 5, 5, 3, 3]
```

In lieu of "class hierarchies," we can use sums:

```
import Control.Monad (guard)

data Shape
    = Circle Float
    | Rectangle Float Float
```

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h

perimeter :: Shape -> Float
perimeter (Circle r) = 2 * pi * r
perimeter (Rectangle w h) = 2 * (w + h)

main = do
    guard $ area (Circle 10) == 100 * pi
    guard $ perimeter (Circle 10) == 20 * pi
    guard $ area (Rectangle 2 8) == 16
    guard $ perimeter (Rectangle 2 8) == 20
```

The sum components of an algebraic type *must* be tagged, but the product components *may* be tagged. Here is an alternative version of the previous shape type with labeled product components. When we take this approach, the labels automatically generate field accessor functions:

```
import Control.Monad (guard)

data Shape
    = Circle {radius :: Float}
    | Rectangle {width :: Float, height :: Float}
    deriving (Show)

area :: Shape -> Float
area (Circle {radius = r}) = pi * r * r
area (Rectangle {width = w, height = h}) = w * h

perimeter :: Shape -> Float
perimeter (Circle {radius = r}) = 2 * pi * r
perimeter (Rectangle {width = w, height = h}) = 2 * (w + h)

main =
    let
        c = Circle {radius = 10}
        r = Rectangle {width = 8, height = 2}
    in do
        guard $ radius c == 10
        guard $ area c == 100 * pi
        guard $ perimeter c == 20 * pi
        guard $ width r == 8 && height r == 2
        guard $ (area r, perimeter r) == (16, 20)
        guard $ show c == "Circle {radius = 10.0}"
        guard $ show r == "Rectangle {width = 8.0, height = 2.0}"
```

Haskell's option type, `Maybe` is also algebraic: it either wraps a value or it doesn't. `Maybe`'s definition is:

```
data Maybe a = Just a | Nothing
  deriving (Eq, Ord)
```

We've seen a fair number of examples of options in previous chapters, so a few trivial examples of using built-in and standard library types will suffice:

```
import Control.Monad (guard)
import Data.List (find, elemIndex, stripPrefix)

main = do
    guard $ elemIndex 'b' "abracadabra" == Just 1
    guard $ elemIndex 'e' "abracadabra" == Nothing
    guard $ stripPrefix "dog" "doghouse" == Just "house"
    guard $ stripPrefix "dog" "cat" == Nothing
    guard $ lookup "two" [("one","uno"),("two","dos")] == Just "dos"
    guard $ lookup "eleven" [("one","uno"),("two","dos")] == Nothing
    guard $ find even [3, 1, 8, 2, 10] == Just 8
    guard $ find even [3, 1, 5, -1, 11] == Nothing
```

As `Maybe` values are sum types, we can pattern match on them. We've seen pattern matching on function arguments above, but now is a good time to show that one can also pattern match in a **case expression**:

```
import Control.Monad (guard)
import Data.List

capitals :: [(String, String)]
capitals =
    [ ("CA", "Sacramento")
    , ("HI", "Honolulu")
    , ("WA", "Olympia")
    ]

tellMeAbout :: String -> String
tellMeAbout state =
    case (lookup state capitals) of
        Just city -> "The capital of " ++ state ++ " is " ++ city
        Nothing -> "I don't know anything about " ++ state

main :: IO ()
main = do
    guard $ tellMeAbout "HI" == "The capital of HI is Honolulu"
    guard $ tellMeAbout "MA" == "I don't know anything about MA"
```

One final useful algebraic type worth mentioning is `Either`, defined as follows:

```
data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show)
```

While `Either` values can be used for just about anything, it's common to use them for operations that can either succeed for fail. Use the left variant for an error and the right for the successful result.

11.7 EFFECTS

One of the more useful dimensions on which to characterize and compare programming languages is the extent to which a language's design, and its culture, encourage, discourage, contain, or try to eliminate **side-effects**. Roughly, a side effect occurs when a function changes a program's state in a way other than producing an explicit return value. Side-effects, and **side-causes**—entities manipulated by a function other than its explicit parameters [57]—can be difficult to manage manually and are often accidentally or maliciously misused. They can be avoided by using only **pure functions**—functions that operate only on their arguments and are side-effect-free. Pure functions produce the same result for the same arguments every time, greatly simplifying testing and formal verification, and creating opportunities for optimization [57].

We can bring causes and effects into the world of pure functions by adding them to a function's inputs and outputs. For example, we can turn the side-effecting function:

```
let log = []                // mutable global variable
function square(x) {
  log.push(`Squared ${x}`) // side cause and side effect!
  return x * x             // typical result
}
```

into a pure function that adds a message to a given log as follows:

```
function loggedSquare([x, log]) {
  return [x * x, log.push(`Squared ${x}`)]
}
```

Persistent data structures implement the same idea. Instead of "mutating list L by overwriting its second element with E," we "return a *new list* similar to the original in all positions except the 2nd, which is now E." Random numbers can be handled similarly: a function can be parameterized by a random number generator and produce the next item from the generator *and* a new generator.

When I/O, databases, and user-interaction enter the picture, however, the idea of augmenting *every* function input with the "state of the world" (input and output history, database snapshots, etc.) and returning a "new state of the world" along with the function's typical result becomes a lot to ask. So Haskell accepts the fact that the real world isn't exactly pure, but ensures that impure code can only execute in impure contexts without affecting pure code. In Haskell, impure I/O actions are rolled into objects that are later executed to perform effects. We'll start with a simple example:

```
main =
    getLine >>= \name -> putStrLn $ "Hello, " ++ name
```

The function `getLine` has type `IO String`, `putStrLn` has type `String -> IO ()`, and operator `>>=` (in this example) has type `IO a -> (a -> IO b) -> IO b`. The type `IO a` represents an action that, when executed, transforms the global state via I/O and returns a value of type a. So `getLine` returns an action that when executed, consumes a line from standard input. `putStrLn` maps a string into an action that appends the string to standard output (a side-effect), and produces the empty tuple `()`. The `>>=` operator composes the two actions into a composite action, which, like all IO actions, is executed only when bound to `main`.

Note that we did not write `"Hello" ++ getLine` as that would be illegal! We can not append a `String` and an `IO String`. Objects of type `IO a` (for any type a) are said to be "in the IO context," which is impure. To use items from an impure context in pure code, we need to extract them from the impure context with specialized operators; remember, Haskell aims to very cleanly separate the pure and impure worlds. The operator `>>=` plays a nice role here. It takes a string out of an IO context and passes it to a function that operates on it and places the result of that operation back into an IO context, perhaps with some additional effects.

We could compose more than just two IO operations with `>>=`:

```
main =
    putStrLn "What is your name?" >>=
    \_ -> getLine >>=
    \name -> (putStrLn $ "Hello, " ++ name) >>=
    \_ -> putStrLn "Have a beautiful day!"
```

Since printing with `putStrLn` returns an `IO ()`, the next function in the IO sequence has an ignored paramter, which looks a little funny. Fortunately, the operator `>>`, pronounced "then," comes in handy here[7]:

```
main =
    putStrLn "What is your name?" >>
    getLine >>=
    \name -> (putStrLn $ "Hello, " ++ name) >>
    putStrLn "Have a beautiful day!"
```

It gets better, even. Haskell's `do` block is a sugared combination of `>>=` and `>>` applications, as needed, with a nice little `<-` notation that gives assignment vibes:

```
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn $ "Hello, " ++ name
    putStrLn "Have a beautiful day!"
```

While Haskell's use of an IO context helped to separate pure and impure code, the idea of computing "in a context" goes far beyond IO. It turns out to be so widespread and useful that our entire next section will be devoted to exploring it in more detail.

11.8 MONADS

In computing, a **context** is a structure or environment that carries extra information, or possibly certain useful constraints, for a computation. The I/O context, as we've seen, is used to manage side effects. There are many others that appear in practice. We'll mention just a few. There are contexts in which:

- Operations may or may not produce a value ("Maybe")

[7] This operator is in fact *defined* as follows: `x >> f = x >>= \_ -> f`.

- Operations may produce one of two kinds of data, generally, a value or an error ("EITHER")
- Operations may produce multiple values, useful in modeling branching or nondeterministic computation ("LIST")
- Operations keep a running log of information as a computation progresses ("WRITER")
- Operations read and write "stateful" information within the context ("STATE")
- Operations that may not complete right away, but will eventually produce a result ("PROMISE")
- Operations work with random data ("RANDOM")

Contexts can be just about anything you like, but one of Haskell's superpowers is that it defines certain operators that you can implement in your own contexts. These operators serve as building blocks for composing computations. Let's look at the most common. We'll use the notation $\boxed{x}$ for "x is in a context," and $...x...$ for "x or a value derived from x."

- The operator `<$>`, known as `fmap`, applies a regular function to a value-in-context, producing a new value in context.

  f `<$>` $\boxed{x} \implies \boxed{...f(x)...}$

- The operator `<*>` applies a function-in-context to a value-in-context, producing a new value-in-context.

  $\boxed{f}$ `<*>` $\boxed{x} \implies \boxed{...f(x)...}$

- `pure` places a value into a context.

  `pure` $x \implies \boxed{x}$

- `>>=`, often called "bind," takes a value out of a context and applies an in-context-producing function to it.

  $\boxed{x}$ `>>=` $g \implies ...g(x)...$ where $g(x)$ is an in-context value

We've given only surface definitions of these operators. In practice, each must satisfy certain properties, or laws, which are given in full in Haskell documentation.[8] Provided the rules are followed, we can define a few important terms. Types providing a context supporting `<$>` are called **functors**. Functors that also support `pure` and `<*>` are called **applicatives**. Applicatives that also support `>>=` are called **monads**. Historically, when working with monads, `pure` is often called `return`.

A surprising number of types are monads (instances of the typeclass `Monad`), including four types we've seen already: `Maybe a`, `Either a b`, `List a`, and `IO a`. Table 11.3 shows how these types satisfy the major requirements of the **Monad** typeclass.

We've just introduced some terms (functor, applicative, monad) without saying why they are important. Clearly, the terms feel relegated to theory land, though the latter, *monad*, does pop up in the documentation for quite a few languages, and you will sometimes hear "Promises are monads," "Optionals are just monads," "Did you know lists are monads?," and even "You probably already invented monads yourself!" [103] Even simply *naming* a concept, even if it is hard to understand, has some value in communicating ideas and

[8] In particular, the laws for `>>=`, namely left identity, right identity, and associativity, are described in [43].

Monad	Description

Maybe a	`pure = Just`
	`f <$> (Just x) = Just (f x)`
	`f <$> Nothing = Nothing`
	`Just f <*> x = f <\$> x`
	`Nothing <*> x = Nothing`
	`(Just x) >>= f = f x`
	`Nothing  >>= f = Nothing`

Either a b	`pure = Right`
	`f <$> (Left x) = Left x`
	`f <$> (Right y) = Right (f y)`
	`Left f <*> x = Left f`
	`Right f <*> x = f <$> x`
	`Left x >>= f = Left x`
	`Right y >>= f = f y`

List a	`pure x = [x]`
	`<$> = map`
	`[] <*> x = []`
	`(f:fs) <*> xs = (f <$> xs) ++ (fs <*> xs)`
	`xs >>= f = concatMap f xs`

IO a	Internally defined

Table 11.3 Some Haskell Monad types

principles, and can lead to further advances in the field. But there are other reasons why monads, in particular, are worthy of study:

- They provide a useful abstraction for sequencing and composing in-context operations.
- They unify approaches to missing data, error handing, and nondeterminism in a single conceptual framework.
- They provide a solution to managing effects in pure functional code.
- Learning monads in the context of *programming* may make it easier to understand category theory should you choose to undertake advanced theoretical studies.
- As a language-independent concept, one can benefit from comparing and contrasting the implementation of monadic computation in various languages.

Although monads are a general concept in computing, we've introduced them in the Haskell chapter since Haskell is rare in using the word Monad in its standard library. Any language with functions-as-values almost certainly "has monads" and can do "monadic computation," but Haskell does this at the typeclass level, rather than giving each monadic type its own built-in syntax or its own set of custom operator names. Monadic *optional chaining*, for example looks like:

```
person?.supervisor?.address?.city?.population
```

in JavaScript, TypeScript, Kotlin, Swift, and other languages, but Haskell's version:

```
person >>= supervisor >>= address >>= city >>= population
```

uses the familiar `>>=` operator shared by all instances of the `Monad` typeclass.

11.9 SPARKS

Haskell, like most modern programming languages, provides a number of features for concurrent and even parallel computing. Both shared-state and message-passing mechanisms are provided. For coarse-grained concurrency, Haskell implementations such as GHC feature threads and locks, and even actors. Like Clojure, concurrent Haskell programming makes use of Software Transactional Memory. For parallel computing, Haskell features **sparks**.

Sparks are not threads, but rather units of work, or tasks, that *can* be allocated to threads. The expression `par` e_1 e_2 sparks the evaluation of e_1, allowing the implementation to run it on a separate thread, while the current thread proceeds with the evaluation of e_2. The result of `par` e_1 e_2 is simply e_2. The programmer's thought process is therefore "I need to evaluate b, but part of this computation involves evaluating a, which *could* be run in parallel before I complete the evaluation of b, so `par` a b is the way to go."

Here's a common introductory example: computing Fibonacci numbers the terribly slow way, but parallel. To compute $fib(n)$ for $n > 1$, we spark the evaluation of fib(n-1), while the current thread evaluates $fib(n-2)$ *then* adds the two previous evaluations.

```
import GHC.Conc

fib :: Int -> Int
fib n
    | n <= 1 = n
    | otherwise = par a (pseq b (a + b))
                      where a = fib (n-1)
                            b = fib (n-2)

main = print (fib 45)
```

`par` expressions often have `pseq` expressions as their second argument. `pseq` e_1 e_2 ensures e_1 is fully evaluated before e_2. Why did we not just write `par a (a+b)`? If we did so, there is a possibility that a may be sparked for evaluation but the current thread would start evaluating a in order to compute $a + b$. Seeing that a is already under evaluation, the spark is said to **fizzle**, and no parallelism occurs. Using `pseq` guarantees that the current thread works on b first before it needs to look at a. In this case, the run time is more likely to actualize the spark for a, that is, run it in parallel with the evaluation of b.

A couple runs of this program using the Glasgow Haskell Compiler, GHC, on a 10-core processor, illustrate true parallelism. In the first run. 45 seconds of processor time was spent, while the process finished in 13.109 seconds of real time, or **wall clock time**:

```
$ ghc -O2 -threaded --make fib.hs

$ time ./fib +RTS -N4
1134903170
./fib +RTS -N4  44.82s user 0.58s system 346% cpu 13.109 total

$ time ./fib +RTS -N8
1134903170
./fib +RTS -N8  37.49s user 1.30s system 619% cpu 6.258 total
```

Combining `par` and `pseq` in this fashion is common, but more sophisticated approaches can be found in the module `Control.Parallel.Strategies`. Sparks are convenient because the programmer never has to explicitly invoke or join threads, nor deal with inter-thread communication. When using pure functions, programs are deterministic and race condition-free and deadlock-free, and we can rely on the spark implementation to make use of multiple processor cores. However, if concurrent features are desired, you will find ample support in various Haskell modules.

HASKELL WRAP UP

In this chapter, we were introduced to Haskell. We learned that:

- Haskell's strength as a language comes from combining three big ideas: pure functions, lazy evaluation, and typeclasses.
- Like Clojure, Haskell greatly facilitates functional programming: functions are first class values, data structures are immutable and persistent, and there is no assignment statement.
- The basic types are `Int`, `Integer`, `Bool`, `Float`, `Double`, and `Char`. New types are created via tuples, functions, and algebraic types.
- Lists and options (known as maybes) are simply algebraic types. Strings are in fact nothing more than lists of characters.
- Curried functions are the norm.
- You can create your own infix operators, defining precedence and associativity.
- Unary operators always bind tighter than binary operators.
- Haskell type inference is based on the Hindley-Milner type system. Type annotations are practically never required.
- Pattern matching and guards appear frequently in conditional execution. There are no loop statements, so recursion is common.
- Types can instance typeclasses, allowing rather elegant expressions of code that requires bounded generics and wildcards in other statically typed languages.
- Effects are strictly isolated into easily identifiable contexts for impure code. The type `IO a` represents an action that will, when executed, produce a value of type `a`. Actions are executed when finally bound to `main`.
- Haskell provides the `Functor`, `Applicative`, and `Monad` typeclasses for enabling very general solutions to "computing in contexts." Commonly used monadic types include `Maybe` for possibly-missing-data, `List` for nondeterministic computation, `Either` for controlling error propagation, and `IO` for controlling side-effects.
- Haskell has many features for concurrent programming. Sparks are used for parallel programming.

For further study in Haskell, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Haskell, see the Haskell page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/haskell.

EXERCISES

Now it's your turn. Continue exploring Haskell with the activities and research questions below, and feel free to branch out on your own.

11.1 Try out several of the examples at an online playground.

11.2 Install Haskell on your own machine and run the three complete programs in the introductory section of this chapter. Find the REPL. Experiment with various kinds of expressions. Generate a few type errors.

11.3 Try using `mapM` instead of `mapM_` in the first program of this chapter. How does the output differ, if at all?

11.4 What is the type of the Haskell expression `\f g x y -> f(g x)y`?

11.5 What is the type of the Haskell expression `2 ^ 5000`?

11.6 Describe the difference between `intercalate` and `intersperse`. Can one be defined in terms of the other?

11.7 Haskell's `concatMap` will look familiar to Java programmers. What does Java call this operator?

11.8 To programmers familiar only with Python, JavaScript, or C-like languages, Haskell's `let` looks like an assignment. But this is not how it would be described in Haskell. Why is it *not* an assignment?

11.9 The composition operator `.` is defined so that $(f\,.\,g)x = f(g(x))$. Is there a built-in operator to compose functions so the left function is applied first? If so, what is it? Whether one exists or not, how would you define it?

11.10 We did not cover the operators `<*` or `*>`. What do they do? How are they defined?

11.11 Suppose you wished to fold the `++` operator over a list of lists. Does it matter if you fold from the left or the right? Why or why not?

11.12 Research some of the scenarios in which the unit type `()` is used in practice.

11.13 What is the curried version of the function `\(x,(y,z)) -> z - y * x`?

11.14 See if you can redefine the precedence of the `+` operator to have higher precedence than multiplication. Were you able to do so?

11.15 Recall the example from this chapter in which we extracted the first ten elements from an "infinite sequence" of primes. Although Python, JavaScript, Java, Kotlin, and Clojure do not advertise themselves as lazy languages, you can simulate this behavior in each quite readily. Translate the program into each of these five languages.

11.16 Locate a description of the function known as the Y Combinator. Can this function be defined in Haskell? Why or why not? Can it be defined in Clojure? Why or why not?

11.17 Read an article on the Hindley-Milner type system and give an explanation of unification.

11.18 Research and describe the typeclasses `Monoid`, `Semigroup`, and `Traversable`.

11.19 (Mini-project) Build your own set type backed by a binary search tree, defined as an algebraic data types.

11.20 Define an algebraic datatype for a user, where users can be anonymous "guests" or logged-in users with names and credentials.

11.21 Do a side-by-side comparison of Java optionals and Haskell maybes. Be as extensive as possible.

11.22 Research how to define your own typeclass

11.23 Research higher-kinded polymorphism [42, Ch. 18] and Rank-N types [44].

11.24 Given

```
data City = City
    { cityName :: String
    , population :: Maybe Integer
    } deriving (Eq, Show)
data Address = Address
    { postalCode :: String
    , city :: Maybe City
    , indicator :: Maybe String
    } deriving (Eq, Show)
data Person = Person
    { name :: String
    , address :: Maybe Address
    , supervisor :: Maybe Person
    } deriving (Eq, Show)
```

Write the function `bossOfBossOfBoss: Person -> Maybe Person` to return the supervisor of the supervisor of the supervisor of its person argument, or `Nothing` if appropriate.

11.25 Given the declarations in the previous problem, show the expression for getting the population of the city of a person.

11.26 Define a type alias (something not covered in this chapter) called `Percentage` (aliasing the `Double` type), and an `Employee` data type containing, among other fields, a name field and a salary field. Write a function called `giveRaise` of type `Person -> Percentage -> Person`.

11.27 Research how random values are produced in Haskell and write a command line-based number guessing game. The program begins by generating a random integer in the range 1..100, and repeatedly prompts the player to enter a value. If the player guesses properly, the program ends with a congratulatory message; otherwise, it reports whether the guess was too low or too high, and asks again.

11.28 The operators `>>=` and `>>` are not the only ways to compose monadic operations. `mapM` and `mapM_` do so as well. Are there any others that are built-in?

11.29 Explain how this script works:

```
main = mapM_ (putStrLn . show) [1..10]
```

11.30 The `do` block syntax, introduced in the Effects section, applies not just to IO but to any monad. Rewrite the boss-of-boss-of-boss and the population-of-city-person solutions above using the do-syntax.

11.31 We visited `Promise`s in our JavaScript chapter. Show how these are monads. (You need only show how the equivalents of `pure` and `bind` behave.)

11.32 In Haskell, monads are applicatives and applicatives are functors. This means any type you define to instance `Monad` must implement `>>=`, `pure`, `<*>`, and `<$>`. See if you can define your own type that instances `Monad` for which you *only* implement `pure` and `>>=`. Will Haskell automatically generate `<$>` and `<*>` for you?

11.33 Experiment with the sparked Fibonacci program from this chapter on a multicore machine. Run the program several times with different settings. Next, try alternate versions using the sub-optimal `par a (a+b)` and `par a (b+a)`. Record your findings over many runs. If you have GHC, you can run the script with `./fib +RTS -N8 -s -RTS` to get detailed information on the number of sparks created, converted, overflowed, and fizzled.

11.34 Write a sparked version of a program that produces the nth Tribonacci (yes, this is a thing) number, where n is a command line argument.

CHAPTER 12

Go

Go is "an open-source programming language that makes it easy to build simple, reliable, and efficient software." [29]

First appeared	2009
Designers	Robert Griesemer, Rob Pike, Ken Thompson
Recognized for	Being simpler than C++, Goroutines
Notable uses	Servers
Tags	Statically Typed, Concurrent, Channel-Oriented
Six words or less	Google's language for Google-sized problems

Go was conceived, designed, and first implemented at Google to help address Google-sized problems. It powers large, scalable, distributed systems running on thousands of machines. In addition to expressive constructs expected in a high-level language, Go also offers some lower-level features such as pointers and capacity settings on slices. Implementations are known for lightning fast compilation and lightweight concurrent processes.

Like Java and Kotlin, Go demands its variables be type-constrained but many constraints can be inferred. You create concurrent programs not with threads, semaphores, polling loops, or other low-level constructs, but rather with **goroutines**, **channels**, and a powerful `select` statement. Go actually feels dynamic: types conform to **interfaces** not via explicit `implements` clauses, but simply by defining the stated methods. You have easy access to run-time type information when needed.

Go's designers strove to keep the language simple, omitting not only threads but implementation inheritance, conditional expressions, and operator overloading. Rather than traditional exception handling, operations that can fail return two values, one of which signals an error (if present). You get safety, too: simply include **deferred operations** to ensure resources are cleaned up even during panics.

Our tour begins with a look at our three traditional scripts and coverage of the basics. We'll then explore a few interesting features—pointers, arrays and slices, interfaces, and panics—and conclude with a dive into Go's signature mechanisms for concurrent programming, goroutines and channels.

12.1 HELLO GO

Here we..., no.

```
package main

import "fmt"

func main() {
    for c := 1; c <= 40; c++ {
        for b := 1; b < c; b++ {
            for a := 1; a < b; a++ {
                if a * a + b * b == c * c {
                    fmt.Printf("%d, %d, %d\n", a, b, c)
                }
            }
        }
    }
}
```

The entry point for a Go program is the function `main` within package `main`. The `Printf` function from the package `fmt` works like that of C; its name is capitalized because capitalized entities are exported from a package while others are private to the package—convention over configuration! The `package` declaration *is* required, as Go will not assume a default package. And like Python but unlike Java, you must explicitly `import` a package in order to use entities from it.

Next we visit our familiar command line argument permutations teaser, again using Heap's algorithm:

```
package main

import (
    "fmt"
    "os"
    "strings"
)

func printPermutations(a []string, n int) {
    if n <= 0 {
        fmt.Println(strings.Join(a, "\t"))
    } else {
        for i := 0; i < n; i++ {
            printPermutations(a, n-1)
            if n % 2 == 0 {
                a[0], a[n] = a[n], a[0]
            } else {
                a[i], a[n] = a[n], a[i]
            }
        }
```

```
        printPermutations(a, n-1)
    }
}

func main() {
    args := os.Args[1:]
    printPermutations(args, len(args)-1)
}
```

`Split` and `Join` are functions from the `strings` package, rather than methods on string objects. The `os` package provides access to command line arguments. We must declare types for parameters, but the types of local variables (like `i` and `args`) can be inferred. The type expression `[]string` reads as "slice of strings." The `:=` symbol introduces Go's **short variable declaration**. We see a verbose `if`-statement because Go lacks a conditional expression. Thankfully, parallel assignment keeps the script from getting too long!

Now let's see our standard data science example in Go:

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "sort"
    "strconv"
    "strings"
)

type Player struct {
    name string
    team string
    ppg  float64
}

func main() {
    players := make([]Player, 0)
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        line := strings.Split(scanner.Text(), ",")
        if len(line) != 4 {
            log.Fatal("Malformed line")
        }
        games, err := strconv.Atoi(line[2])
        if err != nil {
            log.Fatal("Games value is malformed or out of range")
        }
        points, err := strconv.Atoi(line[3])
        if err != nil {
```

```
            log.Fatal("Points value is malformed or out of range")
        }
        if games >= 15 {
            ppg := float64(points) / float64(games)
            player := Player{line[1], line[0], ppg}
            players = append(players, player)
        }
    }
    sort.Slice(players, func(i, j int) bool {
        return players[i].ppg > players[j].ppg
    })
    for _, player := range players[:10] {
        format := "%-22s%-4s%8.2f\n"
        fmt.Printf(format, player.name, player.team, player.ppg)
    }
}
```

Our script begins by defining a **slice** (Go's term for a dynamically allocated and managed sequence of values) to hold the player data (initially set to hold 0 players) and a scanner to read standard input line by line. We must explicitly convert strings to numbers; here `Atoi` is the way to go. As is common practice for operations that may fail, `Atoi` returns two values: (1) the converted integer and (2) an error value, which is `nil` if the operation succeeded. As this is a simple script with only a `main` function, we're failing hard on any error: `log.Fatal` exits after logging a message. The program also illustrates some nice Go features: the use of simple structure types in lieu of classes, a sorting function with an anonymous function for a custom comparator, and the way in which for-loop iteration over slices provides both the index value and the slice element.

12.2 THE BASICS

Go predeclares a few boolean, numeric, and string types:

- The type `bool` with values `true` and `false`.
- Numeric types with explicit bit sizes: the signed integer types `int8` (and its alias `byte`), `int16`, `int32`, and `int64`; the unsigned integer types `uint8`, `uint16`, `uint32`, and `uint64`; the floating point types `float32` and `float64`; and the complex types `complex64` (made from two `float32` values) and `complex128` (from two `float64`s). Go does not have a character type: you process characters by manipulating their code points, normally referred to as **runes**. The type `rune` aliases `int32`.
- Numeric types with implementation-specific sizes: `int`, `uint`, and `uintptr`, designed to correspond to the underlying machine's native data and address bit sizes. These *do not* alias any other type.[1]
- The type `string`, for sequences of bytes—not runes. The operation `len(s)` produces the byte length, not the rune length. The byte sequence of a string *s* representing text will generally be a UTF-8 byte sequence, from which you can obtain the sequence of code points (as a rune slice) via the expression `[]rune(s)`.

[1] Specifically, `int64` and `int` are distinct types, even on a 64-bit machine.

When declaring a variable without an initial value, Go assigns the **zero value** for the type: `0` or `0.0` for numbers, `false` for booleans, and `""` for strings.

The following script illustrates several of the basic types in action.[2]

```
package main

import "fmt"

func ExampleSimpleTypes() {
    var a uint8 = 0x22
    var b int64 = -9377888
    var c rune = '\u03c0'
    var d string
    var e bool
    var f complex128 = 2.5-3i

    fmt.Println(a, b, c, d, e, f)
    // Output: 34 -9377888 960  false (2.5-3i)
}
```

Go has eight kinds of composite types: arrays, functions, structures, maps, pointers, slices, interfaces, and channels. Array types, interestingly, depend on their size: the types `[10]string` and `[20]string` are distinct.

```
package main

import "fmt"

func ExampleArrays() {
    var flags [3]bool
    powers := [5]int{1, 2, 4, 8, 16}
    identity := [2][2]float64{{1,0},{0,1}}

    fmt.Println(flags, powers, identity)
    // Output: [false false false] [1 2 4 8 16] [[1 0] [0 1]]
}
```

Structs work pretty much as expected. Specify struct literals using (1) only the values in order of appearance in the type declaration, or (2) field names and values, in any order. If you omit a field, Go gives you the zero value for the type.

```
package main

import "fmt"
```

[2]Many of the examples in this chapter will use Go's *Runnable Examples* feature. Store the code in a file whose name ends with `_test.go` and run with `go test`, which will find and execute functions whose names begin with `Example`. The output of these functions will be captured and compared with text in comments beginning with `Output:`. Unlike the other languages featured in this book, Go does not have a built-in assertion facility.

```
type Point struct {
    X, Y float64
    Color string
}

func ExampleStructs() {
    p := Point{8, 5, "green"}
    q := Point{Color: "blue", Y: -2.0}
    var r Point
    s := struct{Name string; Ok bool}{"ABC", true}
    fmt.Println(p, q, r, s, p.Y)
    // Output: {8 5 green} {0 -2 blue} {0 0 } {ABC true} 5
}
```

As with other identifiers, capitalized field names will be automatically exported from the current package, while other field names will stay private unless explicitly exported.

The previous two examples show that for both arrays and structs, the "zero-value" (the value assigned to a variable that is not explicitly initialized) is an object filled with zero values of each element or field. In other words, every array or structure takes up as much space as necessary, and is copied when assigned, passed as an argument, or returned from functions. The other six composite types—functions, maps, slices, pointers, channels, and interfaces—each have a zero value called `nil`. There are no optional types in Go: the static type checker will permit you to assign the value `nil` to a map variable, for example, but reading from a `nil` map is a run-time error. Go's static types and dynamic types are not in sync: be aware of the need for run-time checking!

Go's functions are first-class. When specifying a function value, you must mention the types of the parameters and return type:

```
package main

import (
    "math"
    "fmt"
)

func twice(f func(float64)float64, x float64) float64 {
    return f(f(x))
}

func square(x float64) float64 {
    return x * x
}

func ExampleFunctions() {
    addTwo := func(x float64)float64 {return x + 2}
    fmt.Println(twice(square, 4))
    fmt.Println(twice(addTwo, 100))
    fmt.Println(twice(math.Log2, 256))
```

```
    // Output: 256
    // 104
    // 3
}
```

Functions can have more than one return value. This isn't the same as returning a value of a tuple type; Go functions actually do produce multiple result values. Additionally, you may give names to return values; these act as local variables to which you may assign values and have them returned through a simple return statement:

```
package main

import "fmt"

func multipleReturnValues() (bool, int) {
    return true, 5
}

func namedReturnValues(x int) (a int, b string, c float64) {
    a = 8 * x
    b = "ok"
    return        // note no expressions after `return`
}

func ExampleWithMoreFunctions() {
    p, q := multipleReturnValues()
    r, s, t := namedReturnValues(10)
    fmt.Println(p, q, r, s, t)
    // Output: true 5 80 ok 0
}
```

Map types have the form `map[`K`]`V for key type K and value type V. Reading a map returns not an option value, but two values: the value read (or the zero value if the key is not present) followed by a boolean indicating whether the key was present.

```
package main

import "fmt"

type Counts map[string]int

func ExampleMaps() {
    m := Counts{"A": 2, "B": 3, "C": 10, "D": 5}
    m["E"] = 8              // add key-value pair
    delete(m, "D")          // remove key-value pair
    b, hasB := m["B"]       // returns value, wasPresentFlag
    z, hasZ := m["Z"]
    fmt.Println(m["A"], len(m), b, hasB, z, hasZ)
    // Output: 2 4 3 true 0 false
}
```

When invoking operations such as reading from maps that return both a value and a present indicator, as well as similar operations returning a value and an error indicator, you will often take advantage of Go's ability to create variables local to an if-statement just prior to its test. The following shows the idiomatic way to handle both the present and nonpresent cases for keys in a map:

```
package main

import "fmt"

func Example() {
    capitals := map[string]string{
        "Netherlands":  "Amsterdam",
        "Aruba":        "Oranjestad",
        "Curaçao":      "Willemstad",
        "Sint Maarten": "Philipsburg",
    }
    for _, country := range [2]string{"Aruba", "Wales"} {
        if capital, known := capitals[country]; known {
            fmt.Printf("The capital of %s is %s\n", country, capital)
        } else {
            fmt.Printf("I don't know the capital of %s\n", country)
        }
    }
    // Output:
    // The capital of Aruba is Oranjestad
    // I don't know the capital of Wales
}
```

We'll postpone discussion of pointers, slices, interfaces, and channels to future sections, and close our tour of the basics of Go with a look at its statements. Go features a curly-brace syntax without too many noisy parentheses or semicolons. While semicolons terminate most statements, the compiler will insert them for you when the last token on a line "looks like" the end of a statement.[3] You do need semicolons to separate multiple statements on a line (Ruby and Python require this, too), though you will rarely if ever encounter them in actual applications.

Go's statements, which we'll only list for now, are as follows:

- Declarations, of types, variables, constants, functions, and methods.
- Empty statements, which do nothing.
- Expression statements, which execute expressions, only for their side effects.
- Assignments, such as `x = 3` or `x++`. Assignments are not expressions in Go.
- The usual compound statements—`for`, `if`, `switch`—and control transfer statements `break`, `continue`, `fallthrough`, `return`, and `goto`.

[3] In particular, Go inserts a semicolon when a line ends with an identifier (including type names); number, string, or other literals; a right parenthesis or right brace; a `++` or `--` operator; or one of the keywords `return`, `break`, `continue`, or `fallthrough`.

- Defer statements, which schedule a function call to be executed at the time the surrounding function ends (normally or when panicking).
- Concurrency-related statements: `go` to launch a goroutine, a statement to send a value to a channel, and `select`, for coordinating sends and receives. We'll cover concurrency in Section 12.7.

Did the designers miss a `while` statement? Not at all. Look at what the `for` statement can do:

- `for {`*body*`}` — executes body "forever."
- `for` *condition* `{`*body*`}` — executes *body* while *condition* is true.
- `for` *init* `;` *condition* `;` *after* `{`*body*`}` — functions as the classic for-statement in Java, JavaScript, and the C family of languages.
- `for` x `:= range` e `{`*body*`}` — iterates through the indices of array, slice, or string e; or through the keys of map e, or the elements of receive channel e.
- `for` i`,` x `:= range` e `{`*body*`}` — iterates with the index *and* value (or key and value, for maps) of array, slice, string,[4] or map e. Use `_` for the index variable to iterate *only* with the values.

Any statement may be labeled, allowing it to be referenced by a `break`, `continue`, or `goto`. We've seen `break` and `continue` before, but what is a `goto`? *The Go Programming Language Specification* states: "A `goto` statement transfers control to the statement with the corresponding label within the same function" but prohibits the transfer from causing "any variables to come into scope that were not already in scope at the point of the `goto`." [31] In other words, a `goto` mustn't leave its function nor jump over the declaration of any new variables. Long ago, in other languages, gotos were not so restrictive. This prompted Dijkstra to write, in his famous 1968 letter, *Go To Statement Considered Harmful*:

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).
>
> ...The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use [22].

"Bridling" unrestricted control flow means that program flow should be expressed wholly in terms of high level structured statements such as `if` and `for` (the genesis of the term **structured programming**), and perhaps the limited control transfers through `break` and `continue`. Dijkstra's letter was quite influential: `goto` statements are now quite rare, and languages that permit them tend to place significant restrictions on their use.

This concludes our brief coverage of the basics. We now turn to some of the characteristic features of the language, beginning with something new: pointers.

[4] Iterating through the values of a string produces the string's runes, not its bytes.

12.3 POINTERS

As mentioned above, Go *copies* arrays and structs when assigning or passing as parameters. Copying the entire array or struct can carry a significant performance overhead, compared to the assignment and passing of references. If you prefer to deal with arrays and structs through references (as is done automatically for maps, functions, interfaces, slices, and channels), you *can*, but you must do so *explicitly*. Go's explicit references are called **pointers**.

Pointer values have types. The type of pointers that reference objects of type t is denoted $*t$. To make a pointer to an object x, write $\&x$; to **dereference** a pointer p, write $*p$. Figure 12.1 should help to clarify.

Figure 12.1 Pointers in Go

Passing pointers as arguments gives us efficiency, but for better or worse allows mutations through parameters to affect the corresponding arguments. This is perhaps not surprising, as we've seen the same effect for previous languages on our tour which implicitly store references to arrays and structures. Mutating shared data through distinct references is always an issue for programmers to be congnizant of; Go differs from other languages in that, for arrays and structs at least, the existence of a reference is explicit:

```
package main

import "fmt"

func mutateThroughPointer(a *[5]int, b *struct{x, y int}) {
    (*a)[0] = 10
    (*b).x = 5
}

func ExampleMutation() {
    z := [5]int{0, 0, 0, 0, 0}
    p := struct{x, y int}{1, 2}
    mutateThroughPointer(&z, &p)
    fmt.Println(z, p)
    // Output: [10 0 0 0 0] {5 2}
}
```

You need not use pointers *only* to avoid making copies. You will find them extremely useful for creating and modifying arrays and structs at runtime—in other words, for building dynamic data structures. Let's go through a common exercise familiar to most students of programming: writing a type for binary trees:

```go
package binarytree

import (
    "fmt"
    "golang.org/x/exp/constraints"
)

type Tree[T constraints.Ordered] struct {
    Value T
    Left, Right *Tree[T]
}

func (tree *Tree[T]) Size() (count int) {
    if tree == nil {
        return 0
    }
    return 1 + tree.Left.Size() + tree.Right.Size()
}

func (tree *Tree[T]) String() string {
    if tree == nil {
        return "()"
    }
    return fmt.Sprintf("(%s%v%s)", tree.Left, tree.Value, tree.Right)
}
```

We snuck in a few things in this brief example. First, we've introduced Go's support for generics. Our tree type is parameterized by its element type T, which itself has a type `Ordered`. This is similar to Haskell's `Ord` typeclass and Java's `<? extends Comparable>` generic constraints we saw in previous chapters. Second, we wrote `tree.Left` rather than `(*tree).Left`. That's right: field access in Go *automatically dereferences pointers*.[5] The same is true for arrays; we can, and do, write `a[0]` for `(*a)[0]`. Third, we placed, in the function definitions of `Size` and `String`, a parameter *before* the function name. This is how Go defines **methods**. The early parameter is the receiver. As we've seen many times already, method call syntax uses prefix notation, e.g. `tree.Size()`. Finally, we've introduced the value `nil`. Every pointer type has a value `nil`. A `nil` pointer doesn't point to anything, dereferencing `nil` causes a run-time crash. The Billion Dollar Mistake is alive and well in Go.

Now let's take a look at how we use our trees, via the following test:

```go
package binarytree

import "fmt"

func ExampleTree() {
    s := &Tree[int]{
```

[5] Interestingly, automatic dereferencing is not that uncommon. Languages like Ada and Rust also use this feature.

```
        7,
        &Tree[int]{Value:2},
        &Tree[int]{8, &Tree[int]{Value:1}, nil},
    }
    fmt.Println(s)
    fmt.Println(s.Size())
    // Output: (((()2())7((()1())8()))
    // 4
}
```

So we see that an empty tree is simply `nil`, and a one-element tree can be written as either `&Tree{Value:1}` or `&Tree{1,nil,nil}` And calling `fmt.Println` executed the `String` method! How? Keep that in mind when we get to the section on interfaces.

The mechanisms of assigning and passing pointers to arrays and structs do not differ from the languages of the previous ten chapters. The pointers are copied while the referents live somewhere in a shared memory space visible to both the caller and the callee. And at some point a **garbage collector** will come by and free the memory storing the referent when it is no longer used. What Go does offer that the other languages don't, however, is the ability to reference any object whatsoever—even simple numbers, booleans, and strings, regardless of whether they are held in local variables, global variables, or as components of a larger object. This flexible state of affairs might get a little tricky, but responsible software engineers should know the basics of entity lifetimes and memory management, so let's get technical for a minute.

Every call of a function creates an **activation record**, or **frame**, to hold parameters, local variables, temporary scratch space for computation, and other assorted information. Go allocates frames on function entry and deallocates them when the function returns. The active frames thus behave like a (last-in, first-out) stack; in fact, we call the memory holding the frames *the* **stack**. However, when we create pointers to local variables, or local expressions, and pass those pointers *out of the function* via assignment to nonlocal variables or via a return statement, the referents now have a **lifetime** exceeding that of the function creating them. We say that these long-lived referenced entities have **escaped** the function. Because the runtime system deallocates the frame when a function returns, these escaped entities should *not* be allocated inside the frame, otherwise the pointer would be left referencing deallocated memory, in technical terms, it would **dangle**. The official language FAQ [30] states:

> When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors.

An example will help. In the following, the function assigned to f defines three local variables and an anonymous struct. The function makes a pointer to the local variable b and assigns it to a nonlocal variable. It also makes a pointer to the struct and returns the pointer from the function. The Go compiler detects both of these cases during **escape analysis**. It allocates b and the struct on the heap. When f finishes execution, its frame (including locals a and c) go away, but b and the struct survive on the heap. These will be deallocated at some later time by the garbage collector.

```go
package main

import "fmt"

type Point struct{x, y int}

func ExampleEscape() {

    var g *int                      // g means "global(ish)"

    f := func(x int) *Point {
        a := 5                      // stays local
        b := 8
        c := &b
        g = c                       // b escapes!
        return &Point{x, a * 5}     // this escapes too!
    }

    fmt.Print(*f(2))
    fmt.Println(*g)
    // Output: {2 25}8
}
```

Figure 12.2 provides a useful visual for this scenario.

Figure 12.2 Escaped objects allocated on the heap

It's worth noting that arrays, structs, and pointers are copied on assignment and argument passing. Maps, functions, and other complex objects are *implicitly* passed by reference, so Go maps are like maps in nearly every language. In fact `nil` is a legal value for a map. Nil maps differ from an explicitly allocated zero-element map in that nil maps are immutable.

12.4 SLICES

We've just seen how to build a dynamic collection of structs wired together with pointers. But what about dynamic arrays? If an array's length is part of its type (i.e., `[10]int`), how can we write a function that accepts an integer array of any size? Go's notion of the array-of-any-size is a **slice**. A slice is a range of elements from an underlying physical array.

Figure 12.3 shows an array a and slices $s =$ `a[2:8]` and $t =$ `a[6:9]`. We form slices with the inclusive start index of the array and the exclusive end index. The slices themselves contain (1) a pointer to the starting element, (2) the number of elements in the slice, `len`, and (3) the capacity of the slice, `cap`, derived from the length of the underlying array.[6]

Figure 12.3 Go slices

Because slices contain pointers, they share storage with the underlying array and any slices that happen to overlap. For example, in Figure 12.3, `s[4]`, `t[0]`, and `a[6]` are aliases of each other.[7] And like a pointer, a slice can have the value `nil`. A nil slice has zero length and zero capacity...and is the zero value of slice variables.

In practice, we generally construct slices via **slice literals** or the built-in function `make`. Evaluating the slice literal `[]int{1, 1, 2, 3, 5, 8, 13}` builds a 7-element array and a slice covering the entire array (a slice of length 7 and capacity 7). Evaluating `make([]int, 5, 8)` creates an 8-element array, and a slice of length 5 (and capacity 8) overlaying indexes 0 through 4 of the array. You can also write `make([]T, n)` to make a new slice whose length and capacity are both n. And as you'd probably expect, `make` will initialize the slice values to the proper zero value.

You can add elements to the end of a slice with the built-in function `append`, which will continue to use the underlying array if the capacity exists, otherwise a new array will be allocated and the elements copied over prior to appending. If you'd like to copy elements between slices yourself, use the built-in `copy` function.

[6] The capacity is the length of the underlying array minus the start index of the slice plus one.

[7] Go shares slice syntax with Python, including the shorthands `a[:n]` for `a[0:n]` and `a[k:]` for `a[k:len(a)]`.

This is a lot to take in, so we'll provide an annotated example to close out this section.

```
package main

import "fmt"

func Example() {
    a := [5]int{0, 10, 20, 30, 40}
    s := a[1:3]                          // overlaid on a
    fmt.Println(s, len(s), cap(s))
    s = append(s, 700, 900)              // append (still in a)
    fmt.Println(s, len(s), cap(s))
    s = append(s, 5, 5, 5)               // append, overflows
    fmt.Println(s, len(s), cap(s) > 5)   // note capacity grown
    t := make([]int, 10)                 // new 10-element slice
    copy(t, s)                           // copy elements to new location
    fmt.Println(t)                       // note the copy
    // Output: [10 20] 2 4
    // [10 20 700 900] 4 4
    // [10 20 700 900 5 5 5] 7 true
    // [10 20 700 900 5 5 5 0 0 0]
}
```

12.5 INTERFACES

Let's now turn to the Go way of specifying types of "objects that know how to do x and y," namely, the **interface**. We'll start with examples. A runner is something that runs:

```
type Runner interface {
    run()
}
```

A finder is anything that looks for a pattern in a source, return the position at which it was found, and a boolean stating whether it was found:

```
type Finder interface {
    find(pattern string, source string) (int, bool)
}
```

A locker is anything that locks, yielding a key, and that unlocks with a key:

```
type Locker interface {
    Lock() string
    Unlock(string)
}
```

Interfaces are types, so you can declare variables and parameters with interface types. Go will check at compile time that whatever you assign to such a variable or pass to such a parameter, actually supports those methods. Unlike Java, you need not explicitly state that your implementing type "implements the interface." If your type has all the methods, congratulations, it implements the interface:

```go
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    perimeter() float64
    area() float64
}

type Circle struct {
    radius float64
}

func (c Circle) perimeter() float64 {
    return 2.0 * math.Pi * c.radius
}

func (c Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

type Rectangle struct {
    length, width float64
}

func (r Rectangle) perimeter() float64 {
    return 2.0 * (r.width + r.length)
}

func (r Rectangle) area() float64 {
    return r.width * r.length
}

func showDetails(s Shape) {
    fmt.Printf("%T perimeter=%g area=%g\n", s, s.perimeter(), s.area())
}

func Example() {
    showDetails(Rectangle{5.5, 20.0})
    showDetails(Circle{4})
    // Output: main.Rectangle perimeter=51 area=110
    // main.Circle perimeter=25.132741228718345 area=50.26548245743669
}
```

Let's now see how we can implement our by now familiar animals example. Here several types of animals speak the same way but make their own sounds:

```go
package main

import "fmt"

type Animal struct {
    Sounder
    name string
}

type Sounder interface {
    sound() string
}

func speak(a Animal) string {
    return fmt.Sprintf("%s says %s", a.name, a.sound())
}

type Horse struct {}

func (h Horse) sound() string {
    return "neigh"
}

type Cow struct {}

func (c Cow) sound() string {
    return "moooo"
}

type Sheep struct {}

func (s Sheep) sound() string {
    return "baaaa"
}

func ExampleAnimals() {
    h := Animal{Horse{}, "CJ"}
    fmt.Println(speak(h))
    c := Animal{Cow{}, "Bessie"}
    fmt.Println(speak(c))
    fmt.Println(speak(Animal{Sheep{}, "Little Lamb"}))
    // Output: CJ says neigh
    // Bessie says moooo
    // Little Lamb says baaaa
}
```

Here we've introduced an interface `Sounder` for things making a sound; horses, cows, and sheep will all make sounds and thus satisfy the interface. An `Animal` will be a sounder with a name. Note that the animals CJ, Bessie, and Little Lamb are not created as instances of

the types `Horse`, `Cow`, or `Sheep`. They are instances of `Animal`, but with embedded `Sounder` objects. This mechanism of structuring is akin to considering `Animal` to be a sum types, since there is no syntactic mechanism for inheritance at play here.

Interfaces are ubiquitous in the Go libraries. Here's one from the package `fmt` you should know:

```
type Stringer interface {
    String() string
}
```

This is used within the body of `fmt.Printf`, `fmt.Println`, and similar functions to produce a string representation of a value. We've implemented a `String()` method earlier in the chapter—for binary trees—and you'll likely find yourself implementing this method on your own data types from time to time, much like implementing `toString` in JavaScript or Java, defining `__toString` in Lua, defining `__str__` in Python, or the equivalent in many (most?) other languages we've seen.

The interface

```
type error interface {
    Error() string
}
```

also finds its way into many Go programs. This type is not defined in any package; rather it's a **predeclared identifier**, like `int`, `true`, `nil`, `len`, and `delete`. A common Go idiom, for functions that may either succeed or fail, is to return two values from the function: the first the function's actual result (if successful), and the second an object implementing this interface. The package `encoding/base64`, for example, contains the function

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

Given a string that supposedly represents a Base64 encoding [110] of a byte array, this function will produce either (1) the original string and the value `nil`, indicating, per convention, no error, or (2) some object implementing the `error` interface as the second value, in which case we don't care about the first value:

```
package main

import (
    "encoding/base64"
    "fmt"
)

func Example() {
    strings := []string{"SGVsbG8sIHdvcmxk", "Bad&^)#****Chars"}
    for _, s := range strings {
        if bytes, err := base64.StdEncoding.DecodeString(s); err != nil {
            fmt.Println("Oops:", err.Error())
        } else {
            fmt.Println("Decoded:", string(bytes))
        }
    }
```

```
    // Output:
    // Decoded: Hello, world
    // Oops: illegal base64 data at input byte 3
}
```

Notice that we used the value `nil` in a context where an object of type `error` was expected. This is perfectly fine. `nil` is not just for pointers and maps, as we've already seen, but also can be used for—and plays the role of the zero value for—map, slice, function, interface, and channel values.[8]

Here's a final thought for this chapter. Go's design choice of a flavor of (static) duck typing,[9] namely allowing types to implement interfaces not by mentioning the interface name in the type declaration, but simply by virtue of implementing the interface's method, means that *every* type in Go implements `interface{}` (the empty interface). There is no need to implement a **cosmic supertype**, a type (or class) from which all other types derive, such as Java's `Object` or Ruby's `BasicObject`. `interface{}` is the perfect type for a parameter that truly accepts any value at all. You can find many uses of this type in the source code of the Go Standard Packages. We've seen at least one already:

```
func Println(a ...interface{}) (n int, err error)
```

And now we can officially add Go to the list of languages that can pack a variable number of arguments into a single parameter.

12.6 PANICS

We've seen that idiomatic Go makes heavy use of multiple return values for handling errors that *might* occur—opening or reading from files, formatting data, or decoding streams. After all, it's perfectly reasonable that someone may ask for a nonexistent file to be read, or supply a malformed byte stream to be decoded. These are not really "exceptional conditions," so handling them by throwing and catching exceptions is arguably a misuse of the exception facility. But this kind of programming is practically encouraged by JavaScript, Java, and similar languages, that even allow try-catch-finally blocks right in the middle of a function body.

Go doesn't have exceptions, but it does have a way to handle the truly exceptional cases—the ones that really shouldn't happen or the ones you don't expect to recover from. When something truly exceptional happens, the program—technically, the currently executing goroutine—panics. A panic stops the current function, runs any deferred operations, then returns. In the caller, the panic process is repeated, until some deferred operation is able

[8] This odd choice of including `nil` as a member of a handful of static types goes against the trend followed by modern languages such as Kotlin, Rust, and Swift. Go favors simplicity over a perfectly sound static type system, and is quite happy leaving it to the programmer to be aware of the need for runtime type checking.

[9] The moniker "duck typing" gained popularity in programming languages due to the oft-heard phrase "if it looks like a duck, walks like a duck and quacks like a duck, then it's a duck." Rather than declaring that something *is* a duck (as in the Java-esque `class D implements Duck`), a duck-typed language simply ensures the objects in question can do the things ducks can do, i.e., that they implement the corresponding methods. The term "duck typing" is not precisely defined, however; you may sometimes hear the argument that duck typing applies only to dynamically typed languages. Following this line, we would describe Go as having "structural typing" or "implicitly implemented interfaces." The intent, though, is the same: the interface name does not appear.

to *recover*, or the entire chain of calls has been "unwound" and the program **crashes** (exits with a nonzero return code).

To see the mechanism in action, we've built a little contrived demo in where the function `Example` calls `f` which calls `g`, which panics. `g` will run its deferred operation but will not recover from the panic, so `f` panics too. `f` runs its deferred operation but recovers, and returns normally to `Example`.

```go
package main

import "fmt"

func g() {
    defer fmt.Println("Cleaning up g")
    panic("Oh no")
    fmt.Println("This message will not be displayed")
}

func f () {
    defer func() {
        fmt.Println("Cleaning up f")
        if r := recover(); r != nil {
            fmt.Println("Recovered the panic value", r)
        }
    }()
    g()
    fmt.Println("This message will not be displayed")
}

func Example() {
    defer fmt.Println("Cleaning up main")
    f()
    fmt.Println("f completed normally")
    // Output: Cleaning up g
    // Cleaning up f
    // Recovered the panic value Oh no
    // f completed normally
    // Cleaning up main
}
```

If called during a panic, `recover` returns the value passed to `panic`. Otherwise, if no panic is in progress, `recover` returns `nil`.

The accepted convention in Go libraries is to limit panicking and recovery to the internals of a library, while returning explicit error values to clients of the library. Such a design leads to a simpler interface between callers and services. The convention sound from a software engineering and a security perspective. If an operation can produce a recoverable error, it should provide the error object directly to the caller. If not recoverable, the operation should fail fast and not propagate problems to parts of an application that know nothing about such errors. As software libraries are generally self-contained and isolated from higher-level applications; panic-recover as an internal mechanism can be quite acceptable.

12.7 GOROUTINES

Go may be best known for its approach to making concurrent programming expressive and efficient. **Concurrent programming** is the art of structuring software systems by composing and coordinating independent computations. Concurrency isn't the same as parallelism. Parallelism is the state of things happening at the same time; concurrency is about managing activities that overlap in time, whether or not these activities interleave execution on a single processor or run in parallel on distinct processors [90].

Go's concurrent programming philosophy is based on message passing rather than on shared memory and locks. The concurrent execution units, known as **goroutines**, are just functions, with their own local variables, who communicate with other functions via message passing.[10] Goroutines are similar to Java's virtual threads and Kotlin's coroutines in that they are designed to support tens of thousands of concurrent activities at once. Go differs from other languages we've seen in that its mechanism for sending data between concurrent actors is through **channels** which are a built-in language type, rather than a construct imported from a library. Go was designed from the beginning for concurrent applications.

Let's start with a (literal) hello world example. We will create a channel that can send and receive strings. We'll launch a goroutine that writes `"Hello world"` to the channel. The main goroutine will read the channel and print the value received:

```
package main

import "fmt"

func Example() {
    ch := make(chan string)
    go func() {ch <- "Hello, world"}()
    fmt.Println(<-ch)
    // Output: Hello, world
}
```

Here we see that goroutines are launched by prefixing the `go` keyword to a function call; channels are typed (e.g. the type `chan string` can send and receive strings only); channels are created with `make`; *ch* `<-` *x* writes *x* to channel *ch*; the expression `<-`*ch* reads from channel *ch*. Two more facts about channel communication are essential to know:

- Channels created with `make(chan` *T*`)` are synchronous. A sender will block until a receiver is ready to consume a message; and conversely, a receiver will block until a sender sends a message.
- When the main goroutine finishes, the application finishes, even if other goroutines are still running.

Let's dial up the complexity ever so slightly. Here's a famous example in which two goroutines sum each half of an array—note the use of a slice parameter here to avoid copying the entire array—and write their half-sums to a channel. The main goroutine reads the half-sums from the channel, adds them, and prints the result.

[10] The saying in Go is: "Don't communicate by sharing memory, share memory by communicating." [28]

```go
package main

import "fmt"

func sum(a []int, ch chan<- int) {
    result := 0
    for _, value := range a {
        result += value
    }
    ch <- result
}

func Example() {
    var numbers [100000]int
    for i := range numbers {
        numbers[i] = i
    }
    ch := make(chan int, 2)
    go sum(numbers[:len(numbers)/2], ch)
    go sum(numbers[len(numbers)/2:], ch)
    fmt.Println(<-ch + <-ch)
    // Output: 4999950000
}
```

We've created the channel in this example via `make(chan string, 2)`, giving it a *capacity* of two messages. Channels with capacities are called **buffered channels**: if a channel holds fewer messages than its capacity, senders will not block. The type annotation on `sum`'s *ch* parameter, `chan<- int`, indicates the channel may only send. (There is, of course, a corresponding annotation for receive-only channels: `<-chan int`.)

In addition to reading a single item from a channel using `<-`, we can iterate over the channel with the `range` operator, consuming messages until the channel is closed. In the following example, a goroutine sends 100 messages through a channel, then closes the channel. A loop reads and prints each message, then exits cleanly as the channel was closed:

```go
package main

import "fmt"

func generateMessages(ch chan string, n int) {
    for i := 0; i < n; i++ {
        ch <- "Hello"
    }
    close(ch)
}

func main() {
    messages := make(chan string, 5)
    go generateMessages(messages, 100)
```

```
    for message := range messages {
        fmt.Println(message)
    }
}
```

Reading from a closed channel does not block a goroutine. To detect whether a channel is closed, you can use the two-return-value form of `<-`; if closed the second value will be `false`, and the first will be the zero value of the channel's message type.

```
package main

import "fmt"

func Example() {
    ch := make(chan int)
    go func() {
        ch <- 100
        close(ch)
    }()
    value, ok := <- ch
    fmt.Println(value, ok)  // This will get something
    value, ok = <- ch
    fmt.Println(value, ok)  // This won't, but it won't block!
    // Output: 100 true
    // 0 false
}
```

Many concurrent programming scenarios involve goroutines that read or write multiple channels; reads and writes that should time out; or reads and writes that should back off if the other side is not immediately ready. Go's powerful `select` statement handles all of these concerns. It consists of zero or more *cases*, each of which is a channel read or channel write or (at most one) default case:

```
select {
case v1 := <- c1:
    statement_list_1
case v2, x2 := <- c2:
    statement_list_2
// more cases ...
case cn <- xn:
    statement_list_n
default:                    // The default case is optional
    statement_list_n+1
}
```

Go first evaluates, in order, each of the channel expressions and send-expressions. If none of the communications is ready to proceed and there is no default case, the statement blocks until a case can proceed, or forever if none ever becomes ready. If no case is ready and a default is present, the default statement list executes. If ready cases do exist, Go selects one at random to proceed, and executes the corresponding statement list.

The expression `time.After(d)`, for some `Duration`[11] value d, produces a channel that blocks for the given duration, and when read from, yields the current time. This gives a convenient mechanism for timing out sends and receives—simply include a receive from this channel in a select statement. Either a communication will happen within the duration specified, or the channel will unblock the `select` when the delay expires. We'll see an example momentarily.

Now we're ready to build the Go version of the restaurant simulation we saw in our Java chapter. Recall the scenario: three chefs serve fifteen customers. Customers place their orders in a bounded order queue of capacity 5. A customer unable to get an order placed in 7 seconds leaves for a shopping trip and returns later. Cooks prepare the order and deliver the finished meal directly to the customer. Customers leave after eating 10 meals. When all customers have finished, the simulation ends gracefully.

Let's start with the order type and the global order queue:

```
package main

type Order struct {
    food string
    customer string
    reply chan *Order
    preparedBy string
}

var Orders = make(chan *Order, 5)
```

Each order holds the menu item that was ordered, the name of the customer placing the order, and a channel through which to send the prepared order back to the customer. The global (to the package) variable `Orders` is the queue to which customers will place, and cooks will retrieve, orders. Our channels contain pointers to orders, allowing the orders themselves to be modified.

Cooks are goroutines that never terminate (call these **daemons** if you wish). Not explicitly terminating is not a problem; the entire application will end when the main goroutine finishes. A cook repeatedly retrieves an order from the global queue, prepares the meal, signs it in the `preparedBy` field, and sends the order back through the reply channel:

```
package main

func Cook(name string) {
    report(name, "starting work")
    for {
        order := <-Orders
        do(name, "cooking for " + order.customer, 12E9)
        order.preparedBy = name
        order.reply <- order
    }
}
```

[11] The duration type is defined as `type Duration int64` in the package `time`, and represents the length of time between two instants in nanoseconds. Durations, therefore, have a range of approximately ±292 years.

Our utility function `report` logs a simple message, while the function `do` logs a message and sleeps, simulating an action such as cooking, eating, or shopping. Since multiple cooks and customers will log messages, we require a synchronized logging mechanism so that messages do not become interleaved. Fortunately, Go's `log` package contains a number of functions, including `Println`, that are protected against interleaving.

```
package main

import (
    "time"
    "log"
    "math/rand"
)

func report(name string, message string) {
    log.Println(name + " " + message)
}

func do(name string, message string, maxDelay time.Duration) {
    report(name, message)
    delay := int(maxDelay) / 2 + rand.Intn(int(maxDelay) / 2)
    time.Sleep(time.Duration(delay))
}
```

Before we look at customers, let's look at the main goroutine. We want each of the customers to complete their full lifecycle of ten finished meals before we exit the simulation. A **wait group** comes in handy. A `sync.WaitGroup` supports tree methods: (1) `Add(`n`)` to increase the wait group's counter, (2) `Done()`, decreasing the counter, and (3) `Wait()`, to block until the counter reaches zero. Our main goroutine simply creates each of cooks and customers, incrementing the wait count upon customer creation. The last line of the function just invokes `Wait`, relying on each customer to notify the group by calling `Done` when it finishes.

```
package main

import (
    "fmt"
    "sync"
)

const numberOfCooks, numberOfCustomers = 3, 15

func main() {
    var waitGroup sync.WaitGroup
    for i := 0; i < numberOfCustomers; i++ {
        waitGroup.Add(1)
        go Customer(fmt.Sprintf("customer-%v", i), &waitGroup)
    }
    for i := 0; i < numberOfCooks; i++ {
```

```
        go Cook(fmt.Sprintf("cook-%v", i))
    }
    waitGroup.Wait()
}
```

Now we can define the customer goroutine. Each customer is created with a name and the wait group to notify when finished. We begin by setting up the notification as a deferred operation; this common Go idiom of placing deferred operations for finalization and clean up at the beginning of a goroutine is good practice. Deferred operations are always executed, even if a function prematurely returns or panics; in our case we know customers will always notify the wait group no matter how they terminate.

The customer then loops until ten meals have been consumed. Each loop begins by placing a new order. We place the order in a `select` statement with a seven-second time out. Each order will contain the customer's personal channel for receiving prepared meals. During each iteration, the customer either eats a meal or gives up on placing the order. After eating all the meals, the customer reports being finished, and the deferred operation notifies the wait group.

```
package main

import (
    "sync"
    "time"
)

func Customer(name string, wg *sync.WaitGroup) {
    defer wg.Done()
    ch := make(chan *Order)
    for mealsEaten := 0; mealsEaten < 10; {
        select {
        case Orders <- &Order{food: "soup", customer: name, reply: ch}:
            meal := <-ch
            do(name, "eating " + meal.food + " prepeared by " +
                meal.preparedBy, 10E9)
            mealsEaten += 1
        case <-time.After(time.Second * 7):
            do(name, "waiting too long, going shopping", 5E9)
        }
    }
    report(name, "going home")
}
```

Admittedly, the fact that a customer will give up if unable to place an order in a given amount of time but will happily wait forever for a meal to be prepared may not be the best model of reality, but it keeps the example simpler than it would have been otherwise, since leaving while a meal is being prepared would force us to deal with throwing away prepared meals (i.e., draining channels).

We managed in this chapter to introduce many of the concurrency primitives available in Go, and looked at a few different usage patterns. Clearly, we only scratched the surface.

A nice next step in your study of Go might be to write a server capable of dispatching goroutines to do work on behalf of clients. Web servers, chat servers, search servers, and load balancers are actually quite fun to write in Go.

One final note: while high-level communication between goroutines via channels will make up the majority of your concurrent programs' architectures, there are times when a channel is overkill and simple lock, condition variable, or atomic counter is best. Go's `sync` and `sync/atomic` packages will likely have what you need. A few end-of-chapter exercises will give you scenarios to employ a few of these objects.

GO WRAP UP

In this chapter, we were introduced to Go. We learned that:

- Go was designed for large-scale concurrent applications.
- Go is statically-typed with a fair amount of type inference (e.g., in the short variable declaration), though type annotations are required on function parameters and return types.
- Go defines distinct types for integers of various bit sizes and signedness, as well as distinct types for floating point values of different sizes.
- The string type is defined as a byte sequence, not a character sequence. To get code point sequences, you can cast the string to a rune slice.
- Variables are never really uninitialized in Go; if no initial value is provided in a declaration, the variable gets the zero-value of the type.
- Go features eight kinds of composite types: array, function, struct, map, pointer, slice, interface, and channel.
- The length of the array is part of its type, so slices, which can be of varying length, tend to be used more often. Slices are defined over a backing array.
- Functions can have more than one return value. This fact is often used to simulate option types and result types from other languages.
- Go's `for` statement includes functionality of the `while` statement in other languages. The `if` and `switch` statements can define local variables that can be used in these statements' conditions. Go also contains the interesting `goto` statement, though with limited capabilities compared to the unrestricted `goto` much maligned throughout the years.
- Go is the first language in this text where pointers (references) are explicitly marked in the syntax. We can refer to the reference and the referent separately. Pointers are used where sharing is required, and sometimes simply as a convenience to avoid copying arrays and structs.
- Go performs escape analysis during compilation to determine whether pointers to local variables or temporary expressions will outlive the function activation in which they were defined. If so, the Go runtime will ensure the corresponding referents will be heap-allocated.
- In Go, objects satisfy interfaces simply by virtue of defining the methods in the interface. There is no need for an explicit `implements` clause as in Java.

- Go does not have exceptions, but there is a panic and recover mechanism for situations where unexpected or semi-expected problems occur for which recovery is possible.
- Concurrency has a great deal in-language support. Both `go` and `select` are statements, not library calls, channels are instances of a built-in (parameterized) data type, and message send and receive have special syntax.
- Go's concurrent execution units are called goroutines, and many Go implementations support tens or hundreds of thousands of concurrently executing goroutines.
- Senders block on empty channels and receivers block on full channels. A channel with no capacity models synchronous communication where a sender and receiver must wait for each other. Channels with a nonzero capacity model asynchronous communication.
- Senders and receivers can both cancel sends and waits via a timeout mechanism, modeled in Go with a special channel.
- Senders and receivers can both cancel sends and waits if unable to communicate immediately, via a `default` case in the `select` statement.
- In addition to channel-based communication, Go supports wait groups and other synchronization mechanisms.

For further study in Go, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Go, see the Go page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/go.

EXERCISES

Now it's your turn. Continue exploring Go with the activities and research questions below, and feel free to branch out on your own.

12.1 Locate the Go Playground online and try out several of the standalone scripts from this chapter.

12.2 Install the standard Go distribution on your machine. Experiment running scripts (with `go run`) and tests (with `go test`). Explore other uses of the `go` command.

12.3 Find (online) and browse *A Tour of Go*, *Effective Go*, the official FAQ, and *The Go Programming Language Specification*.

12.4 The word count program early in this chapter does not check for reading errors. Make it more robust.

12.5 Write a Go function that returns the identity matrix of a given size, i.e., a call to `identity(3)` should produce `[3][3]float{{1,0,0},{0,1,0},{0,0,1}}`.

12.6 Learn about authoring your own packages in Go. Create a package that includes (1) a type declaration (call the type `Predicate`) for functions that take a single argument of any type and return a boolean result, and (2) functions called `All` and `Any` that accept a predicate p and a slice s, returning whether $p(s_i)$ is true for all or any values of s, respectively.

12.7 Go has a `switch` statement that we did not cover. Research how it works (there are two flavors of this statement) and generate a working example. If you are familiar

with switches in C, Java, and JavaScript, what do you find to be the most glaring difference between Go's switch and those of older languages?

12.8 Read Dijkstra's famous Goto letter [22]. Also look for and read the "rebuttal" article by Rubin. What do you make of Rubin's counterarguments? Do they even apply in modern languages?

12.9 Modify the binary tree example of this chapter to define binary *search* trees. Include a rich set of methods, including those to add and delete nodes, compute the height, and traverse the tree in various orders. Your traversal methods should write to a supplied channel.

12.10 See if you can find the language designers' rationale for making arrays and structs value objects requiring explicit pointer objects to reference them, while maps are always accessed through an implicit reference.

12.11 Perform experiments to discover how your Go implementation increases the capacity of a slice after `Append` operations overflow the existing capacity.

12.12 Research the literature—and the blogosphere!—regarding duck typing. Is there any consensus on whether Go is duck typed? What do you think?

12.13 How would you simulate Java's default interfaces in Go?

12.14 Read and summarize the discussion of Go's `nil` at https://go101.org/article/nil.html. Is `nil` a value? Does it have a type? Or is it something else entirely?

12.15 Write and execute a program that panics without recovery. What was the operating system's return code for the execution?

12.16 The following example is a slight adaptation of code given by Rob Pike in [90]. Give a detailed explanation of its behavior.

```
package main

import "fmt"

func generate(first chan<- int) {
    for i := 2; ; i++ {
        first <- i
    }
}

func filter(in <-chan int, out chan<- int, prime int) {
    for {
        candidate := <-in
        if candidate % prime != 0 {
            out <- candidate
        }
    }
}

func main() {
```

```
    ch := make(chan int)
    go generate(ch)
    for i := 0; i < 1000; i++ {
        prime := <-ch
        fmt.Println(prime)
        nextCh := make(chan int)
        go filter(ch, nextCh, prime)
        ch = nextCh
    }
}
```

12.17 Reimplement the word count example from the first section of this chapter as a concurrent application. Read the input file with a goroutine and write the parsed and lowercased words to a channel. Process the words in the main goroutine.

12.18 Here's a common pattern:

```
select {
case service <- value:
    doSomething(value)
case <-quit:
    return
}
```

What is this pattern "saying?"

12.19 Perform a study to see how well your Go implementation takes advantage of your (hopefully) multiprocessor or multicore system. First, find the value of `runtime.NumCPU()` and make sure the value of `runtime.GOMAXPROCS` (don't forget to research the descriptions of these functions) is set correctly. Compare the runtimes of summing a massive array (1) split into $NumCPU$ pieces summed concurrently by goroutines, with (2) a single-goroutine sequential sum of the entire array. How close was the speedup to the number of CPUs? How do you explain your results?

12.20 Research the following types from the `sync` package: `Cond`, `Locker`, `Once`, `Mutex`, `RWMutex`, and `Pool`. Write scripts illustrating their use.

12.21 Research the functions from the `sync/atomic` package to get a feel for their capabilities. Look in best practices guides and other literature on the language to see which applications these are best suited for.

12.22 What is the difference between the format specifiers `%v` and `%#v`?

12.23 Write a function that prints the field names and types of a given struct. Would this be considered a "For educational purposes only" exercise? If so, what kind of real-world examples of Go reflection exist, beyond the JSON-marshalling example we encountered in the chapter?

CHAPTER 13

C

C is the quintessential systems programming language. Though dating from 1972, it remains today one of the most popular programming languages in the world.

First appeared	1972
Designed by	Dennis Ritchie at Bell Labs
Recognized for	Systems programming
Notable uses	Unix and Unix-like operating systems
Tags	Imperative, Low-level, System, Nonmanaged
Six words or less	The world's most popular systems language

C was created in 1972 by Dennis Ritchie, and has undergone several revisions (most notably in 1989, 1999, 2011, and 2023). C is a very small language, though with a rich library. When writing in C, the programmer needs to think in machine-specific terms, often specifying exactly how many bytes to allocate for data structures such as the memory buffers used to transfer data to and from files and network streams. These buffers do not grow and shrink automatically, making simple functions using arrays and strings notoriously difficult to write and error-prone. There is no garbage collector. Programmers must explicitly allocate dynamic memory blocks and must explicitly deallocate them when no longer needed. Such work is difficult: one study found that for the last decade or more, 70 percent of security vulnerabilities are due to memory safety issues [16].

But programming close to the machine layer, with full control of the hardware, empowers us to write operating systems, device drivers, and firmware for embedded devices. Extremely efficient applications are in our grasp: most operators and statements map fairly closely to machine instructions.

Despite being insecure by nature, C programs run fast. Very fast. In fact, most of the "high-performance" libraries used by Python are actually written in C, then lightly wrapped with a tiny bit of Python code. The software in higher-level languages that run "on top" of C provides the security, flexibility, and convenience that C lacks. This makes sense: complex systems are built in layers. Newer languages may pose challenges to its popularity, but C isn't going anywhere soon. With this in mind, let's take a look at the most popular systems language on the planet.

13.1 HELLO C

Once again, we begin with our three teaser programs. Triples first:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    for (int c = 1; c <= 40; c++) {
        for (int b = 1; b < c; b++) {
            for (int a = 1; a < b; a++) {
                if (a * a + b * b == c * c) {
                    printf("%d, %d, %d\n", a, b, c);
                }
            }
        }
    }
    return EXIT_SUCCESS;
}
```

The operating system runs your C program by calling the function `main`, which returns an **exit status**. The value `EXIT_SUCCESS`, defined in the *included* system file *stdlib.h*, announces a successful run. As a small language, C lacks range operators for iterating, so we use a three-part for-loop. Formatted printing to standard output happens with `printf`, defined in the system file *stdio.h*. The specifier `%d` formats decimal integers.

Our traditional second teaser prints all permutations of its command line arguments. The length of this program is notable: C is an extremely low-level language. You won't find a string-joining function, and you swap array elements in very tiny steps.

```
#include <stdio.h>
#include <stdlib.h>

void print_string_array(char *a[], int size) {
    if (size == 0) {
        return;
    }
    printf("%s", a[0]);
    for (int i = 1; i < size; i++) {
        printf("\t%s", a[i]);
    }
    printf("\n");
}

void print_permutations(char *a[], int n, int size) {
    if (n == 0) {
        print_string_array(a, size);
    } else {
        for (int i = 0; i < n; i++) {
            print_permutations(a, n - 1, size);
```

```
            int j = n % 2 == 0 ? 0 : i;
            char* old_last = a[n];
            a[n] = a[j];
            a[j] = old_last;
        }
        print_permutations(a, n - 1, size);
    }
}

int main(int argc, char *argv[]) {
    print_permutations(argv + 1, argc - 2, argc - 1);
    return EXIT_SUCCESS;
}
```

The most notable feature in this script may be the way arrays are passed to functions. In C, arrays *decay* into memory addresses when passed—they lose their size! Therefore we must pass two arguments: (1) the (decayed) address of a memory buffer containing our array elements, and (2) the number of elements of this buffer that are significant. The type annotation `char*[]` on the first parameter of each of our first two functions reads as "the starting address of an array of strings." Details will be described later in the chapter. Because command line arguments are also arrays of strings, the `main` function has two parameters: the starting address of the argument array `argv` (argument *values*) and the number of arguments `argc` (argument *count*).

Our third teaser program follows.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char team[8];   // room for 7 characters + the terminator
    char name[64];  // room for 63 characters + the terminator
    double ppg;
} Player;

int ppg_compare(const void *a, const void *b) {
    double ppg_a = ((Player*)a)->ppg;
    double ppg_b = ((Player*)b)->ppg;
    return (ppg_a < ppg_b) ? 1 : (ppg_a > ppg_b) ? -1 : 0;
}

int main() {
    Player players[1000];
    int playerCount = 0;

    char line[256];

    while (fgets(line, sizeof(line), stdin)) {
        if (strlen(line) >= sizeof(line) - 1) {
```

```
            fprintf(stderr, "Error: Line too long\n");
            return -1;
        }

        char team[8], name[64];
        int games, points;
        if (sscanf(line, "%7[^,],%63[^,],%d,%d",
                team, name, &games, &points) != 4) {
            fprintf(stderr, "Error: Invalid input line format\n");
            return -2;
        }
        if (games < 15) continue;
        if (playerCount >= sizeof players / sizeof(Player)) {
            fprintf(stderr, "Error: Too many players\n");
            return -3;
        }
        Player* player = &players[playerCount];
        strlcpy(player->team, team, sizeof player->team);
        strlcpy(player->name, name, sizeof player->name);
        player->ppg = (double)points / games;
        playerCount++;
    }

    qsort(players, playerCount, sizeof(Player), ppg_compare);
    int printedPlayerCount = playerCount < 10 ? playerCount : 10;
    for (int i = 0; i < printedPlayerCount; i++) {
        printf("%-22s%-4s%8.2lf\n", players[i].name, players[i].team,
            players[i].ppg);
    }

    return EXIT_SUCCESS;
}
```

One reason for the surprising length of this program is the amount of string processing required. C was not designed to make strings easy. C programs require you, the programmer, to specify exactly how much memory is needed for each string in your program, and if you don't know, you must either:

- Set hard upper limits on each string, and emit an error (or crash) if guessed too low. The programmer, not the language, is responsible for ensuring these checks are written and that they actually work.
- Create a custom data type that grows and shrinks as needed. To grow, you specifically ask the operating system for some memory; to shrink, you explicitly return the memory. These operations can be encapsulated into functions to keep the code a bit more manageable.

We chose the first option, allocating 8-byte buffers for each team name and 64 for each player name. Since strings have varying size, a convention is needed for recording how many characters within the buffer comprise the actual string. In C, the first zero byte serves as a **string terminator**. Only the bytes prior to the zero are part of the string, so team names

can be at most 7 bytes long and player names at most 63. We also allocate buffers to hold the current line being read, and for the array of player objects on which we compute points per game and sort.

A very large percentage of this script dealt with allocating memory buffers, moving data from and into buffers, and checking that our data actually fit into the buffers. The actual implementation left a bit to be desired: the values 8 and 64 were repeated, and the 7 and 63 inside the `sscanf` format string were dependent on these values. Should we desire the change any of these sizes, several portions of our code would be affected. You'll be gaining skills in managing buffer sizes as you work to clean up this script in one of the end-of-chapter exercises. Before we get to the exercises, though, there is much to explore in this venerable, half-century old, systems programming language.

13.2 THE BASICS

C is a very small language. There are few statements, many of which, including assignment, calls, if, switch, while, and do-while will be familiar to C++, JavaScript, and Java programmers, as these languages copied C's syntax. Some statements don't appear at all: there is no way to throw exceptions, for example.

The basic signed integer types (`signed char`, `short`, `int`, `long`, and `long long`) and unsigned integer types (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`) have implementation-specific sizes, with the requirements that (1) `char`, `signed char`, and `unsigned char` have a size of 1 byte, and (2) each of the types listed above has a size that at least that of the type listed immediately before it. The standard library provides a few integer types with exact sizes if needed. Floating point types include `float`, `double`, and `long double`. Complex and imaginary types also exist for each floating point type. Arrays are sequences of values of the same type, indexed from zero. Structs are objects with named fields. Unions are similar, but the storage of each "field" overlaps, giving us a sum type.

Let's get right to the most characteristic aspect of C, obtaining the **address** of variables and functions in the program's memory. We simply use the `&` operator:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    double half = 0.5;
    char vowel = 'a';
    short years[3] = { 1776, 1791, 1789 };
    struct { int x; int y; } point = { 5, -12 };
    printf("main: %p\n", &main);
    printf("half: %p\n", &half);
    printf("vowel: %p\n", &vowel);
    printf("years, 0, 1, 2: %p %p %p %p \n",
        &years, &years[0], &years[1], &years[2]);
    printf("point, x, y: %p %p %p\n", &point, &point.x, &point.y);
    return EXIT_SUCCESS;
}
```

One run of this program produced:

```
main: 0x100707e18
half: 0x16ee2ad40
vowel: 0x16ee2ad3f
years, 0, 1, 2: 0x16ee2ad38 0x16ee2ad38 0x16ee2ad3a 0x16ee2ad3c
point, x, y: 0x16ee2ad30 0x16ee2ad30 0x16ee2ad34
```

C lets us see the memory of a running program as one large array of bytes. All of the program's data and code lives in his memory space. For the run above, the machine code of `main` was stored separately from its local variables. The locals were stored together, laid out as in Figure 13.1.

Figure 13.1 Memory layout of a function's local variables

From this output, we can infer that on the machine running this program, integers take up four bytes, shorts two, and doubles eight. Inferring sizes is fine, but we can be absolutely sure by using the `sizeof` operator, which reports the number of bytes required for either (1) instances of a given *type* (which must be parenthesized), or (2) for a given *expression*, if the operand is not a type name in parentheses. Let's explore, using the same variables as above:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    double half = 0.5;
    char vowel = 'a';
    short years[3] = { 1776, 1791, 1789 };
    struct { int x; int y; } point = { 5, -12 };
    printf("sizeof types char=%lu, short=%lu, int=%lu double=%lu\n",
        sizeof(char), sizeof(short), sizeof(int), sizeof(double));
    printf("sizeof vars half=%lu, vowel=%lu, years=%lu, point=%lu\n",
        sizeof half, sizeof vowel, sizeof years, sizeof point);
    return EXIT_SUCCESS;
}
```

Here's the output on the same machine:

```
sizeof types char=1, short=2, int=4 double=8
sizeof vars half=8, vowel=1, years=6, point=8
```

This looks good. The variable `point` takes up 8 bytes because it is made up of two four-byte integers, and `years` takes up 6 bytes as it consists of three two-byte shorts.

Did you notice the gap (the unused memory) at address 16ee2ad3e in Figure 13.1? This is due to **alignment**—the placement of data in memory at addresses divisible by a certain value. Generally, two-byte values are allocated at addresses divisible by 2, 4-byte values at addresses divisible by 4, 8-byte values at addresses divisible by 8, and ditto for 16. There are different rules for different hardware architectures. Unless you are manipulating addresses directly, alignment does not affect the meaning of your program, but because of the way hardware works, proper alignment can result in huge performance gains. The compiler will take care of the details for you.

Arrays and structs behave very differently when assigned (with `=`). Quite differently in fact, as (whole) arrays are *not* assignable:

```c
#include <stdlib.h>
#include <assert.h>

typedef struct { int x; int y; } point;

int main() {
    short a[3] = { 1776, 1791, 1789 };
    short b[3] = { 500, 100, 1500 };

    point p = { 5, -8 };
    point q = { -3, 21 };

    // a = b;         // DOES NOT COMPILE!
                      // array type 'short[3]' is not assignable
    a[1] = b[1];      // but this is ok

    p = q;            // all 8 bytes copied
    p.x = 0;          // writing to p's memory, not q's
    p.y = 0;
    assert(q.x == -3);
    assert(q.y == 21); // q unchanged!

    return EXIT_SUCCESS;
}
```

Arrays are not assignable even if never explicitly marked immutable or read-only. They cannot even be made assignable even if we *wanted* to assign the whole array. Structs, as the script shows, *are* assignable, and each of the elements is copied. This does mean that assigning very large structs (using the simple `=` operator) can be slow.

Now, if there is an array *inside* of a struct, the entire struct will be copied, including the (entire) internal array:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct { int turn; int diceRolls[2]; } player;

int main() {
    player alice = { 1, { 6, 6 } };
    player bob = { 2, { 5, 3 } };

    alice = bob;                    // ok, copies all 12 bytes
    alice.diceRolls[1] = 1;         // also ok!
    assert(bob.diceRolls[1] == 3); // unchanged! Array was copied
    // bob.diceRolls = { 1, 2 };   // NOT ok
    return EXIT_SUCCESS;
}
```

The `const` modifier is worth a mention. When attached to an array or struct variable, not only can the variable not be mutated as a whole, the individual elements and fields cannot be modified through the variable either:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct { int x; int y; } point;

int main() {
    const double half = 0.5;
    // half = 0.75; // ERR: assignment of read-only variable ‘half’

    const point p = { 5, -8 };
    // p = (point) { 0, 0 }; // ERR: assignment of read-only var ‘p’
    // p.x = 0; // ERR: assignment of read-only member ‘x’ in ‘p’

    const int rolls[5] = { 2, 2, 6, 1, 3 };
    // rolls = { 1, 1, 1, 1, 1 }; // We already know this is illegal
    // rolls[0] = 1; // ERR: assignment of read-only location ‘rolls[0]’
    return EXIT_SUCCESS;
}
```

Our last basic feature of the C is the union:

```
#include <stdio.h>
#include <stdlib.h>

union {
    int level;
    char button;
} action;
```

```
int main() {
    printf("&level=%p, &button=%p\n", &action.level, &action.button);
    action.level = 333;
    printf("level=%d, button=%c\n", action.level, action.button);
    return EXIT_SUCCESS;
}
```

One run of this program produced:

```
&level=0x104c20000, &button=0x104c20000
level=333, button=M
```

A C union looks very much like the unions we've seen in previous chapters on TypeScript and Julia. But there's a difference. TypeScript and Julia retain, at run time, the component type of the union value. If a union object in these languages were assigned the value 333, it's an integer—you cannot examine its "character nature." In C, a union is just a way for variants to exist at the same memory location. The value 333 is encoded as two bytes: `01 4d` (shown in hexadecimal as is the convention for displaying bytes). Now `char`s in C take up one byte, and since the two variants are overlaid in memory, the `button` variant sees the byte `4d` which is the encoding of the character LATIN CAPITAL LETTER M.

While the ability to access the "wrong type" of a union feels like a security issue, the design is intentional. C, unlike Julia and TypeScript, does not keep track of the intended type at run time, as doing so would require runtime overhead in space (to store the type indicator) and in time (to perform type checking). C chooses performance over security.

Exploring more of C, including strings, variable-sized arrays, dictionaries, and function calls, requires a much deeper dive into the world of addressing, which is the topic we'll turn to now.

13.3 POINTERS

In the previous section, we encountered the address-of operator, `&`. The result of applying this operator to a variable produced a value that looks surprisingly like an integer. If it were an integer, we should be able to perform arithmetic on it. Can we?

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    double x = 395.22;
    int y = 5;
    int z = -393;
    printf("&x = %p, &y = %p, &z = %p\n", &x, &y, &z);
    printf("&z+1 = %p, &z+2 = %p, &z+3 = %p\n", &z+1, &z+2, &z+3);
    return EXIT_SUCCESS;
}
```

One run of this program produced:

```
&x = 0x16d76b440, &y = 0x16d76b43c, &z = 0x16d76b438
&z+1 = 0x16d76b43c, &z+2 = 0x16d76b440, &z+3 = 0x16d76b444
```

Why did adding one to `&z` appear to add four? The answer is that the address-of operator produces not a number, but a **pointer**, and pointer values are typed. The expression `&z` produces "the address of int z," or "a pointer to `int` z." Because x has type `int`, the type of `&z` is `int*`, or "pointer to int." When adding (integer) **offsets** to pointer values, the values are scaled by the size of the referent type. This is useful, as it means `&z+1` is the address of the integer variable following z, in this case y (see Figure 13.2).

Figure 13.2 Another memory layout

To access the contents of memory at a given address, we use the **dereference operator**, `*`. This works for both reading and writing memory. Let's see how we can change the value of y in the previous example using a pointer manufactured from the address of z:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int y = 5;
    int z = -393;
    *(&z + 1) = 233;
    printf("y = %d\n", y);
    return EXIT_SUCCESS;
}
```

On one machine this program did produce `y = 233`.

The fact that pointer arithmetic allows access to adjacent values in memory creates an interesting similarity between pointer arithmetic and array indexing. Note, in Figure 13.3 the *equivalence* between the expressions `&(a[2])` and `&(a[0])+2`.

Figure 13.3 Array indexing and pointer arithmetic

Here's where things get interesting. The C standard states that any expression of type "array of T" is *automatically converted* to an expression of type "pointer to T" that points to the first element of the array *except* when the array expression is (1) the operand of the address-of operator `&`, (2) the operand of `sizeof`, or (3) a string literal used to initialize an array. This automatic conversion of array expression `a` into the pointer `&(a[0])` is known as **array decay**. "Decay" is an apt term, since converting an array to a pointer means the array expression loses its size.

Because of array decay, we can write `a+2` in place of `&a[0]+2`, which we know is `&(a[2])`, so `a[2]` is actually `*(a+2)`. Pointer arithmetic does not require the pointer to be on the left of the `+`, so this expression can also be written `*(2+a)`, and therefore...`2[a]`. Let's verify this with a small script:

```
#include <stdlib.h>
#include <assert.h>

int main() {
    int a[5] = { 10, 20, 30, 40, 50 };
    assert(a[2] == 30 && 2[a] == 30);
    assert(*(a + 2) == 30 && *(2 + a) == 30);
    assert(a == &a[0]);
    assert(&a[2] == a + 2);
    return EXIT_SUCCESS;
}
```

When passing an array as an argument to a function, decay kicks in and only a pointer is passed, meaning the function should almost certainly take the array length as a second parameter. While generally inconvenient, there are two silver linings: (1) arrays are not copied when passed as arguments, improving performance for large arrays, and (2) it simplifies the passing of array slices.

```
#include <assert.h>
#include <stdlib.h>

int sum(int* a, unsigned int length) {
    int total = 0;
    for (unsigned int i = 0; i < length; i++) {
        total += a[i];
    }
    return total;
}

int main() {
    int a[] = {7, 3, -8, 4, 1, 0, 11, 2};

    assert(sum(a, 3) == 2);       // sum of first three elements
    assert(sum(a, 8) == 20);      // sum of all elements
    assert(sum(a + 3, 2) == 5);   // a[3]+a[4]
    return EXIT_SUCCESS;
}
```

Although pointer values are typed, C allows you to **cast** one pointer type into another. This allows you to reinterpret the bytes at a memory location as any desired type. For example the bit pattern `11000010101110101000000000000000` interpreted as a 32-bit integer is the value -1027964928, but as a 32-bit float is the value -93.25. We can perform this "reinterpretation" by (1) storing the integer value in memory, (2) obtaining a pointer to this storage, (3) casting this integer-pointer to a double-pointer, then (4) dereferencing the double-pointer. It's a dense line of code, but characteristic of bit-level programming:

```
#include <inttypes.h>
#include <assert.h>
#include <stdlib.h>

int main() {
    int32_t a = 0b11000010101110101000000000000000;
    float b = *(float*)&a;
    assert(a == -1027964928);
    assert(b == -93.25);
    return EXIT_SUCCESS;
}
```

Other languages can do bit-level reinterpretation of memory, but only in a restricted number of cases. In Java, one can obtain the bits of a floating point number and interpret them as integers:.`Float.floatToIntBits(-93.25f)` evaluates to -1027964928. JavaScript's `DataViews` and Python's `struct` module can do the same. C's ability, though, to interpret *any* chunk of memory bits as (almost) *any* type, is rare among high-level languages.

To wrap up our introduction to pointers, let's explore a **null pointer** and a **wild pointer**:

```
int* p = NULL;     // null pointer
int* q;            // wild pointer
```

A null pointer, according to the language standard, is guaranteed to "compare unequal to a pointer to any object or function"; in other words, it points to nothing. Dereferencing a null pointer (i.e., evaluating `*p` in the example above) leads to **undefined behavior**—affectionately known as "UB"—more than likely an abrupt program crash. A wild pointer is one that is uninitialized. At runtime, the memory associated with the wild pointer variable *q* above will contain *some* bits—bits that may or may not be interpretable a legal address. There's no way to know, so a wild pointer dereference produces undefined behavior.

13.4 STORAGE DURATIONS

Because C allows the manufacture of pointers to *any* variable at all, one might create a pointer that outlives its referent—a **dangling pointer**—so an understanding of lifetimes is a prerequisite for C programmers. C's concept of a variable's lifetime is called a **storage duration**. Local variables and parameters defined within the function come into existence when the function is called and are destroyed when the function returns. As this creation and destruction happens automatically, these variables have **automatic storage duration**. Global variables, and those defined inside a function with the modifier `static` live as long as the program, and have **static storage duration**.

Dereferencing a dangling pointer not only produces undefined behavior, but leads to several security vulnerabilities including data corruption and information leakage [14]. The following script has undefined behavior, but one run of this code printed the value 8. As we saw in the previous chapter (covering Go), the call frames of function invocations are usually allocated and deallocated in a stack-like fashion, so the region of memory managing call frames is known as "the stack." It is not uncommon for frames of subsequent calls to end up at the same memory location as that of a previous call. If this happens in the script below, the dangling pointer that was created to point to f's local x will "see" g's local y. If the runtime allocates frames in a different manner, the output may be different. This is the essence of undefined behavior: you just don't know.

```
#include <stdlib.h>
#include <stdio.h>

int* f() {
    static int z;  // z has static storage duration
    int x = 5;     // x has automatic storage duration
    return &x;     // Terrible code, will return a dangling pointer
}                  // Compiling this code should generate a warning

void g() { int y = 8; }

int main() {
    int* p = f();           // Terrible code
    g();                    // g's frame might overlay f's
    printf("%d\n", *p);     // Undefined behavior, but might print 8
    return EXIT_SUCCESS;
}
```

If a function must create data that outlives the function itself, it can request memory via the library function `malloc`, or its friends `calloc` and `realloc`. The allocator function obtains the requested number of bytes in an area of memory known as "the heap," but more technically, we say the data has **allocated storage duration**. Such objects live from the instant they are created until the instant they are *explicitly* destroyed with the library function `free`. If the requested amount of memory is available, `malloc` allocates the memory and returns a pointer to the allocated block. The pointer will have type `void*`, which is assignable to variables of any pointer type. If the system cannot satisfy the request, a null pointer is returned.

Though C and Go both allow "pointers to any variable," the two languages differ greatly in how referenced data is managed. C allows dangling pointers. Go prevents them through escape analysis. C requires programmers to explicitly obtain and release heap memory. Go programmers aren't really aware that a help exists; Go automatically allocates heap data and relies on a garbage collector to deallocate.

To illustrate the dynamic allocation and deallocation of memory, we're going to build a stack-of-integers data structure. This exploration will not only illustrate typical uses of `malloc` and `free`, but also show how one can encapsulate these functions behind an abstraction layer that the rest of the system need not see. We'll also use the opportunity to show the interesting `->` operator: `p->x` simply means `(*p).x`.

Let's begin by placing the so-called "specification" of our stack datatype in its own file, *stack.h*. This specification file, or **header file** will contain only enough information for clients to use the stack, without divulging internal representation and implementation details. Note how C allows for partial specification of structs and functions:

```
#include <stdbool.h>
#define MAX_CAPACITY 4096

// Make the representation of the stack unknown to code that uses it.
// The type stack is a pointer to a struct that is defined elsewhere.
typedef struct _Stack* stack;

typedef enum {
  no_error,
  illegal_stack_capacity,
  out_of_memory,
  stack_full,
  stack_empty
} response_code;

// Result object for operations returning a stack. If an error
// occurs, the stack field will be NULL.
typedef struct {
    response_code code;
    stack stack;
} stack_response;

// Result object for operations returning a stack item. If an error
// occurs, the item field will be NULL.
typedef struct {
    response_code code;
    int item;
} item_response;

stack_response create(int capacity); // Must destroy() returned stack
int size(const stack s);
bool is_empty(const stack s);
bool is_full(const stack s);
response_code push(stack s, int item);
item_response pop(stack s);
void destroy(stack* s);
```

Our specification defines an **opaque type** `stack` and seven associated functions. All we know about the `stack` type is that it is a pointer to a struct, but we don't know the details of that struct. As C does not have exceptions, we handle errors by requiring all failable operations to return a status code, together with an optional result. And while we are at least not requiring our stack users to manually invoke `malloc` and `free`, we can't (without a *lot* of work) hide the fact that some memory management *has* to happen, so we will require users to call our `destroy` function when finished with their created stack.

In the implementation file, *stack.c*, we choose a representation and flesh out the implementation.

```
#include "stack.h"
#include <stdlib.h>

struct _Stack {
    int* items;       // array contents, capacity set on create()
    int capacity;     // have to store this separately!
    int top;          // index of next slot to fill, also the size
};

stack_response create(int capacity) {
    if (capacity < 1 || capacity > MAX_CAPACITY) {
      return (stack_response){illegal_stack_capacity, NULL};
    }
    stack s = malloc(sizeof(struct _Stack));
    if (s == NULL) {
      return (stack_response){out_of_memory, NULL};
    }
    s->capacity = capacity;
    s->top = 0;
    s->items = malloc(capacity * sizeof(int));
    if (s->items == NULL) {
      return (stack_response){out_of_memory, NULL};
    }
    return (stack_response){no_error, s};
}

int size(const stack s) { return s->top; }
bool is_empty(const stack s) { return s->top == 0; }
bool is_full(const stack s) { return s->top == s->capacity; }

response_code push(stack s, int item) {
    if (is_full(s)) {
      return stack_full;
    }
    s->items[s->top++] = item;
    return no_error;
}

item_response pop(stack s) {
    if (is_empty(s)) {
        return (item_response){stack_empty, 0};
    }
    int popped_value = s->items[--s->top];
    return (item_response){no_error, popped_value};
}
```

```
void destroy(stack* s) {
    // Both the array and the stack itself were malloc'd
    free((*s)->items);
    free(*s);
    *s = NULL;  // It is good practice to null a ptr after free
}
```

Dangling pointers—pointers referencing memory belonging destroyed variables (either automatic variables whose enclosing function has returned or allocated variables that have been freed)—are not the only duration-related source of vulnerabilities. Allocating variables and forgetting to free them is a problem known as a **memory leak**. Invoking `free` on the same pointer multiple times is a **double free**. Each are problematic, but a proper understanding of storage durations[1] is helpful to avoid them.

13.5 STRINGS

Strings are complicated in C. They are essentially character arrays, and like all arrays, must at times decay into pointers and require programmer discipline to ensure enough memory is available for the desired operations. But not every element of the array is part of the string. The string contents are those characters up to, but not including, the first zero-valued element of the array, regardless of the array size. Arrays without a zero element are seriously problematic—more on this later. Strings can be arrays of type `char` (e.g. `"Hello"`), `char8_t` (e.g., `u8"Hello"`), `char16_t` (e.g., `u"Hello"`), or `char32_t` (e.g., `U"Hello"`). For simplicity, we'll restrict our discussion to arrays of `char`, and assume a UTF-8 encoding.

The simplest way to obtain a string is via a **string literal**. A literal expression creates an array (of static storage duration) containing the string characters plus one extra zero element at the end, called the **string terminator**. For example, the literal

"τ=2π"

allocates a 7-byte buffer, 6-bytes to encode the string (τ is `CF 84`, = is `3D`, etc.) and a zero-byte at the end:

			CF	84	3D	32	CF	80	00		

Interestingly, C allows you to assign a string literal to a nonconstant variable, but attempts to modify the string through the variable result in undefined behavior.

You can also allocate the memory yourself via an array variable declaration (e.g., `char buffer[100];`) or via dynamic memory allocation (e.g., `char* buffer = malloc(100);`). You can then populate the array elements by direct assignment, reading lines from a file, fetching content over a network, or using a library function for copying elements from one string to another, but must always be acutely aware to (1) ensure you never try to stuff more characters into the array than slots available, and (2) always place the zero-terminator somewhere in the array! Once obtained, strings values, can be manipulated by a number of functions from *string.h*:

[1] In the interest of completeness, there is a fourth duration, called **thread storage duration**, in which a variable lives as long as its owning thread. There's a little more to these variables; you can explore them in one of the end-of-chapter exercises.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char s[25];                         // large buffer, ok
    s[0] = 'H';
    s[1] = 'i';
    s[2] = '!';
    s[3] = '\x00';                      // zero terminator!
    assert(strlen(s) == 3);             // length of string
    assert(strcmp(s, "Hi!") == 0);      // compare (equal)
    assert(strcmp(s, "Hat") < 0);       // compare (less)
    assert(strcmp(s, "Yo") > 0);        // compare (greater)
    assert(strstr(s, "i!") == s + 1);   // find substring
    assert(strstr(s, "x") == NULL);
    assert(strchr(s, '!') == s + 2);    // find character
}
```

These string functions work by scanning the string character-by-character until the terminating zero element is found. If no zero element exists in the array, the operations will scan beyond the end of the array looking for a zero—possibly divulging confidential information on reads, and possibly corrupting data on writes. A large run of nonzeros could cause the scan to run for an unbounded amount of time, or run into memory not owned by the currently executing program, causing a crash. Overrunning buffers happens more frequently in practice than we'd like to admit, and the consequences are rather severe, that it's worth devoting a brief section to discuss the topic.

13.6 BUFFER OVERRUNS

Missing a zero terminator in a string is not the only way to overrun a buffer. An attempt to *fill* a buffer with characters can also overrun if the programmer is not careful to stop filling before array space is used up. There are three reviled functions that are so prone to buffer overruns that many generally recommend avoiding them at all costs.

- `strcpy(char*` s`, const char*` t`)`. Copies the characters starting at t, up to and including the zero terminator, into the buffer starting at s. There is no way for this function to check whether s is "big enough."[2]
- `strcat(char*` s`, const char*` t`)`. Copies the zero-terminated string starting at t to the end of the zero-terminated string starting at s. The character `t[0]` replaces the original zero terminator at the end of s. The zero-terminator of t is copied as well, which is nice, but again, there is no way to tell whether s is "big enough."
- The worst of the worst, a function named `gets`, which was removed from the language's standard library in 2011.

[2] There is also no way to tell whether or not s is even a decayed character array to begin with, nor whether t is itself a proper zero-terminated string, or even whether the two memory regions of s and t overlap. While these case are just as problematic, our focus on this section is that of buffer overruns due to insufficiently allocated space for the destination array pointed to by s.

If one must copy characters from one string to another, various replacements for `strcpy` exist. `strncpy(`s`,`t`,`n`)` copies *at most* n characters of t into s. Here the programmer is responsible for setting n correctly, but when set to a constant that can be checked by a human code reviewer or automated program analyzer, disaster may be avoided. Unfortunately, this function will *not* copy the zero-terminator if all n characters need to be copied. The nonstandard `strlcpy(`s`,`t`,`n`)` copies at most $n-1$ characters and *always* places a single zero-terminator in the destination. However, it still leaves open the possibility of silently truncating the source string t if it is longer than the destination buffer for s. Standard C has specified, but leaves optional to implementors to include, `strlcpy(`s`,`n`,`t`)`. This might be the best of the bunch, as it returns an error indicator for anything that can go wrong, e.g., destination buffer too small, null pointers for either argument, memory overlap, or truncation. Alas, even if `strcpy_s` is available on one's platform, programmers still have the responsibility to check the return value for errors.

C explicitly allows the possibility of buffer overruns because it intentionally favors speed and flexibility over security. There is a great deal of literature available on exploiting different types of buffer overruns [85]. Some exploits may result in attackers obtaining control of a victim machine. In fact, seventy percent of security vulnerabilities are due to programmer errors with memory management [16]. So, how might be best avoid these problems?

One way is to simply not use C. All of the previous languages on our tour, including Python, Java, Kotlin, TypeScript, and even Go, either produce errors on out-of-bounds array access or automatically grow a buffer when it needs to handle more data than its current capacity. These languages do not have double-free problems because there is no explicit allocation and deallocation, nor can one ever "forget" to free memory because a garbage collector will automatically free unused memory.[3] However, garbage collection comes at a performance cost which C refuses to spend.

If C has to be used, one can create specialized datatypes for strings (and other collections) which grow and shrink automatically behind the scenes. The `realloc` function features prominently in such scenarios. The C++ language—a nearly a perfect, but not quite superset of C—makes heavy use of this approach. In C++, one writes classes whose constructors allocate memory, mutators reallocate when needed, and destructors free memory. The implementations of these methods are all behind the scenes, freeing users of these classes from memory management concerns; however, implementations of these classes must still manage memory carefully. If alternatives to C is an option, but running a garbage collector is not, there is hope: modern languages such as Swift and Rust *do* achieve memory safety without a garbage collector. These two popular languages will be coming up soon on our tour.

13.7 THE PREPROCESSOR

The text of a C program may contain a number of **directives** which are processed *before* the program is compiled. The directives are processed by the aptly named **preprocessor**. The `#include` directive, seen throughout this chapter, may be the most widely used: it injects the source code of another file directly into the current file. The `#define` directive introduces a new symbol, possibly parameterized, together with its definition. The preprocessor replaces occurrences of defined patterns, known as **macros**, before compiling the program. Here's a brief example:

[3] Interestingly, memory leaks can occur in garbage-collected languages but they have a completely different flavor: memory "leaks" whenever a global variable holds on to data it will never use again.

```c
#include <stdio.h>
#include <stdlib.h>

#define triple(x) 3*x

int main() {
    printf("%d\n", triple(5 + 2));
    return EXIT_SUCCESS;
}
```

Here the preprocessor expands the macro `triple`, replacing occurrences of `x` with the text `5 + 2`, resulting in `3*5 + 2`. The program prints 17. Within *stdlib.h* the symbol `EXIT_SUCCESS` is also `#define`d. On a machine where this symbol is equated to 0, the preprocessed version of `main` looks like:

```c
int main() {
    printf("%d\n", 3*5 + 2);
    return 0;
}
```

C's use of textual substitution of parameters in macros is far simpler than what we've seen in Julia and Clojure, and will see in Rust, languages whose macros work on higher-level syntactic units. In Julia, for example

The fact that C's macro system works on a textual level is definitely in line with it's being simple and somewhat insecure. Julia, Rust, and Clojure, among other languages, have more interesting macro systems that are all hygienic, meaning variables are not captured. In Julia invoking the equivalent `macro triple(x) :(3 * $x) end` with `5 + 2` produces 21. You can get this effect in C by adding parentheses:

```c
#define triple(x) (3*(x))
```

Adding excessive parentheses is naturally a best practice.

A handful of other directives—`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#elifdef`, `#elifndef`, and `#endif`—support conditional compilation. We'll leave these to you to research. You can even cause errors and warnings to occur with `#error` and `#warning`. The `#line` directive changes the line number used in reporting. Here is a small program illustrating a few directives, and well as a few built-in macros!

```c
// This little exploration features a bunch of directives.
// Run with `gcc -E directives.c` to see prepreprocessor output.

#include <assert.h>
#define ZERO 0
#define MAX(a,b) ((a) > (b) ? (a) : (b))

#ifdef ZERO
    #define ONE 1
#else
    #define ONE 0
#endif
```

```
#warning "This is a warning, don't worry about it (j/k)."

int main() {
    int x = MAX(ONE, ZERO);
    #line 1000
    int y = MAX(ONE, ZERO);
    #line 700
    int z = x * MAX(x + y, ZERO);
    assert(701 == __LINE__);

    int line =  __LINE__;
    char* filename = __FILE__;
    int version = __STDC_VERSION__;
    char* compiledAt = __DATE__ __TIME__;

    #warning "Line numbers are crazy, right?"
    return ZERO * ONE;
}

#undef ONE
```

It may be fun to try to predict this program's output before executing it!

C WRAP UP

In this chapter, we were introduced to C. We learned that:

- C is a low-level language that was intentionally designed to favor extreme simplicity and speed over safety. There is no garbage collector. Programmers must explicitly allocate and deallocate dynamic memory.
- Execution begins at `main`, whose arguments are the command line arguments, and whose return value is the exit status of the program itself.
- Perhaps the most characteristic aspect of C is its `&` operator that produces a pointer to a variable. Pointers represent the address in memory of a variable, and hence resemble integers, but addition of these values is scaled by the size of type of element referenced.
- Structs are copied with the assignment operator, `=`, which arrays cannot be copied with this operator.
- Unless operated on with `&` or `sizeof`, or represented as a string literal, array expressions decay into pointers.
- There are four types of storage durations: automatic, static, allocated, and thread.
- A dangling pointer occurs when a pointer references a variable that has been destroyed, that is, one whose storage duration has expired.
- (Proper) strings are arrays of character objects with a zero element. Only the elements before the zero element are part of the string. Character types include `char`, `char8_t`, `char16_t`, and `char32_t`.

- Buffer overruns are possible in C, but because of the potential security vulnerabilities surrounding them, are made impossible in many other languages. While overruns can occur with any type of array, they are often associated with string processing, given the complexity of string manipulation with the zero-terminators.
- Various alternatives to `strcpy` are in use, but none are able to completely free programmers from the responsibility to track buffer sizes.
- C has a very visible preprocessor, one of whose features is the ability to create macros via text substitution.

For further study in C, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using C, see the C page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/c.

EXERCISES

Now it's your turn. Continue exploring C with the activities and research questions below, and feel free to branch out on your own.

13.1 Install a C compiler on your local machine and run the three programs from the first section in the chapter.

13.2 Despite the rather large number of known security vulnerabilities that can occur in C and no other modern high-level language, C remains important. What are some reasons to continue learning and using C? (Consider such dimensions as legacy code, performance, and interoperability with other languages in your answer.)

13.3 Explain the meaning of the expression `sizeof(t)` where (a) t is the name of a type, and (b) t is an expression.

13.4 While arrays cannot be copied via the `=` operator, the `memcpy` function from *string.h* can copy arrays. Research this function and give examples of its use.

13.5 One hack to allow array copying with `=` is to wrap the array as the sole element of a struct. Construct an example program illustrating the approach, and compare it to simply using `memcpy`. What do you think about the hack? What are some pros and cons of this approach?

13.6 Structs are not only copied with the assignment operator `=`, but are also copied when passed as function arguments. If a function is intended to read, but not mutate, a struct, how can allow the function to work without requiring the full struct to be copied? (Hint: pointers are your friend here.)

13.7 In the example associated with Figure 13.3, we used showed that `&(a[2])` was equivalent to `&(a[0])+2`. Why did we not say the former was equivalent to `(&a)+2`? After all, don't `a` and `a[0]` live at the same address?

13.8 Can a pointer be manufactured to reference anything other than a variable? For example, is an expression such as `&5` legal? If so, what does it mean? If not, what error is reported?

13.9 How does pointer "subtraction" work in C?

13.10 Modify the stack-of-integers example from this chapter to start with a *capacity* of size 16, and every time an element is pushed, the stack is first checked to see if its size is at capacity. If so, the stack should be reallocated to a buffer twice its current capacity (using `realloc`) before pushing. Make sure, though, that the stack is not allowed to grow past a absolute maximum size of 32768 elements. After popping, halve the capacity if the stack size has gone below 25% capacity, but only if the capacity is larger than 16.

13.11 The call frames described in the section on storage durations in this chapter contain not only local variables and possibly parameters, but also bookkeeping information such as the **return address** of the function, that is, the address in memory of the code that should be executed after the function returns. Buffer overruns of local variables can overwrite a return address, potentially executing code hand-crafted by an attacker. Read AlephOne's famous article describing this exploit, *Smashing the Stack for Fun and Profit* [2], and summarize its main points.

13.12 The data science program from the opening section was not written as carefully as it should have been. Buffer overflows were prevented by ensuring team names and player names never overran their upper limits of 7 and 63 bytes respectively and that zero terminators were always inserted. However, size information was repeated several times throughout the code. Rewrite this program so that the maximum sizes for player and team names are defined only once.

13.13 Because C's macros use textual substitution, they are naturally nonhygienic. Construct an example to demonstrate the lack of macro hygiene.

CHAPTER 14

Swift

Swift is an innovative, open-source language developed by Apple to make software safer, faster, and more fun to create.

First appeared	2014
Creator	Apple
Recognized for	Expressivity and safety
Notable uses	Native applications for macOS, iOS, and their cousins
Tags	Imperative, Safe
Six words or less	The language for iOS apps

Throughout the 2010s and beyond, two operating systems dominated the mobile market: Android [32] and iOS [6]. While some mobile developers target only a device's web browser, many will opt to write *native* applications in a language and environment built to talk directly with the operating system. Android devices are natively programmed in Java or Kotlin. iOS applications are built with Objective-C or Swift.

Swift is a powerful, general-purpose systems language that feels like a scripting language. While best known for its support for the Cocoa and SwiftUI frameworks that provide services to iOS and its cousins macOS, iPadOS, watchOS, tvOS, and visionOS, our focus in this chapter will be on Swift the language, not the APIs. Swift has much to offer the student of languages, especially as its initial developer, Chris Lattner, is famous for originating LLVM—a compiler framework used by many modern language implementations.

Swift's design exhibits an elegant, consistent integration of the best modern features around. We'll start with our usual three example scripts and present the basic elements of the language. We'll pay particular attention to the features of Swift that make it a *safe* language, such as guaranteed memory initialization, static typing, overflow checking, and excellent syntactic support for optionals. We'll also study **argument labels**, matching, the role of **protocols** and **extensions** in the type system, and **automatic reference counting**, a mechanism for memory management that requires no background garbage collection. We'll close with Swift's concurrency features, noting their design for safety.

14.1 HELLO SWIFT

Let's begin, as we do every chapter, with a script to list small right triangle integers:

```
for c in 1...40 {
    for b in 1..<c {
        for a in 1..<b where a * a + b * b == c * c {
            print("\(a), \(b), \(c)")
        }
    }
}
```

Swift uses `...` for inclusive ranges and `..<` for ranges exclusive at the upper end. The `where` clause on the for-loop avoids yet another level of indentation and reads rather nicely. String interpolation uses `\()`.

Our second traditional script, Heap's algorithm for permutations, illustrates a fair number of Swift features and idioms:

```
func generatePermutations(of a: inout [String], upTo n: Int) {
    if n == 0 {
        print(a.joined(separator: "\t"))
    } else {
        for i in 0..<n {
            generatePermutations(of: &a, upTo: n-1)
            a.swapAt(n % 2 == 0 ? 0 : i, n)
        }
        generatePermutations(of: &a, upTo: n-1)
    }
}

var args = Array(CommandLine.arguments[1...])
generatePermutations(of: &args, upTo: args.count-1)
```

Swift is heavily statically typed. As in Java, Kotlin, and Go, we must explicitly specify parameter types, but local variable types can usually be inferred. Parameters can be declared with **argument labels** to permit readable function calls, and modified with `inout` to enable changes to the parameter be reflected in the passed argument, which then must be marked with the `&` prefix. The array of command line arguments contains the program name at index 0, so we grab the values starting at index 1 into an array slice (`[1...]`). Swift distinguishes slices from true arrays, but a simple type conversion makes a new array from the slice.

Here is our third teaser, the data science script originally presented on page 17:

```
import Foundation

struct Player {
    let name: String
    let team: String
    let ppg: Double
}
```

```
var players = [Player]()
while let line = readLine() {
    let info = line.components(separatedBy: ",")
    guard let games = Int(info[2]) else { continue }
    guard let points = Int(info[3]) else { continue }
    let player = Player(
        name: info[1],
        team: info[0],
        ppg: Double(points) / Double(games)
    )
    if games >= 15 {
        players.append(player)
    }
}
for player in players.sorted(by: {$0.ppg > $1.ppg})[..<10] {
    let name = player.name.padding(
        toLength: 22, withPad: " ", startingAt: 0)
    let team = player.team.padding(
        toLength: 4, withPad: " ", startingAt: 0)
    print("\(name)\(team)\(String(format: "%8.2f", player.ppg))")
}
```

This script illustrates quite a few interesting Swift features. The pattern matching in the conditions of `while` and `guard` statements are more powerful than they look: both `readLine` and `Int` produce optional values, and the match only succeeds for nonnil values. We also see how parameters to closures can be written `$0`, `$1`, etc., and even a nice syntax for slicing arrays. These examples offered a small taste of Swift. Let's explore more.

14.2 THE BASICS

Swift is a modern language, sporting default parameters, parameter packing and unpacking, static typing with type inference, first-class functions and closures, extensive pattern matching and binding capabilities, option types in lieu of null references, result objects as an alternate error handling mechanism, asynchronous features, and generics. We'll tour the basics quickly, introducing new features and terms as they come up. We'll leave some of the flagship features such as protocols and extensions, and discussions of safety features, to subsequent sections.

The standard library includes the signed integer types `Int8`, `Int16`, `Int32`, `Int64`, and `Int`; unsigned integer types `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `UInt`; and floating-point types `Float` (32-bits) and `Double` (64-bits). The types `Bool`, `Character`, and `String` should be familiar, though—refreshingly—Swift's character and string types understand graphemes! For example, the Swift string literal `"e\u{301}"` is formed from the two code points `U+0078` (LATIN SMALL LETTER E) and `U+0301` (COMBINING ACUTE ACCENT), which in sequence represent a *single* character. Swift also knows that Unicode has provided a **precomposed** character for this combination, namely `U+00E9` (LATIN SMALL LETTER E WITH ACUTE) and will equate the precomposed character with the two-code point sequence:

```
// e with acute accent
let s1 = "e\u{301}"                // e + combining acute accent
let s2 = "\u{e9}"                  // the precomposed character
assert(s1.count == 1)              // Swift combines to count
assert(s1 == s2)                   // == compares the characters

// x in a circle
let s = "x\u{20dd}"                // x + combining circle
assert(s.count == 1)               // one actual character
assert(s.utf16.count == 2)         // 0078 20DD
assert(s.utf8.count == 4)          // 78 E2 83 9D
```

Interestingly, Swift has neither primitive types nor built-in types. The basic types are defined in the standard library along with well over one hundred other types. All types in Swift belong to one of seven kinds:

- *Structures*, with value semantics;
- *Classes*, with reference semantics;
- *Actors*, which (to a first approximation) are thread-safe classes;
- *Enumerations*, Swift's sum type, also with value semantics;
- *Protocols*, sets of requirements *adopted* by structures, classes, and enumerations;
- *Tuple types*, Swift's product type;
- *Function types*, containing the parameter and return types for functions.

Under **value semantics**, assignment, argument passing, and function return behave as if they create copies, while under **reference semantics**, these behave as if sharing takes place:

```
struct S {var x = 0}     // structs have value semantics
var s1 = S()             // create an *instance* of type S
var s2 = s1              // copy, by value
s1.x = 5                 // mutate through s1
assert(s2.x == 0)        // s2 unchanged

class C {var x = 0}      // classes have value semantics
var c1 = C()             // create an *instance* of type C
var c2 = c1              // reference copy
c1.x = 5                 // mutate through c1
assert(c2.x == 5)        // c2 sees change
```

The numeric types, `Bool`, `Character`, and `String` are all structures. So are:

- `[T]`, arrays of elements of type T
- `Set<T>`, sets of elements of type T
- `[K:V]`, dictionaries with key type K and value type V

Arrays, sets, and dictionaries have a `count` method, equality operators (`==`) and (`!=`) that compare *values* (not just references!), and iteration capabilities via the `for-in` statement. Let's explore some array operations:

```
var a = [10, 20, 30]
var b = [Int](repeating: 1, count: 4)
a[2] = 40
a.append(50)
a += b
assert(a.count == 8)
a[1...3] = [0, 2, 4]
a.remove(at: 3)
assert(a == [10, 0, 2, 1, 1, 1, 1])
```

and some set operations:

```
let a: Set<Int> = [10, 20, 30]
let b: Set<Int> = [20, 30, 40]
assert(a.union(b).count == 4)
assert(a.intersection(b) == Set([20, 30]))
assert(a.contains(10))
assert(!b.isEmpty)
```

and a dictionary, mapping the four constituent countries of the Kingdom of the Netherlands to their capitals:

```
let capitals = [
    "Netherlands": "Amsterdam",
    "Aruba": "Oranjestad",
    "Cura\u{e7}ao": "Willemstad",
    "Sint Maarten": "Philipsburg",
]

assert(capitals.count == 4)
assert(capitals.keys.contains("Aruba"))
assert(capitals.values.sorted().last == "Willemstad")

for (country, capital) in capitals {
    print("The capital of \(country) is \(capital).")
}
```

We use `let` for immutable variables and `var` for mutable variables. Like Rust but unlike many other languages, not only are `let`-defined structure variables nonreassignable, but the structure *elements* are immutable as well:

```
let a = [10, 20, 30]
// a[2] = 40    // is a compile time error!
// a.append(40) // is a compile time error!
```

Function definitions must indicate the types of all parameters and the return value if any. Parameters have both a name (to use within the function) and an **argument label** (used in the call). By default, the argument label is identical to the parameter name. When calling a function, you must specify the argument label, unless it is the special name `_`:

```
func perimeter(base: Int, height: Int) -> Int {
    return 2 * (base + height)
}

func average(of x: Int, and y: Int) -> Double {
    return Double(x + y) / 2.0
}

func middle(_ x: Int, _ y: Int) -> Double {
    return Double(x + y) / 2.0
}

assert(perimeter(base: 4, height: 20) == 48)
assert(average(of: 5, and: 8) == 6.5)
assert(middle(5, 8) == 6.5)
```

Swift considers functions to be special cases of **closures**—code blocks that can be assigned, passed as parameters, and invoked. Swift closures (other than global functions) capture bindings from the context containing the closure's definition, and are thus lexical closures. Two forms of **closure expressions**, featuring type inference, implicit returns, allow succinct code. One form uses named parameters and the other features **shorthand argument names** (`$0`, `$1`, etc.):

```
let a = [(3,1), (2,8), (3,20)]
assert(a.map({x, y in x*x + y*y}) == [10, 68, 409])
assert(a.map({$0*$0 + $1*$1}) == [10, 68, 409])
```

Classes and structs group properties (defined with `var` and `let`), methods (with `func`), subscripts (with `subscript`), and initializers (with `init`). Properties can be stored or computed:

```
import Foundation

struct Circle {
    var radius: Double                              // stored property
    var area: Double {                              // computed property
        get {return Double.pi * radius * radius}
        set(a) {radius = sqrt(a/Double.pi)}
    }
}

var c = Circle(radius: 10)
assert(c.radius == 10 && c.area == 100*Double.pi)  // invokes get
c.area = 16*Double.pi                               // invokes set
assert(c.radius == 4 && c.area == 16*Double.pi)
```

A computed property can be made read-only by providing only a getter. When only a getter is present, you can omit the `get` keyword, writing for example, `var area: Double { return M_PI * radius * radius }`. All stored properties must be initialized on instance creation, either through default values on the property definition, or assigned within an explicit

initializer. Structures are not required to have explicit initializers: an initializer accepting all stored properties as arguments is generated for you. Classes must have an explicit initializer, unless all stored properties have defaults (though in this case the automatically-generated initializer takes no arguments).

Classes, unlike structures, can **inherit** from each other, and their members may be overridden. Initializers come in two flavors, **designated** and convenience. Convenience initializers must invoke designated initializers; they generally supply convenient defaults:

```
class Rectangle {
    let width: Double
    let height: Double

    init(width: Double, height: Double) {
        self.width = width
        self.height = height
    }

    var description: String {
        return "Rectangle \(self.width) x \(self.height)"
    }

    func area () -> Double {
        return self.width * self.height
    }
}

class Square: Rectangle {
    init(side: Double) {
        super.init(width: side, height: side)
    }

    convenience init() {
        self.init(side: 1.0)
    }

    override var description: String {
        return "Square with side \(self.width)"
    }
}

let r = Rectangle(width: 20, height: 30)
assert(r.width == 20)
assert(r.area() == 600)
assert(r.description == "Rectangle 20.0 x 30.0")
let s = Square(side: 50)
assert(s.width == 50)
assert(s.area() == 2500)
assert(s.description == "Square with side 50.0")
```

We'll introduce Swift's disjoint sum type, the **enumeration**, by way of example. We'll define a token as (1) the value `plus`, (2) the value `minus`, (3) a double, or (4) a string. Enumeration values must be tagged—the string `"x"` does not belong to our token type; `Token.identifier("x")` does. Processing enumerations often occurs via destructuring in switch statements:

```
enum Token {
    case plus
    case minus
    case number(Double)
    case identifier(String)

    func lexeme() -> String {
        switch self {
            case .plus: return "+"
            case .minus: return "-"
            case .number(let value): return String(value)
            case .identifier(let name): return name
        }
    }
}

let expression: [Token] = [.number(3.0), .plus, .identifier("x")]
let text = expression.map{$0.lexeme()}.joined(separator: " ")
assert(text == "3.0 + x")
```

Values of an enumeration type may be assigned **raw values**, from the enumeration's underlying **raw type**. This allows additional information to be associated with each enum value (not wholly unlike, but not the same as, structure field tags in Go). Raw types may be strings, numbers, or characters.

```
enum Suit: String {
    case spades, hearts, diamonds, clubs
}

enum Direction: Int {
    case north = 0, east = 90, south = 180, west = 270
}

assert(Suit.hearts.rawValue == "hearts")
assert(Direction.south.rawValue == 180)
assert(Direction.west == Direction(rawValue: 270))
```

Recursive enumerations, such as our starter binary tree below, must be marked `indirect`. A tree is either (1) empty, or (2) a node with a left subtree and a right subtree:

```
indirect enum Tree<T> {
    case empty
    case node(Tree, T, Tree)
```

```
    var size: Int {
        switch self {
            case .empty: return 0
            case .node(let left, _, let right):
                return left.size + 1 + right.size
        }
    }
}

let t: Tree<Int> = .node(.node(.empty, 3, .empty), 7, .empty)
assert(t.size == 2)
```

The T in `Tree<T>` is a parameter of the **generic type** Tree. We may parameterize structures, classes, and enumerations, and even functions. Here's a trivial generic function:

```
func twice<T>(_ x: T) -> [T] {
    return [x, x]
}

assert(twice(5) == [5, 5])
assert(twice("yak") == ["yak", "yak"])
```

In the following example, we apply the `>` operator to values of a parameterized type. As this operator is defined within the Comparable protocol[1], we must constrain the generic type parameter to those types that adopt Comparable:

```
func median<T:Comparable>(_ x: T, _ y: T, _ z: T) -> T {
    let (a, b) = (min(x, y), max(x, y))
    return b < z ? b : max(a, z)
}

for (x,y,z) in [(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)] {
    assert(median(x, y, z) == 2)
}
```

Swift's arrays and dictionaries are regular generic structure types defined in the standard library. The type expression `[T]` simply sugars `Array<T>` and `[K:V]` sugars `Dictionary<K:Hashable, V>`. Swift even sugars its optional type: `T?` stands for `Optional<T>`. Optionals can be unwrapped with a `let`-binding in an `if`, `while`, or `guard` statement:

```
for s in ["42", "dog", "3xy", "-2E3", "-1"] {
    if let i = Int(s) {
        print("\"\(s)\" parsed as an integer is \(i)")
    } else {
        print("\"\(s)\" can't be parsed to an integer")
    }
}
```

[1] Protocols are discussed in an upcoming section.

The `guard-else` form takes an "early-exit" approach: the else-branch must transfer control out of the enclosing scope, via `break`, `continue`, `return`, `throw`, or calling a `noreturn` function, avoiding excessive indentation and keeping code focused on the "happy-path." [78]

```
for s in ["42", "dog", "3xy", "-2E3", "-1"] {
    guard let i = Int(s) else {
        print("\"\(s)\" can't be parsed to an integer")
        continue
    }
    print("\"\(s)\" parsed as an integer is \(i)")
}
```

In cases where you need the unwrapped value of an optional but have a default value in mind, use the (short-circuit) nil-coalescing operator:

```
let x: Int? = 5
let y: Int? = nil

assert(x ?? 3 == 5)
assert(y ?? 3 == 3)
assert(x ?? [1,2,3][100] == 5)  // short circuit
```

We can also avoid unwrapping optional arrays during indexing, and optional functions in function calls:

```
var a: [Int]? = [10, 20, 30]
var b: [Int]? = nil
var f: ((Int)->Int)? = {$0 * 3}

assert(a?[2] == 30)
assert(b?[2] == nil)
assert(f?(100) == 300)
```

and optional instances in a similar fashion:

```
class Person {
    var name: String
    var supervisor: Person? = nil

    init(name: String, supervisor: Person? = nil) {
        self.name = name
        self.supervisor = supervisor
    }
}

var a = Person(name: "Alice")
var b = Person(name: "Bob", supervisor: a)
assert(a.supervisor?.name == nil)
assert(b.supervisor?.name == "Alice")
```

The `?.`, or `?[]`, and `?()` operators figure prominently in **option chaining** expressions, such as (the admittedly contrived) `previousWinner?().address?.city?.parks?[2]`.

14.3 OPERATORS

Just as Swift does not bake in its basic types but instead defines them in its standard library, so too does it define operators as regular functions within the library. The standard library defines a rich set of operators, including the prefix operators `!` (logical not), `~` (bitwise not), `+` (unary plus), and `-` (unary negation). The binary operators, from highest to lowest precedence, appear in Table 14.1:

Operator(s)	PrecedenceGroup	Asc	Description
`<<` `>>`	BitwiseShift	None	left shift, right shift
`*` `/` `%` `&*` `&`	Multiplication	L	multiply, divide, remainder, multiply w/overflow, bit-and
`+` `-` `&+` `&-` `|` `^`	Addition	L	add, subtract, add w/overflow, subtract w/overflow, bit-or, bit-xor
`..<` `...`	Range	None	half-open range, closed range
`is` `as` `as?` `as!`	Casting	L	type check, cast, cast as optional, unwrap cast
`??`	NilCoalescing	R	nil coalesce

`<` `<=` `>` `>=`
`=` `!=`
`===` `!==`
`~=`
	Comparison	None	less, less or equal, greater, greater or equal, equal, not equal, identical, not identical, pattern match

`&&`	LogicalConjunction	L	(short-circuit) logical and
`||`	LogicalDisjunction	L	(short-circuit) logical or
`?:`	Ternary	R	conditional

`=` `*=` `/=` `%=`
`+=` `-=` `<<=` `>>=`
`&=` `|=` `^=`
`&&=` `||=`
	Assignment	R	assignment

Table 14.1 Swift operators

Swift operators are just functions, so you can define your own implementations of them on your own data types. We've done this below in our recurring vectors example, to which we've taken the liberty to add a prefix unary negation operator:

```
import Foundation

struct Vector: CustomStringConvertible {
    let i: Double
    let j: Double

    func magnititude() -> Double {
        return sqrt(i * i + j * j)
    }
```

```
    var description: String {
        return "<\(i),\(j)>"
    }
}

func + (left: Vector, right: Vector) -> Vector {
    return Vector(i: left.i + right.i, j: left.j + right.j)
}

func * (left: Vector, right: Vector) -> Double {
    return left.i * right.i + left.j * right.j
}

prefix func - (v: Vector) -> Vector {
    return Vector(i: -v.i, j: -v.j)
}

let u = Vector(i: 3, j: 4)
let v = Vector(i: -5, j: 10)
assert(u.i == 3)
assert(u.j == 4)
assert(u.magnititude() == 5)
assert(String(describing: u + v) == "<-2.0,14.0>")
assert(u * v == 25)
assert(String(describing: -v) == "<5.0,-10.0>")
```

You can even define your own operators. There are quite a few rules surrounding which characters can appear in these **custom operators** (see [7] for full details). Among these rules: custom operator names must be made up of "symbol"-like characters, operators containing dots (`.`) must begin with a dot, a lone question mark is not an operator, and no postfix operator may begin with a question mark or exclamation mark.

Let's quickly run through the steps in creating a couple custom operators. We must *declare* the operators before defining them, assigning binary operators to a precedence group (either an existing group, defined in the Swift Standard Library, or one we define ourselves) and assigning an associativity (`left`, `right`, or `none`). Here we've created an infix `~|*|~` that binds more tightly than addition but less tightly than multiplication, by first creating a new precedence group:

```
precedencegroup JustForFun {
  higherThan: AdditionPrecedence
  lowerThan: MultiplicationPrecedence
  associativity: left
}

infix operator ~|*|~ : JustForFun
postfix operator ^^
```

```
func ~|*|~ (x: Int, y: Int) -> Int {
    return 2 * x + y
}

postfix func ^^ (x: Int) -> Int {
    return x - 2
}

assert(8^^ ~|*|~ 3 == 15)
```

14.4 PROTOCOLS

Now let's return to Swift's type system. We've seen five of the six kinds of types so far: structures (including numbers booleans, strings, arrays, and dictionaries), enumerations (including optionals), classes, tuples, and functions. The sixth, **protocols**, is Swift's analog of Java's interfaces and Ruby's mixins. A protocol specifies certain requirements that structures, enumerations and classes that wish to **adopt** the protocol must **conform** to.

For our introductory example, we define a protocol `Summarizable` for things with summaries, give a struct and enum that adopt it, and invoke the summary property through variables declared with the protocol type:

```
import Foundation

protocol Summarizable {
    var summary: String { get }
}

struct Circle: Summarizable {
    var radius = 1.0
    var summary: String {return "Circle with radius \(radius)"}
}

enum Direction: Int, Summarizable {
    case north, east, south, west
    var summary: String {return "Bearing \(90 * self.rawValue)"}
}

let a: [Summarizable] = [Circle(radius: 5), Direction.west]
assert(a[0].summary == "Circle with radius 5.0")
assert(a[1].summary == "Bearing 270")
```

The adopted protocols appear in the type declaration, comma separated, following any superclass specification (for classes) or raw type specification (for enums). The protocol definition itself defines a number of requirements for not only properties but initializers and functions (methods) as well.

The Swift standard library defines over 100 structures, enums, and classes, and several dozen protocols. Table 14.2 summarizes a handful of these structures and the protocols they adopt.

Struct	Conforms to Protocols
Array	ArrayLiteralConvertible, CollectionType, CustomDebugStringConvertible, CustomStringConvertible, MutableCollectionType
Bool	BooleanLiteralConvertible, BooleanType, CustomStringConvertible, Equatable, Hashable
Dictionary	CollectionType, CustomDebugStringConvertible, CustomStringConvertible, DictionaryLiteralConvertible
Double	AbsoluteValuable Comparable, CustomDebugStringConvertible, CustomStringConvertible, Equatable, FloatLiteralConvertible, FloatingPointType, Hashable, IntegerLiteralConvertible, Strideable
Int64	BitwiseOperationsType, CVarArgType, Comparable, CustomStringConvertible, Equatable, Hashable, RandomAccessIndexType, SignedIntegerType, SignedNumberType
Range	CollectionType, CustomDebugStringConvertible, CustomStringConvertible, Equatable
Set	ArrayLiteralConvertible, CollectionType, CustomDebugStringConvertible, CustomStringConvertible, Hashable
String	Comparable, CustomDebugStringConvertible, Equatable, ExtendedGraphemeClusterLiteralConvertible, Hashable, MirrorPathType, OutputStreamType, Streamable, StringInterpolationConvertible, StringLiteralConvertible, UnicodeScalarLiteralConvertible

Table 14.2 A selection of adoptions from the Swift Standard Library

Many of these standard protocols have names that indicate their intent very well. Adopt `Equatable` if it makes sense for your type to support value equality, adopt `Hashable` if you would like instances of your type to be used as dictionary keys, and adopt `Comparable` if your type has an underlying ordering. If you'd like values of your type to render in a certain way when printed, adopt (as we did in the previous section!) `CustomStringConvertible`. It's instructive to see how the Swift authors defined some of these protocols.

```
public protocol Equatable {
    @warn_unused_result
    func == (lhs: Self, rhs: Self) -> Bool
}
```

The type `Self` (note the capital "`S`") refers to the type adopting the protocol.[2]

If you adopt `Equatable` for your new type, you need only define `==`. You'll get its negation for free, as the library defines this function:

```
@warn_unused_result
public func != <T : Equatable>(lhs: T, rhs: T) -> Bool {
    return !(lhs == rhs)
}
```

`Comparable` requires you only to implement `<`, and `==`. The protocol already adopts `Equatable`, and the operators (functions) `<=`, `>`, and `>=` are defined in terms of `<`.

`Hashable` is another single-requirement protocol:

[2] Anyone who's had to define `equals` methods in Java might find this feature very welcome.

```
public protocol Hashable : Equatable {
    var hashValue: Int { get }
}
```

Building the language's basic types and collections—and indeed its entire standard library—around protocols rather than classes, has earned Swift the title of the world's first *protocol-oriented language* [1]. Protocols support value types as well as reference types (Swift's classes). They are not limited to single inheritance. And, as we'll see in the next section, they interact nicely with Swift's approach to retroactive modeling: extensions.

14.5 EXTENSIONS

As in Kotlin and Rust, there's no need in Swift to create named subtypes to add properties and methods to an existing type. **Extensions** add new properties, methods, initializers, subscripts, and nested types to existing types, and can even allow an existing type conform to a new protocol. When extending a type, the new functionality applies to all instances of the type, even those created before the extension was executed. To illustrate, let's add some properties and methods to the `Int` type. We'll add an **instance property** `abs`, a **type property** `allOnes`, and an **instance method** `factorial`:

```
extension Int {
    var abs: Int {return self >= 0 ? self : -self}

    func factorial() -> Int {
        return (1...self).reduce(1, {$0 * $1})
    }

    static var allOnes: Int {return ~0}
}

assert(8.abs == 8)
assert(Int.allOnes == -1)
assert(10.factorial() == 3628800)
```

Extensions solve the expression problem we first encountered with Clojure. We'll start with shapes as things which have an area. In Swift, we "start with a protocol, not a class" [1]. We'll add a type implementing this protocol, too:

```
protocol Shape {
    func area() -> Double
}

struct Square: Shape {
    var side: Double
    func area() -> Double {return side * side}
}

let s = Square(side: 10)
assert(s.area() == 100)
```

Nothing surprising here. But now, let's add functionality. We'll add a perimeter method to squares, but through a new protocol, since other things—not only shapes!—may want boundaries. An extension allows us to adopt a new protocol and add a new method *without modifying existing source code.*

```
protocol Boundaried {
    func perimeter() -> Double
}

extension Square: Boundaried {
    func perimeter() -> Double {return side * 4}
}

assert(s.perimeter() == 40)
```

We can create new types that implement multiple protocols:

```
import Foundation

struct Circle: Shape, Boundaried {
    var radius: Double
    func area() -> Double {return Double.pi * radius * radius}
    func perimeter() -> Double {return 2 * Double.pi * radius}
}

let c = Circle(radius: 8)
assert(c.area() == 64 * Double.pi)
assert(c.perimeter() == 16 * Double.pi)
```

We can extend protocols with implemented methods, much like Java's default methods. All types adopting the extended protocol pick up the new methods with the given implementation:

```
extension Shape {
    var json: String { return "{\"area\": \(area())}" }
}

assert(s.json == "{\"kind\": \"square\", \"side\": 10.0}")
```

Conforming classes may wish to provide their own implementation of a method from a protocol extension:

```
extension Square {
    var json: String {return "{\"kind\": \"square\", \"side\": \(side)}"}
}

assert(s.json == "{\"kind\": \"square\", \"side\": 10.0}")
```

14.6 SAFETY FEATURES

In the previous section, we saw ways in which Swift enables us to write *extensible* systems. We now turn to those elements of Swift's design that help us write *correct* software. A so-called **safe language** aids the programmer in writing correct code by providing as many guarantees and as few ambiguities as possible. Swift, for example, guarantees all variables have a value before use. Not only does the one-line script

```
print(x)
```

fail to compile (`error: use of unresolved identifier 'x'`) but the two-line script

```
var x: Int
print(x)
```

also fails to compile (`variable 'x' used before being initialized`). Swift is a **definitive initialization** language: the compiler performs data flow analysis to make sure all paths leading to a use of a variable go through an initialization. Some languages, including C, give no initialization guarantees, leaving the value of an "uninitialized" variable to be "whatever happens to be in memory" at the time; the behavior, therefore, of such a C program is undefined. Go takes a different approach, **default implicit initialization**, giving uninitialized variables the zero value of the declared type. Lua takes the same approach as Go, initializing with `nil`. Still other languages force the programmer to include initializing expressions with every variable declaration. Swift's solution realizes that requiring an initial value may be unwieldy, and defaulting to a zero value (not to mention `nil`) might not be the most sensible choice, as it does provide any checks against a programmer's possible unintentional omission of an initializer.

The convenience of specifying immutability of objects (in addition to preventing reassignment) through `let` declarations also limits the chance of some programmer error, as does the use of optionals for possibly missing data—no Billion Dollar Mistake. Another safety feature shared with modern, statically-typed languages (but not with C), is the requirement that patterns in matching constructs must be exhaustive. If cases are left uncovered (as may happen when adding features to code), the compiler will reject the program and force to you address the missing cases.

Swift also eliminates the surprises that come from **modular arithmetic** of fixed-size integers. In Go, eight-bit integers, for example, have values between -128 and 127. Adding one to 127 "wraps around" to -128, and adding 100 to 100 wraps to -56. In Swift, however, arithmetic overflow crashes the program, hard. If you desire wraparound semantics, use the operators `&+`, `&-`, and `&*`:

```
let x: Int8 = 100
let y: Int8 = 100
assert(x &+ y == -56)  // x + y would crash
```

While unrecoverable errors fail hard with an application crash, Swift provides clean ways to deal with errors that you can recover from. Begin by creating a type (structs or enumerations are common choices) that conforms to the standard protocol `Error`. You then have two options for functions that may fail: (1) return an instance of the enum type `Result<Success, Failure:Error>` (with `.success` holding the expected value and `.failure` holding the error), or (2) *throw* an instance of the error type:

```
enum BadLength: Error {
    case tooLong(by: Int)
    case tooShort
}

func checkBetween5And10(_ s: String) throws -> String {
    guard s.count >= 5 else {
        throw BadLength.tooShort
    }
    guard s.count <= 10 else {
        throw BadLength.tooLong(by: s.count - 10)
    }
    return s
}

func checkBetween3And5(_ s: String) -> Result<String, BadLength> {
    guard s.count >= 3  else {
        return .failure(.tooShort)
    }
    guard s.count <= 5 else {
        return .failure(.tooLong(by: s.count - 5))
    }
    return .success(s)
}

do {
    let s = try checkBetween5And10("0123456789ABCDEF")
    print("\(s) is okay")
} catch BadLength.tooShort {
    print("Too short")
} catch BadLength.tooLong(let howMuch) {
    print("Too long by \(howMuch)")
}

switch checkBetween3And5("11111") {
case .success(let s):
    print("\(s) is okay")
case .failure(BadLength.tooShort):
    print("Too short")
case .failure(BadLength.tooLong(let howMuch)):
    print("Too long by \(howMuch)")
}
```

Functions that may throw errors—known as **throwing functions**—must declare this fact by placing the keyword `throws` after the parameter list. Calls to throwing functions must be marked with `try`. Errors are caught in a `do-catch` statement; uncaught errors are propagated to the surrounding scope. You can, if desired, convert an error to an optional by invoking `try? f()`, producing the (wrapped) result of the call, if any, or `nil` if the function f throws.

14.7 AUTOMATIC REFERENCE COUNTING

Safety in programming practice is concerned not only with the lack of ambiguity, but also with the lack of surprises. Therefore, Swift does not employ a tracing garbage collector, as such collectors are known for causing unpredictable pauses or slowdowns. Yet neither does Swift leave memory management entirely to the programmer as does C. It uses a particular technique by which dynamically allocated (heap) data is predictably freed when it becomes no longer accessible.

In Swift, each heap-allocated object (class instance) is given a reference count, starting at 1 when first created. Each time a reference to the object is assigned to a new variable, or passed to a parameter, the reference count is incremented. The count is decremented when one of the referencing variables goes out of scope. When the count reaches zero, the deinitializer method is invoked and the object is reclaimed. This mechanism, **automatic reference counting**, or **ARC**, requires no garbage collection thread. Since allocation and deallocation occurs at identifiable points in execution, there is no need for a preemptive garbage collector. The following brief example shows an object being allocated and deallocated at predictable points in a script's execution:

```
var history: [String] = []

class C {
    init() {history.append("init")}
    deinit {history.append("deinit")}
}

func f(_ c: C) {              // 3. count==2
    history.append("f")
    print(c)
    history.append("/f")
}                             // 4. c out of scope, count == 1

func main() {
    history.append("main")
    let x = C()               // 1. object created, count==1
    f(x)                      // 2. reference passed
    history.append("/main")
}                             // 5. x out of scope, count == 0

main()
assert(history == ["main", "init", "f", "/f", "/main", "deinit"])
```

This example begins with a call to `main`, which allocates a local variable x in its stack frame. x is assigned a class instance, which in Swift is heap-allocated. The instance at this point has a reference count of 1 (as indicated with the big red circle):

Next, `main` calls `f`, passing x to the parameter c. Swift passes references to class instances, so our object now has a reference count of 2:

After `f` returns, its local variable c goes out of scope, removing one of the references to our object, reducing the reference count to 1:

When `main` returns, the object count goes to zero and the object is deinitialized.

In return for avoiding the significant complexity and potential unpredictability of a tracing, thread-based garbage collector, Swift requires developers to understand the workings of ARC. The following example illustrates a memory-management pitfall, known as a **memory leak**, that programmers must know how to avoid:

```
class Person {
    var name: String
    var boss: Person?
    var assistant: Person?   // CAUSES A MEMORY LEAK

    init(name: String, boss: Person? = nil) {
        initCount += 1
        self.name = name
        self.boss = boss
    }
    deinit {
        deinitCount += 1
    }
}

func main() {
    let alice = Person(name: "Alice")
    let bob = Person(name: "Bob", boss: alice)
    alice.assistant = bob
}

var (initCount, deinitCount) = (0, 0)
main()
assert(initCount == 2 && deinitCount == 0)
```

This script allocates two objects on the heap that each reference each other, forming a **reference cycle**. Figure 14.1 shows a snapshot of program execution just before the function `main` exits. Each object has a reference count of 2. After main returns, the local variables will go out of scope, reducing the reference count of each object to 1. However, the two objects are left on the heap, holding on to memory that ARC can never reclaim.

Figure 14.1 A reference cycle causing a memory leak under ARC

To prevent the memory leak, we will make one of the references (we'll choose the `assistant` property) to be a **weak reference**:

```
weak var assistant: Person?
```

The references we've seen so far have all been **strong references**. In Swift, a heap-allocated object can only be reclaimed when its strong reference count goes to zero. Figure 14.2 illustrates our fix to the memory leak in the previous example. Before `main` exits, the object representing Alice has strong reference count of 2, while only one strong reference refers to her assistant, Bob.

When `main` exits, the local variables go out of scope, decrementing the strong reference counts for both Alice and Bob. Alice's count goes to 1, and Bob's goes to 0. Therefore the Bob object can be reclaimed, decrementing the reference count for Alice to 0, at which time Alice is reclaimed as well.

Use weak references together with option-typed references to class instances. When cycles can occur among nonoptional instances, you will use **unowned references**. We won't be discussing these here, but we do mention them in the end-of-chapter explorations.

As an aside, Swift is able to interoperate with code written in Objective-C or C, languages teeming with unsafe features. To communicate with these external systems, Swift provides `UnsafePointer<T>` and `UnsafeMutablePointer<T>` among others, which serve as a stand-in for pointer types in C. Instances of these types are pointers to objects of type T and provide *no* automatic memory management whatsoever. They even lack the safety features

Figure 14.2 Weak references preventing a memory leak

of guaranteed initialization. *You* must both allocate and deallocate them yourself. For the curious reader, this little script illustrates the lifecycle of a typical unsafe pointer:

```
var p: UnsafeMutablePointer<Int>
// p is not allocated yet, using it here would be an error
p = UnsafeMutablePointer<Int>(allocatingCapacity: 1)
// allocated but not initialized, p.memory is an arbitarary integer
p.pointee = 8
// now p is initialized
assert(p.pointee == 8)
p.deinitialize(count: 1)
// at this point, the memory p points to is "garbage"
p.deallocateCapacity(1)
// now the memory is all freed up
```

14.8 STRUCTURED CONCURRENCY

As a language designed for programming interactive systems, Swift provides direct support for concurrent and even parallel execution. Swift's concurrency model is built on top of threads, but you never need to work with threads directly. Instead, you focus on:

- **Tasks**, units of work that can run concurrently with other tasks;
- Asynchronous functions (marked `async`), which can suspend execution at explicitly marked points (with `await`), yielding control of their thread until they are resumed; and
- **Actors**, that encapsulate and control access to shared mutable state.

While you *can* fire off tasks anytime and anywhere in your code, programmers are encouraged to use Swift's *structured* approach to concurrency, in which tasks are organized into a hierarchy. Parent tasks create and wait for child tasks—a parent cannot complete while a child is still running. As space is limited, we will give only one trivial but illustrative example of structured concurrency, leaving the large world of general Swift concurrency to your further study, of which [5] is a great place to start.

```
let numbers = Array(1...300)

func sum(ofSlice i: Int) -> Int {
    return numbers[i * 100 ..< (i+1) * 100].reduce(0, +)
}

@main
struct ConcurrentSumApp {
    func main() async {
        async let first = sum(ofSlice: 0)
        async let second = sum(ofSlice: 1)
        async let third = sum(ofSlice: 2)
        let partials = await [first, second, third]
        print("Sum is: \(partials.reduce(0, +))")
    }
}
```

Child tasks are launched with async-let or task groups. We've chosen the former to create three tasks to each sum a third of an array of numbers. These tasks are launched concurrently. The parent task awaits the completion of each child task, then adds the partial sums together and reports the result.

You may notice that this script has a special entry point. Here's why. Invoking `await` has the effect of suspending execution, and can only be done in an asynchronous context. Swift limits the places in which async functions can be called to: (1) other async functions, (2) code in the static main() method of a structure, class, or enumeration that's marked with `@main`, or (3) code in an *unstructured* task [5]. Option (3) would have required less code for our script, but we've chosen to build the entire example without relying on any unstructured forms.

Structured concurrency is a concept that has been around for decades. Where task relationships and lifetimes are tied to program structure, rather than spread throughout the code in an unstructured way, opportunities for programmer error are lessened. In addition, the compiler can use its knowledge of program structure to detect potential errors early, and simplify sometimes tricky priority and cancellation details among groups. Structured concurrency mechanisms (at least as implemented in Swift) also help provide a great deal of assistance with catching and propagating errors.

SWIFT WRAP UP

In this chapter, we were introduced to Swift. We learned that:

- Apple first released Swift in 2014 with iOS applications in mind, though the language is modern, comprehensive, and general-purpose.

- Swift is known for its safety features. It checks types at compile time, ensures that all variables are initialized before use, manages memory with ARC, and provides `let` declarations to not only prohibit reassignment of a variable but to make the variable's object immutable. Integer arithmetic does *not* "wrap around" by default. There are no null references.
- Though statically typed, Swift uses type inference extensively, but not quite as extensively as possible—function parameters, for one, require explicit annotations.
- Swift natively, without the use of an external library, can work with graphemes, an ability that very few languages have.
- Argument labels support highly readable calls.
- Functions are considered a type of closure. There are two syntactic forms for anonymous closures, `{$0 * $0}` and `{x in x * x}`
- Rather than being wired into the language, the basic types—numbers, strings, characters, booleans—are defined as structures in the standard library, as are arrays, dictionaries, and optionals. The same is true for operators, which are defined as library functions.
- All types in Swift belong to one of six *kinds*: structures, enumerations, classes, protocols, tuples, and functions. Structures and enumerations have value semantics (copied if necessary when assigned or passed as arguments); while classes have reference semantics.
- Arrays, sets, dictionaries, and optionals are defined as generic types in the standard library, though a good deal of syntactic sugar makes them pleasant to use.
- Classes can have superclasses and inherit properties and methods. Enumerations can specify a raw type. Classes, structs, and enumerations can adopt protocols.
- Extensions allow us to add new properties, methods, initializers, subscripts, and nested types without modifying the source code of the existing type. It is possible to extend a protocol, but only with implemented methods.
- Errors can be controlled by returning `Result` objects or by throwing.
- Swift's approach to garbage collection is to use automatic reference counting. The programmer does not have to insert code to free memory, but must code in such a way that heap-allocated memory will eventually become unreachable and thus freed. Weak and unowned pointers must be employed when objects appear in reference cycles.
- Swift supports structured concurrency through tasks and asynchronous functions, which can suspended execution with `await`. Structured concurrency via async-let and task groups carefully manage cancellation. Actors manage concurrent mutable state safely.

For further study in Swift, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Swift, see the Swift page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/swift.

EXERCISES

Now it's your turn. Continue exploring Swift with the activities and research questions below, and feel free to branch out on your own.

14.1 Locate the Swift home page, and browse its tutorials and references.

14.2 Practice with the Swift REPL.

14.3 Run the first three scripts from this chapter as command-line applications.

14.4 Install the Swift Playground and experiment with pre-built applications. Spend time working through the examples in the Swift Book [8].

14.5 Many of the examples in this chapter began with `import Foundation`. What is `Foundation` exactly, and what capabilities does importing it bring to our code?

14.6 Other imports commonly seen in Swift applications include `Darwin` and `Cocoa`. What capabilities are gained by importing these?

14.7 What is the type of the expression `9..<12`?

14.8 How is the function `print` from the standard library declared? Give the names, types, and default values of each of its parameters.

14.9 What happens if you try to invoke a function with an `inout Int` parameter with the argument `&5`? Is an error generated? If so, at compile time or run time? Or is a warning issued? If no error occurs, how does Swift handle a change to the corresponding parameter?

14.10 Can an expression of type `Int` be assigned to a variable of type `Int64` without an explicit cast? What are the rules in Swift, if any, regarding which assignments can be made without explicit casts?

14.11 For the string literal `"\u{1f1e8}\u{1f1f7}"`: (a) How many characters does this string have? (b) Express the string as a sequence of UTF-16 code points. (c) Express the string as a sequence of UTF-8 code points. (d) What does this string represent?

14.12 How does one define a function to force every argument in a call to be named? How does one allow every argument in a call not to be named?

14.13 In this chapter, we populated our classes and structs only with properties, methods, initializers, and deinitializers. Give an example that defines a subscript.

14.14 Find out if there is any way for a running Swift application to recover from a precondition failure.

14.15 Research the ways in which Swift errors are similar to, and different from, exceptions in Python.

14.16 For the indirect tree enumeration in this chapter, see if it is possible to assign an object of type tree as a member of the tree's own children. Does the compiler stop you from doing so? Do you get a run time error? Were you able to build a cyclic object? Should this even be possible, given that enumerations have value semantics?

14.17 We mentioned, but did not explain, the `noreturn` keyword. What is this used for?

14.18 What happens if an error is thrown but never caught?

14.19 Evaluate the expression `37 & 95 + 14` in both Python and Swift. Explain in detail why the answers are different.

14.20 The Swift Language Reference describes cases in which the spacing, or lack of spacing, around certain operators changes the meaning of the program. See if you can make a comprehensive list of these cases. Do you find it problematic or confusing? Or pragmatic? Why?

14.21 Do some research on Swift custom operators. Which characters are allowed in operators? Why are prefix and postfix operators not assigned a numeric precedence? If a term consists of both a prefix and a postfix operator, which gets applied first?

14.22 Rewrite the animals example from this chapter—the one we used to illustrate classes—to use protocols instead.

14.23 Use extensions to augment the animals example to vocalize the sounds.

14.24 Provide an analysis of the similarities and differences between Haskell's

```
f :: (Eq a, Ord b) => a -> a -> b -> Bool
f x y z = (x == y) && (z < z)
```

and Swift's

```
func f<T:Equatable, U:Comparable>(x: T, y: T, z: U) -> Bool {
    return x == y && z < z
}
```

14.25 One safety feature not mentioned was the defer statement. Research and given an example of its use.

14.26 Create an example that shows the difference between weak and unowned references.

14.27 Learn about `TaskGroup`s and rewrite the concurrent sum example to use them. Extend the program so that the array contains tens of thousands of elements, and the number of chunks to concurrently sum is variable.

CHAPTER 15

Rust

Rust is "a language empowering everyone to build reliable and efficient software." [99]

First appeared	2012
Created at	Mozilla
Recognized for	Safe systems programming
Notable uses	Servers
Tags	Systems, Safe, Fast
Six words or less	Systems programming without memory errors

Graydon Hoare began working on Rust in 2006 in his spare time while employed at Mozilla. Hoare's goal for the language was to mitigate many of the memory, type safety, and other preventable developer errors common in systems languages like C++ without compromising on speed. Mozilla took interest and supported the development of the language. Now backed by the Rust Foundation, Hoare's project has evolved into a true end-to-end language suitable for writing operating systems, web applications, servers, and high-performance games. Like C, it compiles "to the metal" and has no tracing garbage collector, yet is memory safe—memory leaks and dangling pointers simply do not occur.

Rust is a multi-paradigm language. Portions of the language will be familiar to fans of functional languages, while others will appeal to fans of object-oriented languages. The means by which Rust's *compiler* manages to prevent memory errors and data races is welcome news to fans of low-level programming. The language itself has many fans—Rust has won the title of most-loved programming language on the Stack Overflow Developer Survey every year from 2016-2023.

After a quick look at our traditional teaser programs, we'll dive into the language basics and present Rust's framing of memory management issues, centered around the notions of **ownership**, **borrowing**, and **lifetimes**. We'll examine how ownership can be transferred and borrowed and how the powerful compile-time **borrow checker** ensures memory safety at runtime. We'll cover other features that make Rust an interesting, modern language, such as pattern matching, enums, traits, functional programming features such as closures and iterators, and how concurrent programming is facilitated with threads and channels.

15.1 HELLO RUST

We start with the usual lucky numbers:

```
fn main() {
    for c in 1..=40 {
        for b in 1..c {
            for a in 1..b {
                if a * a + b * b == c * c {
                    println!("{a}, {b}, {c}");
                }
            }
        }
    }
}
```

Rust is statically typed, but does a pretty good job of inferring types. The raw integer literals such as `1` and `40` are inferred to have type `i32` (of 32-bit signed integers). The `..` and `..=` operators produce **iterators** with exclusive and inclusive upper bounds, respectively. `println!` is a **macro** (a form expanded at compile time into another syntactic form). Rust conveniently identifies macros with names name ending with a `!`.

Our permutations teaser illustrates some unique features:

```
fn generate_permutations(n: usize, a: &mut [String]) {
    if n == 0 {
        println!("{}", a.join("\t"));
    } else {
        for i in 0..n {
            generate_permutations(n - 1, a);
            a.swap(if n % 2 == 0 { 0 } else { i }, n);
        }
        generate_permutations(n - 1, a);
    }
}

fn main() {
    let mut args: Vec<String> = std::env::args().skip(1).collect();
    generate_permutations(args.len() - 1, &mut args);
}
```

Here the function `args`, from the module `env` inside the **crate** `std`, produces an iterator over the command line tokens. Since the first token is the program name, we skip over this value before producing a vector from the iterator using `collect()`. This vector is *borrowed mutably* by the permutation generator function. Rust does not have a tracing garbage collector; instead, dynamically allocated objects have the notion of ownership (described later in this chapter) that guarantees memory is automatically freed when the owner goes out of scope. The program is able to be quite short.

Now we are ready for our third standard teaser involving reading from standard input and doing some data science.

```rust
use std::io::{self, BufRead};

struct Player {
    name: String,
    team: String,
    ppg: f64,
}

fn main() {
    let stdin = io::stdin();
    let players: Vec<Player> = stdin
        .lock()
        .lines()
        .filter_map(|line| line.ok())
        .filter_map(|line| {
            let fields: Vec<&str> = line.split(',').collect();
            let games: i32 = fields[2].parse().ok()?;
            let points: i32 = fields[3].parse().ok()?;
            if games < 15 {
                None
            } else {
                Some(Player {
                    name: fields[1].to_string(),
                    team: fields[0].to_string(),
                    ppg: points as f64 / games as f64,
                })
            }
        })
        .collect();

    let mut sorted_players = players;
    sorted_players.sort_by(|a, b| b.ppg.partial_cmp(&a.ppg).unwrap());

    for player in sorted_players.iter().take(10) {
        println!(
            "{:22}{:4}{:8.2}", player.name, player.team, player.ppg);
    }
}
```

This little teaser is certainly loaded with a *lot* of interesting Rust features. Although Rust is a blazingly fast systems programming language with no garbage collector, we see functional programming features: we are splitting, filtering, mapping, and flattening a vector in a streaming fashion, and a (higher-order) sort function is passed a comparator. We did not have to specify the exact amount of memory to allocate nor did we manually deallocate—somehow it all worked. Rust has no exceptions to throw, but we used the very nice `?` operator that immediately returns from a function when its operand is a `None` variant of an optional. Systems programming *can* look as light and uncluttered as dynamic languages. The rest of this chapter will attempt to give you a feel for how Rust is able to make this happen.

15.2 THE BASICS

Rust is both efficient and safe. It is compiled ahead of time and has no garbage collector (and hence no execution pauses due to garbage collection). Safety features include a lack of null references, a requirement that variables must be initialized before use and explicitly marked mutable (`mut`) in order to be updated, and an ownership system that prevents memory leaks, dangling pointers, double frees, and most data races.

The language has a rich set of static types:

- Numbers, partitioned into signed integers `i8`, `i16`, `i32`, `i64`, `isize`; unsigned integers `u8`, `u16`, `u32`, `u64`, `usize`; and floats `f32`, `f64`.
- A Boolean type, `bool`, a character type, `char`, whose instances are 32-bit Unicode scalars, and the type `!`, pronounced "Never," containing no instances.
- Arrays, $[T; n]$, and tuples, $(T_1, \ldots, T_n)$, both of which have a fixed number of elements. Note that `[i32;5]` and `[i32;8]` are distinct types.
- Slices, $[T]$, which are views into arrays and other collections with a contiguous sequence of elements. The built-in type `str` is a slice of bytes (i.e., `[u8]`) that must be valid UTF-8 sequences.
- Functions, closures, structs, enums, unions, traits, and various kinds of pointers.

All variables have type constraints. These can often be inferred, though function parameters must be type-annotated. Some generic functions, though, require additional annotations, most notably `parse`, for making an item from a string, and `collect`, for making an actual collection object from an iterator. We can annotate the variable itself, or add type information to the expression with the **turbofish** (`::<>`) syntax:[1]

```
fn main() {
    let unique: Vec<char> = "Rustacean".chars().collect();
    let list = "Rustacean".chars().collect::<Vec<char>>();
    assert_eq!(unique, list);

    let two_pi: f64 = "6.283185307179586".parse().unwrap();
    let tau = "6.283185307179586".parse::<f64>().unwrap();
    assert_eq!(two_pi, tau);
}
```

Like many languages, Rust allows shadowing in nested blocks. Unlike many languages, variables can be shadowed in the *same* scope:

```
fn main() {
    let x = 5;
    assert_eq!(x, 5);
    let x = "I am a different x!";        // a new variable
    assert_eq!(x, "I am a different x!")  // first variable shadowed
}
```

[1] This moniker was coined by Anna Harren, a.k.a. the guardian angel of the Turbofish [92].

While array sizes are fixed and part of their type, the standard library provides a `Vec` type for growable sequences. Out-of-bounds accesses will cause a run-time **panic** (essentially an unrecoverable error). Slices can be made as a views into either arrays or vectors (or into any object implemented as a contiguous sequence of elements).

```
fn main() {
    let a = [10, 20, 30, 40, 50];      // type [i32; 5]
    let a_slice = &a[2..4];            // type &[i32]
    let v = vec![10, 20, 30, 40, 50]; // type Vec<i32>
    let v_slice = &v[2..4];            // type &[i32]
    assert_eq!(a_slice, &[30, 40]);
    assert_eq!(v_slice, &[30, 40]);
    // println!("{}", v[99]);          // panics
}
```

In the script above, the type of the array is `[i32;5]` and the type of the slice itself is `[i32]`. A variable cannot store a slice directly—slices have variable size—but only a reference (denoted by `&`) to a slice; hence variable `a_slice` is constrained to the type `&[i32]`.

Rust strings are sequences of validly encoded UTF-8 byte sequences. Instances of the built-in type `str` are slices into such sequences, while the type `String` from the standard library is used for growable, potentially mutable character sequences. Use methods `bytes` or `chars` over either kind of string to obtain iterators, then access components with `nth` or `last`. The access methods return option values (sum types with `Some` and `None` variants):

```
fn main() {
    let s = "Ć←afé";                          // inferred type is &str
    assert_eq!(s.len(), 9);                   // 9 bytes
    assert_eq!(s.bytes().count(), 9);         // 9 bytes
    assert_eq!(s.chars().count(), 5);         // 5 chars
    assert_eq!(s.chars().nth(2), Some('a'));  // 3rd char
    assert_eq!(s.chars().last(), Some('é'));  // last char
    assert_eq!(s.chars().nth(99), None);

    let mut note = String::from(s);           // str -> String
    note.push('!');                           // push a char
    for c in note.chars() {
        println!("{c}");
    }
}
```

Statements include declaration statements, expression statements, and the empty statement. Declarations introduce variables or other **items** (types, modules, traits, impls, modules, constants, among others) into the current block. Expression statements simply evalute expressions, which may have side effects. Blocks consist of statements *separated* by semicolons, the last of which, if it is an expression statement, becomes the value of the block. If the last statement is an empty statement, the block's value is `()`. Function bodies are blocks, so the final expression becomes the return value of the function. Many constructs that are *statements* in other languages (e.g., `if`, `match`, `while`, `for`) are expressions in Rust, so these have values too:

```
fn main() {
    // Value of block expression is the last expression
    let eight = {
        let x = 3;
        x + 5                       // No final semicolon, produces 9
    };
    let unit = {
        let x = 3;
        x + 5;                      // Final semicolon, produces ()
    };
    assert_eq!(eight, 8);
    assert_eq!(unit, ());

    // A plain loop expression produces ! if it runs forever,
    // or the value of the break expression if it breaks out
    let mut count = 0;
    let loop_result = loop {
        count += 1;
        if count == 10 {
            break count * 2;
        }
    };
    assert_eq!(loop_result, 20);

    // While (and for) expressions produce ()
    let mut count = 1;
    let while_result = while count < 10 { count += 1 };
    assert_eq!(while_result, ());

    // Match expressions are straightforward
    let match_result = match count {
        0 | 1 => "binary-ish",
        2 => "couple",
        3..=8 => "a few",
        _ => "lots"
    };
    assert_eq!(match_result, "lots");
}
```

Rust uses the term `closure` for anonymous functions that "capture" the environment in which they were defined.

```
fn main() {
    let x = 5;
    let times_five = |y| y * x;        // captures first x
    let x = "I am a different x!";     // shadows first x
    assert_eq!(50, times_five(10));    // uses (the captured) first x
    assert_eq!(x, "I am a different x!")
}
```

The remainder of this chapter will focus on a few things that make Rust special. Somehow this systems programming language with no garbage collector manages to guarantee the absence of memory leaks and dangling pointers. And somehow the high-level features—pattern matching, closures, iterators, generics, and safe concurrency—are implemented with **zero cost abstractions**, that is, their runtime performance has no overhead compared to crafting these features from lower-level constructs. We'll begin with memory safety.

15.3 OWNERSHIP

In Chapter 13 we saw how C is prone to memory leaks, dangling pointers, and double frees due to its placing the burden of memory management on the programmer. C programmers have to remember to free memory that is no longer needed, ensure referents never live longer than the pointers referencing them, and be careful free resources exactly once. As memory issues lead to security vulnerabilities, many languages disallow manual memory management and require a garbage collector to run in the background. Others skip the garbage collector but build in some safety features.[2] C++, for example, is a near-superset of C that provides **smart pointers** and destructors (akin to Swift's deinitializers) that help somewhat; however, one still needs discipline to use those features properly, and when misused, all of the problems of C still exist. Swift avoids memory problems by building reference counting in, but asks programmers to remember to use weak references to avoid cycles. Rust has something new—an *ownership* system to avoid memory problems, all entirely handled by the compiler!

Here's the idea: every resource has a single owner, a variable, and when the owner "goes away", the resource is **dropped**, meaning its storage is reclaimed. Here's an illustration with strings and vectors:

```
fn main() {
    let mut picks = vec![8, 21, 5, 1, 2];
    say_hello();
    picks.push(34);
    println!("{:?}", picks);
} // picks goes out of scope and is dropped (freed) here

fn say_hello() {
    let mut greeting = String::from("Hello");
    greeting.push_str(", world!");
    println!("{}", greeting);
} // greeting goes out of scope and is dropped (freed) here
```

As data is dropped when the owner goes out of scope, Rust prohibits a value from having multiple owners. Specifically, assignment and argument passing either **move** (transferring ownership) or **copy** (making a new resource) values. An attempt to access a moved resource is a compile-time error:

[2] While automatic garbage collection does make programs a bit easier to write, they may incur unacceptable overhead in certain environments, particularly those that must meet real-time constraints. Garbage collectors can kick in at inopportune times, causing unpredictable latency. Performance could be impacted since collectors chase references through large memory pools scattered throughout memory, taking away cache advantages of modern processors that rely on locality of reference.

```
fn main() {
    let a = String::from("hi");    // owned by variable a
    println!("{a:?}");
    let b = a;                     // now owned by variable b
    // println!("{a:?}");          // WILL NOT COMPILE, string was moved
    println!("{b:?}");             // This is ok, b is the owner now
}
```

Generally, items with a small fixed size such as numbers, booleans, and characters will be copied, but vectors, strings, hashmaps, and any dynamically allocatable structure that can grow and shrink will be moved. Copying such objects can be quite expensive! Resources are moved unless their type has the `Copy` trait; we'll explain what this means when we cover traits in Section 15.6.

But moving may not always be the best choice. If a value is passed as an argument to a function (with a move that transfers ownership) but needs to be used after the call, the called function must transfer ownership back to the argument, which can clutter the code and look quite strange. Therefore, in addition to moving and copying, Rust allows **borrowing** by assigning or passing **references**:

```
fn main() {
    let a = String::from("hi");    // owned by variable a
    println!("{}", is_long(&a));   // pass a reference to borrow
    println!("{a}");               // a still owns the string
}

fn is_long(b: &String) -> bool {   // b is just borrowing
    b.len() > 10                   // (i.e., b is not the owner)
}                                  // done borrowing
```

Borrowing mutably requires the `&mut`-style reference:

```
fn main() {
    let mut a = String::from("wo"); // owned by variable a
    add_ah(&mut a);                 // pass a reference to borrow
    println!("{a}");                // a still owns the string
}

fn add_ah(b: &mut String) {         // b is just borrowing
    b.push_str("ah")                // (i.e., b is not the owner)
}                                   // done borrowing
```

If a resource is borrowed mutably, *no other borrows* may be in effect during the mutable borrow. A resource can, however, have several simultaneous immutable borrows. Language rules ensure there is no way a resource can be modified through one reference while another reference is surprised by the change. Rust avoids shared mutable data.

References are not the only way to refer to data in Rust; you'll also find various kinds of **smart pointers**. A smart pointer is an object that contains a pointer to, or address of, a variable, along with some additional metadata. Most smart pointers *own* their referenced data, while references only *borrow*. Smart pointers are often used for data that must be

heap-allocated since its size is not known at compile time. Strings and vectors are smart pointers: their data is heap-allocated as it can grow. The pointer "object" holds the pointer to the contents, as well as the current size and capacity. Rust has a rich set of different kinds of smart pointers, of which we'll cover only two in subsequent sections—boxes and reference-counted pointers.

15.4 LIFETIMES

Rust's compile-time-enforced ownership and borrowing rules prevent memory leaks, double frees, and data races. To prevent dangling references, the Rust compiler analyzes **lifetimes** to ensure a reference never outlives its referent. The following program fails to compile since there's an attempt to use x after its referent has gone away:

```
fn main() {                     // WILL NOT COMPILE
    let x;
    {
        let y = 5;
        x = &y;                 // ERROR `y` does not live long enough
    }
    println!("{x}");            // (Would be dangling if allowed)
}
```

Dangling references are only (theoretically) possible when references are returned from functions, or, as in the example above, assigned to in some nested scope. As space is limited, we'll explore only (the more interesting case of) how Rust handles returning references from functions. If a reference is returned from a function,[3] correctness requires that we know that the referent (1) lives as long as the whole program (has the `'static` lifetime), or (2) has a lifetime related to an input parameter that is a reference. Otherwise, we'd be returning a reference to a resource owned by a variable within the function. For example, Rust rejects the following program:

```
fn five() -> &i32 {             // WILL NOT COMPILE
    let x = 5; &x
}
```

We get the compile-time error "`expected named lifetime parameter`" on the function return type. Let's provide a lifetime name as suggested. By convention lifetime names are `'a`, `'b`, `'c`, etc.:

```
fn five<'a>() -> &'a i32 {      // STILL WILL NOT COMPILE
    let x = 5; &x
}
```

Now the error is "`cannot return reference to local variable` `x`" and the help text you'll see from a good development environment is "returns a reference to data owned by the current function." As expected. But why did Rust first ask for a lifetime annotation? To help the compile-time analyzer—the **borrow checker**—we have to provide some information about the lifetime of the returned reference. Often this is inferable! If the function has exactly one reference parameter, or is a *method* (with a parameter of `&self` or `&mut self`,

[3] We're technically looking at top-level functions, defined with `fb` here, but the way things work with closures are not too far off. The idea is the same, but there are some complexities involving environment capture that we need not go over on a first pass.

introduced in the next section) the return value will be implicitly tagged with the lifetime of that parameter.[4] Now if there are multiple input reference parameters, what can we say about the return value lifetime? The following program is rejected:

```
fn shorter_string(s: &str, t: &str) -> &str {  // WILL NOT COMPILE
    if s.len() < t.len() { s } else { t }
}
```

Again we see the error "`expected named lifetime parameter`" with help text "this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from *s* or *t*." Rust cannot infer at compile time which lifetime is correct for further analysis. The common solution in this case is to provide the same lifetime annotation throughout. This is interpreted by Rust to mean that it can assume, for the return value, the shorter lifetime of the two input parameters:

```
fn shorter_string<'a>(s: &'a str, t: &'a str) -> &'a str {
    if s.len() < t.len() { s } else { t }
}

fn main() {
    let s = "Hello";
    let t = "Hi";
    assert_eq!(shorter_string(s, t), "Hi");
}
```

The built-in lifetime `'static`, briefly mentioned above, is used for constants and string literals. It is often inferred but must be provided when a function returns a reference to a static resource:

```
const FIVE: i32 = 5;

fn five() -> &'static i32 {
    &FIVE
}
```

We've covered only the basics of this complex topic. We've provided a few exercises for further explorations, but as always you're encouraged to dive into the documentation and create complex projects of your own.

15.5 STRUCTS AND ENUMS

Rust's approach to bundling data and computation—a common organizing principle in modern software engineering—is to define data with structs or enums and provide (zero or more) **implementation blocks** with methods and associated functions. Structs are product types with named fields;[5] enums are sum types with named variants.

[4] This is not to say the return value is forced to have a new lifetime: you can't change lifetimes in code. Rather, lifetime annotations are used by Rust's borrow checker to detect and prevent dangling references, and the information provided by the rules above gives the borrow checker sufficient information to do its job.

[5] There's also a tuple struct with unnamed fields, accessed via `.0`, `.1`, etc., but we'll not be focusing on them in this chapter.

```rust
struct Rectangle {
    width: f64,
    height: f64,
}

impl Rectangle {
    fn new(width: f64, height: f64) -> Self {
        Rectangle { width, height }
    }

    fn dimensions(&self) -> (f64, f64) {
        (self.width, self.height)
    }

    fn area(&self) -> f64 {
        self.width * self.height
    }
}

fn main() {
    let r = Rectangle::new(5.0, 10.0);
    println!("Area: {}", Rectangle::area(&r));  // Rare syntax
    println!("Area: {}", r.area());             // Common syntax
    assert_eq!(r.dimensions(), (5.0, 10.0));
    println!("{}", r.width);
}
```

In the implementation, `Self` refers to the type to which the block applies. If the first parameter of an associated function has the type `&Self`, the function is called a **method**, and you can write the parameter specification as just `&self`, which sugars `self: &Self`. All associated functions, whether methods or not, can be accessed outside the implementation block with the `::` operator. Methods, however, can be invoked with dot notation.

Methods, by convention, are parameterized with a *reference* to `self`. We want borrowing: transferring ownership or copying is likely inefficient for method calls. So why then did we write `r.dimensions()` rather than `(&r).dimensions()`? Rust provides *automatic referencing and dereferencing*. In many, but not all, cases Rust will add or remove reference operators (`&`) and dereference operators (`*`) to make expressions "work." Our method calls illustrated automatic referencing; the bodies of the methods illustrated automatic dereferencing, simplifying the body of `area` from the verbose `(*self).width * (*self).height`.

Fields of a struct, and its associated functions, are visible only within the module that contains the struct. Since we did not use a module declaration the "module" is the whole file. Modules do not have to line up with files; they can be defined within a file like so:

```rust
mod shapes {
    pub struct Rectangle {
        width: f64,          // we decided not to use pub here
        height: f64,         // nor here
    }
```

```
    impl Rectangle {
        pub fn new(width: f64, height: f64) -> Self {
            Rectangle { width, height }
        }

        pub fn dimensions(&self) -> (f64, f64) {
            (self.width, self.height)
        }

        pub fn area(&self) -> f64 {
            self.width * self.height
        }
    }
}

fn main() {
    let r = shapes::Rectangle::new(5.0, 10.0);
    println!("Area: {}", r.area());
    assert_eq!(r.dimensions(), (5.0, 10.0));
}
```

Though not shown in our example, selected fields of a struct can be marked `pub` to expose them outside the module, and associated functions and methods need not be marked `pub` if they exist only to service other public functions. It is even possible to define other items (structs, enums, etc.) at the top level of the module that are not marked `pub`. They will be "private to" the module (accessible only within).

Rust's enums are traditional tagged unions, with a syntax rather similar to Swift's, and are typically pattern matched. As expected from a safe language, it is a compile-time error if the `match` expression arms are not exhaustive. Here's the Rust version of the Swift token enum from the previous chapter:

```
enum Token {
    Plus,
    Minus,
    Number(f64),
    Identifier(String),
}

impl Token {
    fn lexeme(&self) -> String {
        match self {
            Token::Plus => String::from("+"),
            Token::Minus => String::from("-"),
            Token::Number(value) => value.to_string(),
            Token::Identifier(name) => name.clone(),
        }
    }
}
```

```
fn main() {
    let expression: Vec<Token> = vec![
        Token::Number(3.0),
        Token::Plus,
        Token::Identifier("x".to_string()),
    ];
    let text = expression.iter()
        .map(|t| t.lexeme())
        .collect::<Vec<String>>()
        .join(" ");
    assert_eq!(text, "3 + x");
}
```

Matching and binding can also take place in the tests of `if`, `while`, and `for` expressions, where a match failure acts as a false condition or a loop termination condition:

```
enum RaceResult {
    Time(f64),
    DidNotFinish,
}

fn main() {
    let alice_result = RaceResult::Time(53.23);
    let _bob_result = RaceResult::DidNotFinish;

    if let RaceResult::Time(alice_time) = alice_result {
        println!("Alice's time: {}", alice_time);
    }
}
```

Like many other languages with static type checking, Rust has generics. The type `Vec<T>` above is one example. Two very import generic types are the option and result enums; they are quite similar to their Swift counterparts we encountered in the last chapter. `Option<T>` has variants `Some(T)` and `None`. `Result<T, E>` has variants `Ok(T)` and `Err(E)`.

While frequently pattern-matched like all enums, options, and results can be used with the fabulous `?` operator. The expression o? for optional o produces the contents of the `Some` variant for present optionals and *immediately returns* `None` from the current function otherwise. Similarly, r? for result r produces the `Ok` variant if r is okay; otherwise, it immediately produces the `Error` variant:

```
use std::num::ParseIntError;

fn add(s: &str, t: &str) -> Result<i32, ParseIntError> {
    let x = s.parse::<i32>()?;
    let y = t.parse::<i32>()?;
    Ok(x + y)
}
```

```
fn main() {
    assert_eq!(add("1", "2"), Ok(3));
    assert!(add("1", "abc").is_err());
}
```

Options and results are quite flexible: each have methods `unwrap` (which should be used rarely, as it panics on `None`s and error variants), `unwrap_or`, `unwrap_or_else`, and `unwrap_or_default`. Results can be converted to option with the `ok` method.

Structs and enums are stack-allocated, which raises the question of how one might build recursive structures, since 'linked" structures require dynamically (heap) allocated nodes. The simplest approach is to use the `Box` smart pointer. Here's a simple tree struct we've seen in previous chapters:

```
enum Tree<T> {
    Empty,
    Node(Box<Tree<T>>, T, Box<Tree<T>>),
}

use Tree::{Empty, Node};

impl<T> Tree<T> {
    fn size(&self) -> usize {
        match self {
            Empty => 0,
            Node(left, _, right) => left.size() + 1 + right.size(),
        }
    }
}

fn main() {
    let t: Tree<i32> = Node(
        Box::new(Node(Box::new(Empty), 3, Box::new(Empty))),
        7,
        Box::new(Empty),
    );
    assert_eq!(t.size(), 2);
}
```

You can think of a box as a heap-allocated pointer. A box owns it data, so when it goes away, the referenced data is dropped. Boxes are suitable for simple lists and trees, but for complex objects such as graphs, with multiple references *to* nodes being possible, the reference counted smart pointer, `Rc<T>` is preferred. The ideas behind reference counting were covered in detail in the previous chapter on Swift, so we won't repeat them here. The code looks somewhat different, of course, in Rust, you obtain weak pointers with the function `Rc::downgrade`. For allocations, you can directly access the reference counts via `Rc::strong_count` and `Rc::weak_count`. As always, the Rust official documentation is your source for detailed information.

15.6 TRAITS

Rust's version of Ruby's mixins, Java's interfaces, Haskell's typeclasses, and Swift protocols is the **trait**. Traits specify behaviors that can be shared by different types. Let's go straight to a simple example:

```
trait Speaker {
    fn name(&self) -> &'static str;
    fn sound(&self) -> &'static str;
    fn speak(&self) -> String {
        format!("{} says {}", self.name(), self.sound())
    }
}

struct Horse {
    name: &'static str,
}

impl Speaker for Horse {
    fn name(&self) -> &'static str { self.name }
    fn sound(&self) -> &'static str { "neigh" }
}

struct Cow {
    name: &'static str,
}

impl Speaker for Cow {
    fn name(&self) -> &'static str { self.name }
    fn sound(&self) -> &'static str { "moooo" }
}

// Static dispatch example (trait bounds)
fn print_dialog_line<T: Speaker>(animal: &T) {
    println!("Narrator: {}", animal.speak());
}

fn main() {
    let h = Horse { name: "CJ" };
    let c = Cow { name: "Bessie" };

    print_dialog_line(&h);

    // Dynamic dispatch example (trait objects)
    let animals: Vec<&dyn Speaker> = vec![&h, &c];
    for animal in animals.iter() {
        println!("{}", animal.speak());
    }
}
```

Here the `Speaker` trait gives only the specifications of the methods `name` and `sound`, but supplies a default implementation for `speak`. The types `Horse` and `Cow` implement the trait. The script illustrates both static dispatch, via generic trait bounds, and dynamic dispatch, through **trait objects**. The `dyn` keyword creates a trait object by combining a pointer to some value of a type implementing the trait as well as a run-time table of method implementations for that type. This allows methods to work for any type implementing the trait, even those that have yet to be written.

A number of traits are defined in Rust's standard library; a few are described in Table 15.1. The equality and ordering traits make the relational operators (`==`, `<`, etc.) available for your type. The first eight of these may, in many situations, be implementable using the `#derive` **attribute**, providing the "obvious" implementation of the trait methods. Here's how a few are derived:

Trait	Description	Req'd methods
`PartialEq`	has a partial equivalence relation	`eq`
`Eq`	has an equivalence relation (requires `PartialEq`)	—
`Debug`	formattable with the `"{:?}"` (detail) specifier	`fmt`
`Clone`	copyable via custom behavior	`clone`
`Copy`	bitwise copyable (requires `Clone`)	—
`Default`	has a default value for the type	`default`
`Ord`	has a total order (requires `PartialOrd` and `Eq`)	`cmp`
`PartialOrd`	has a partial order	`partial_cmp`
`Display`	formattable with the `"{}"` (user-friendly) specifier	`fmt`
`Iterator`	iterable	`next`

Table 15.1 A few Rust predefined traits

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Default)]
struct Tag {
    pub id: u64,
    pub label: &'static str,
    pub featured: bool
}

fn main() {
    let t1 = Tag { id: 1, label: "Rust", featured: true };
    let t2 = Tag { id: 2, label: "Swift", featured: false };
    let t3 = Tag::default();
    assert!(t1 != t2);
    assert!(t1 < t2);
    assert_eq!(t3, Tag { id: 0, label: "", featured: false });
    assert_eq!(format!("{:?}", t1),
        "Tag { id: 1, label: \"Rust\", featured: true }");
}
```

A number of traits appear in Rust's `std::ops` module, including `Add`, `Sub`, `Mul`, `Div`, `Neg`, `Not`, `Shr`, `BitAnd`, and more. As you may guess from your Lua and Python experience, implementing these traits will allow operator overloading for your own types.

15.7 FEARLESS CONCURRENCY

The degree to which safety emerges naturally from Rust's type system and its ownership model extends to concurrent programming. **Data races**, where two concurrent threads have unsynchronized access to the same memory location and at least one of which is a writer, are impossible in Rust, since a mutable reference to a resource must be exclusive. As many common concurrent programming errors in other languages are avoided in Rust, Rust developers can approach concurrent programming fearlessly, or at least with much less trepidation than they might be used to. Rust's standard library has rich support for various concurrent paradigms; we have space for only brief example. Let's explore Rust's message passing ("share nothing") capabilities. Rust uses channels, much like Go:

```
use std::thread;
use std::sync::mpsc;

fn sum(a: &[i32], tx: mpsc::Sender<i32>) {
    let result: i32 = a.iter().sum();
    tx.send(result).expect("Error sending result");
}

fn main() {
    const SIZE: usize = 10000;
    let nums: [i32; SIZE] = core::array::from_fn(|i| i as i32);

    let (tx1, rx) = mpsc::channel();
    let tx2 = tx1.clone();
    let handle1 = thread::spawn(move || { sum(&nums[..SIZE/2], tx1); });
    let handle2 = thread::spawn(move || { sum(&nums[SIZE/2..], tx2); });

    let result = match (rx.recv(), rx.recv()) {
        (Ok(result1), Ok(result2)) => result1 + result2,
        _ => { eprintln!("Error receiving results"); return; }
    };
    match (handle1.join(), handle2.join()) {
        (Ok(_), Ok(_)) => {},
        _ => { eprintln!("Error joining threads"); return; }
    }
    assert_eq!(result, 49995000);
}
```

This example uses an `mpsc` channel, which stands for **m**ultiple **p**roducer, **s**ingle **c**onsumer. It's almost the same program we saw in our chapter on Go: two threads sum half of a large array, sending their partial sums through a channel to the main thread which combines them. The variables *tx* and *rx* abbreviate *transmitter* and *receiver*. The example also introduces the `move` prefix on closures to allow our two threads to take ownership of the two transmitters.

Rust also allows shared-state concurrency, with data protected via the `std::sync::Mutex` and a large number of atomic types from module `std::sync::atomic`. These low-level mechanisms are quite involved and take a while to lean, so do consult [100, Sec 16.3] for

a tutorial. We've left quite a bit for you to explore, notably the `Send` and `Sync` traits, the use of *atomic* reference counted smart pointers (`Arc<T>`), and the `rayon` crate for parallel computing. Rust offers so much, but programmers can always introduce logic errors which the compiler, to be fair, will be unable to detect. Still, it is worth celebrating the number of conditions that can be detected.

15.8 MACROS

Like Julia and Clojure, Rust has hygienic macros that work on abstract syntax trees. They're easy to recognize in Rust, as their names must end with `!`, as we've seen in `println!` and `vec!`. As we covered hygienic macros in great detail in our chapter on Julia, we'll keep our treatment here short and just introduce Rust macros with a very simple example:

```
macro_rules! double {
  ($x:expr) => (2*$x);
}

fn main() {
  println!("{}", double!(3 + 5));
}
```

We've defined a macro called `double!` whose argument x is given the **designator** `expr`. Designators are actual Rust syntactic forms, others are: `ident` for identifiers, `stmt` for statements, `pat` for patterns, `literal` for literals, `block` for blocks, `ty` for type, and many more. In addition, Rust also features three types of what it calls **procedural macros** that also generate Rust code at compile time, but in different ways. We'll give you a chance to explore some more complex macros in the end-of-chapter exercises.

RUST WRAP UP

In this chapter, we were introduced to Rust. We learned that:

- Rust is a statically typed, low-level systems programming language with no tracing garbage collector, known for its compile-time mechanisms for preventing most memory leaks, dangling pointers, and data races.
- Although a systems-level language, Rust sports several features from functional programming such as closures and iterators, and dynamic dispatch from object-oriented programing, all as zero-cost abstractions.
- The three ownership rules are: (1) each value has an owner, (2) there can only be one owner at a time, and (3) when the owner goes out of scope, the value will be dropped.
- The two borrowing rules are: (1) At any time a value may have a single mutable reference *or* any number of immutable references, and (2) references can not outlive the owned value.
- On assignment, values are either moved or copied. A type must implement the `Copy` trait to be copied. Borrowing is an alternative to moving or copying.
- Lifetimes prevent dangling pointers. The compiler can usually determine lifetimes automatically, but the programmer sometimes has to add lifetime annotations.

- User-defined data types are generally built with structs and enums. Behaviors are added with implementation blocks that contain associated functions, some of which are known as methods.
- The `Box` and `Rc` smart pointers are often used when heap allocation is necessary.
- Traits are Rust's way of defining behavior that is shared among types. Static dispatch is created with functions with parameters typed with trait bounds; dynamic dispatch is also possible, and make explicit with the `dyn` keyword. A number of built-in traits exist, several of which can be implemented with the `derive` attribute.
- Rust's compile-time support for safe concurrency has been dubbed *fearless*, since the ownership and borrowing rules naturally prevent errors common to multi-threaded and distributed programming.
- Rust's macros are hygienic.

For further study in Rust, including language features not covered in this chapter as well as links to articles, tutorials, official documentation, and open-source projects using Rust, see the Rust page in the book's companion GitHub repository at https://github.com/rtoal/ple/tree/main/rust.

EXERCISES

Now it's your turn. Continue exploring Rust with the activities and research questions below, and feel free to branch out on your own.

15.1 Find the online Rust Playpen and use it to run the right triangle program at the beginning of this chapter. In addition to running the program, take a look at the LLVM IR, Wasm, and ASM translations (just for fun).

15.2 Find the Rust home page, the online *Rust Book*, and the (also online) *Rust Reference*. Read or skim as desired.

15.3 Install Rust on your own machine and build native executables for the first three programs in this chapter.

15.4 In our permutations example, we defined the `generatePermutations` function to first take the length and then the (mutable) vector. If we switched the order of the parameters, and the expressions in the two calls, the program would not compile. Why not? How would you have to rewrite the code so that it would work?

15.5 In the top-ten scorers application from the beginning of the chapter, we used a plain `unwrap` method call in our sorting comparator. After all, any failure in `partial_cmp` would be the program could not generate a proper report, so a panic might be justifiable. However, there are ways to make the program exit gracefully with an error message on failure to sort. Implement such a strategy of your choice.

15.6 Rust claims it `char` type represents Unicode scalars (code points in the ranges $0..d7ff_{16}$ and $e000_{16}..10ffff_{16}$ inclusive). Test this out. See if the Rust compiler accepts `'\u{e000}'` but rejects `\u{dd0d}`.

15.7 Write an iterative function to compute the greatest common divisor of two `u128` arguments.

15.8 Write an experiment to determine in a closure that captures a variable takes ownership of the value away from the variable. If not, how can you force it to take ownership?

15.9 The line of code `let x = &5;` is actually legal in Rust. But how? References can only reference variables, i.e., items stored in memory with an address and size. So this line of Rust actually sugars two statements. Which ones?

15.10 Do some research to see if there is any Rust code in the wild with a function that uses three distinct lifetime parameters (more than likely `'a`, `'b`, and `'c`).

15.11 Rust has a `let-else` expression. How does it work? Craft an example illustrating its use.

15.12 Translate to Rust the type of two-dimensional vectors introduced in the Lua chapter. Use the traits from `std::ops` that enable operator overloading. Implement `Copy`, `Clone`, `Display`, `Debug`, `PartialEq`, and `Eq`.

15.13 Smart pointers implement the `Deref` and `Drop` traits. Research these traits.

15.14 We did not describe the differences between `Eq` and `PartialEq` and between `PartialOrd` and `Ord`. What are these differences?

15.15 What is the predefined trait `Hash` for? Can it be used with `#[derive]`?

15.16 Port the prime number printer application from the Java chapter to Rust.

15.17 Port the restaurant simulation from the Java chapter to Rust.

15.18 Write a macro that accepts one or more statements, and executes exactly one of them arbitrarily.

15.19 The kind of macro we described in this chapter was a declarative macro (defined with `macro_rules!`. But Rust also has three types of procedural macros. Craft examples for each type of procedural macro.

15.20 Although designed as a very safe language, Rust has a way to write unsafe code, primarily for interfacing with C. How is unsafe code written in Rust? What are some examples of unsafe constructs?

15.21 As time allows, research the Zig language and perform a comparison of this language with Rust. Are the two languages in competition, or do they target different use cases?

CHAPTER 16

Additional Languages

The preceding fourteen chapters took us on a tour of several modern languages, introducing us to concepts and practices important to both professional developers and students of computer science. We make no apologies for leaving out your personal favorite language, or languages you feel we *should* have covered. However, we recognize that any small selection of languages will miss several interesting topics. So in this chapter, we're extending our tour with a few dozen lightning stops, visiting a mix of historically important, influential, experimental, and popular languages not covered in the main tour.

As space is limited, very few of the languages in our extended tour come with inline example code. We have, however, made several runnable examples available at the companion GitHub repository (`http://github.io/rtoal/ple`). The repository includes, at times, additional language information for the languages covered here, as well as descriptions and examples for a number of additional languages we feel are worth a look.

16.1 THE CLASSICS

Our first few lightning stops take us through some of the classics. These "early" languages were often the first of their kind, and their historical importance and influence on subsequent languages cannot be overstated. Some of the languages in this group survive today in modern forms, while some have simply morphed into successor languages.

16.1.1 Fortran

Fortran debuted in the mid-1950s as an "automatic coding system"—automatic in the sense that the system, not the programmer, produced the actual machine code (considered at the time to be *the* code). It debuted at a time when the need for high-level languages was still being questioned, so the designers made every effort to ensure the compiler and the generated code was as efficient as possible.

The first Fortran, Fortran I, looked very different from modern versions: code was not indented, recursion was prohibited, and conditionals and while loops were made by referencing line numbers. Here's a short Fortran I program, taken from [34]:

```
C       PROGRAM FOR FINDING THE LARGEST VALUE
C     X      ATTAINED BY A SET OF NUMBERS
        DIMENSION A(999)
        FREQUENCY 30(2,1,10), 5(100)
        READ 1, N, (A(I), I= 1,N)
    1   FORMAT (I3/(12F6.2))
        BIGA= A(1)
    5   DO 20 I= 2,N
   30   IF (BIGA-A(I)) 10,20,20
   10   BIGA= A(I)
   20   CONTINUE
        PRINT 2, N, BIGA
    2   FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
        STOP 77777
```

The instruction on the line labeled "30" is a **computed goto**. It evaluates a conditional expression and transfers control to one of (up to) three instructions according to whether the condition is negative, zero, or positive.

Fortran has evolved a great deal throughout the years. Line numbers, **fixed formatting** (did you notice that all instructions started in column 7 above?), and the computed goto are part of the language's past. Today's Fortran sports many modern features including support for object-oriented and parallel computing. The language remains widely used in the areas of numerical and scientific computing where high performance is a major concern. Modern dialects of Fortran include excellent support for parallel computing through distribution of data across processors.

16.1.2 COBOL

Another language from the 1950s, COBOL, emerged to support business and finance applications. It is best known for its English-like syntax (with `MOVE...TO` for assignment and operators with names including `ADD`, `MULTIPLY`, `GREATER THAN`), a large number of reserved words, and its use of the new (at the time) record type. COBOL had a run as one of the most widely used languages in the world. While less popular than in the past, COBOL-based applications can still be found today powering many business software systems.

COBOL was designed and developed by CODASYL, a consortium of government and industry members. Perhaps the most famous technical consultant to the group was Grace Hopper, the inventor of FLOW-MATIC (1995), the first English-like data processing language, and the A-0 System (1951), credited as being the first compiler for an electronic computer [101]. Hopper's FLOW-MATIC was a major influence on COBOL. Hopper herself played an active role in programming languages in her career, including a period as director of the U.S. Navy's Programming Languages Group [119].

16.1.3 Lisp

The 1950s also brought us Lisp, a language widely praised [35] for its elegance, expressiveness, beauty, and homoiconicity, and sometimes teased for its unusual parentheses-laden syntax. It has been called "the greatest single programming language ever designed." [105]

Much of what makes Lisp Lisp was covered in our chapter on Clojure, and you can read more about its essential character and the reverence in which it is sometimes held in [37] and [36]. The latter reference is required reading for those interested in programming language history, as it relates the story of Steve Russell's realization that Lisp was not just a notation for writing functions, as McCarthy himself believed, but a true programming language that could be implemented.

Russell in fact *did* write the first Lisp interpreter. It didn't take too long to write, as McCarthy's language specification was very brief. Alan Kay noted, "the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were Maxwell's Equations of Software!" A few functions, including `eval` and `apply`, were all that was needed to define a complete programming language. For an in-depth analysis of the half page of code and the rationale for Kay's realization, see [79].

16.1.4 Algol

The language Algol (short for Algorithmic Language), perhaps best known as the direct ancestor of many of today's most popular languages (including C, C++, C#, Java, and JavaScript), began as the product of a joint committee of European and American Computer Scientists in 1958. The publication of *The Algol 60 Report* introduced the world to now-ubiquitous method for specifying syntax, Backus-Naur Form (BNF). Among Algol's innovations were block structure and lexical scope, characteristics frequently taken for granted today.

The second Algol, Algol 60, was not only influential but good. Sir Tony Hoare, on the impact of Algol 60, remarked, "Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors" [47]. In fact, Algol 68, the language designed to succeed Algol 60, gained a degree of infamy as a large and complex language, and acquired quite a number of famous detractors [24].

16.1.5 Simula

In the early 1960s, Ole-Johan Dahl and Kristen Nygaard created an extension of Algol 60 which came to be known as Simula. Simula was originally developed as a vehicle for writing simulations, a design goal its authors ultimately satisfied with classes, objects, inheritance, and coroutines [124].

A second iteration of the language, Simula 67, was the first of a number of languages, including Mesa, CLU, Modula, and Euclid, which highlighted the importance of encapsulation in the construction of software systems.

Simula is now considered to be the first object-oriented programming language. Its influence on its better-known immediate object-oriented descendants, Smalltalk, C++, and Java is difficult to overstate. A brief history of the language can be found at [61] and a textbook with a language description and a wealth of examples is [93].

16.1.6 Smalltalk

Many of the innovations pioneered by Simula were more fully developed, refined, and expanded by a group at the Xerox Palo Alto Research Center (PARC), led by Alan Kay, Dan Ingalls, and Adele Goldberg in the early 1970s. Smalltalk was conceived and grew

up in the same laboratory as did the mouse-driven user interface, Ethernet, and the laser printer [86]. It found many applications through the years in education, financial analysis and modeling.

Although a great admirer of LISP, Kay's motivation to create Smalltalk stemmed in part from realizing that

> ...the surface beauty of LISP was marred by some of its key parts having to be introduced as "special forms" rather than as its supposed universal building block of functions. The actual beauty of LISP came more from the *promise* of its metastructures than its actual model. I spent a fair amount of time thinking about how objects could be characterized as universal computers without having to have any exceptions in the central metaphor [60].

The notion of objects as "universal computers" arose from Kay's application of the notion, credited to Bob Barton by Kay, that the key to successful decomposition of a system is to make the parts resemble the whole. Decomposing a computer into dissimilar data structures and algorithms violates this idea; decomposing it into little computers (bundles of data and code now known as **objects**) does not. Smalltalk, therefore, eventually came to follow six principles, recorded in [60]:

1. Everything is an object.
2. Objects communicate by sending and receiving **messages** (in terms of objects).
3. Objects have their own **memory** (in terms of objects).
4. Every object is an instance of a **class** (which must be an object).
5. The class holds the shared **behavior** for its instances (in the form of objects in a program list).
6. To evaluate a program list, control is passed to the first object and the remainder is treated as its message.

16.2 CRYSTALLIZATIONS OF STYLE

Some languages are strong exemplars of a particular style or paradigm, often featuring small, elegant specifications that take up only a few pages. Nearly every computation in such a language looks about the same. Everything is an object, or everything is an array. Each computation step generally has the same form: manipulate a stack (Factor), send a message to an object (Io), apply a function to an array (K). We've seen two such languages a few pages back: Lisp, exemplifying the functional paradigm, and Smalltalk, exemplifying the object paradigm. In this section, we'll encounter languages that epitomize the array, logic, prototype, and concatenative (stack) paradigms.

16.2.1 APL

Kenneth Iverson's work on a mathematical notation to manipulate arrays culminated in a 1962 publication at Harvard called *A Programming Language*. By 1964 at IBM, Iverson adapted this notation into a programming language, APL.

Few languages can match APL for conciseness. APL programs are expressions in which operators are (successively) applied to arrays. APL features many dozens of operators, which

often have different meanings when used in a unary (monadic)[1] versus a binary (dyadic) context. We can get a feel for array processing in the famous expression for producing an array of all prime numbers less than or equal to R:

$$(\sim R \in R \circ . \times R) \,/\, R \leftarrow 1 \downarrow \iota R$$

For concreteness, assume R is 8. In APL, all operators associate to the right, so we first evaluate $\iota 8$ which produces `[1 2 3 4 5 6 7 8]`. Next we apply the binary drop ($\downarrow$) operator to remove one element from the front, resulting in `[2 3 4 5 6 7 8]`. We then assign ($\leftarrow$) this array to R. On the left side of the compress operator ($/$), we have $R \circ . \times R$ which builds a two-dimensional array containing the outer product, or multiplication table, of R with itself:

4	6	8	10	12	14	16
6	9	12	15	18	21	24
8	12	16	20	24	28	32
10	15	20	25	30	35	40
12	18	24	30	36	42	48
14	21	28	35	42	49	56
16	24	32	40	48	56	64

Next we apply $\in$ to R and this table, producing the array containing 1 for each element of R present in the table and 0 where not present, resulting in `[0 0 1 0 1 0 1]`.[2] Applying the $\sim$ (not) operator over each element yields `[1 1 0 1 0 1 0]`. Now we apply the compression operator which selects those elements of the right operand having a 1 in the left:

1	1	0	1	0	1	0
$\downarrow$	$\downarrow$		$\downarrow$		$\downarrow$	
2	3	4	5	6	7	8
$\downarrow$	$\downarrow$		$\downarrow$		$\downarrow$	
2	3		5		7	

The prime numbers up to 8 are thus: `[2 3 5 7]`.

APL and it descendants, such as J, Q, and K have found applications in financial domains and in scientific research.

16.2.2 Prolog

Prolog is the brainchild of a collaboration between Alain Colmerauer and Phillipe Roussel of the University of Aix-Marseille and Robert Kowalski of the University of Edinburgh in the late 1960s and early 1970s. As one of the first logic programming languages, Prolog has been extremely influential in the domains of theorem proving, expert systems, and natural language processing. It also greatly influenced the development of Erlang.

A Prolog "program" consists of a list of facts and inference rules, onto which queries can be asked. Programming is generally declarative; the engine figures out how to complete queries based on the given facts and rules. Here's a very simple Prolog "database":

[1] Don't confuse APL's use of the term *monadic* with Haskell's; the two meanings are wholly distinct.
[2] Read this result as saying that of the values 2..8. only 4, 6, and 8 were found in the table.

```
plays(kira,waterpolo).
plays(colette,diving).
plays(emily,waterpolo).
plays(olivia,diving).
plays(michelle,piano).
club(colette,rosebowl).
club(kira,rosebowl).
club(emily,rosebowl).

diver(X) :- plays(X,diving).
teammate(X,Y) :- plays(X,Z), plays(Y,Z), club(X,C), club(Y,C), X\=Y.
```

We can now run several different computations, or **queries**, against this database. For example:

- `?- plays(olivia,S)` binds the variable S to the atom `diving`.
- `?- plays(P,waterpolo)` produces multiple successive bindings for P, for each water polo player.
- `?- diver(colette)` produces `yes`, while `?- diver(alex)` produces `no`. There may be a diver named Alex, but Prolog cannot prove this fact from the given database. In Prolog, `yes` means "I can prove this" and `no` means "I cannot prove this."
- `?- teammate(olivia,colette)` produces `no`. We don't know Olivia's club.
- `?- teammate(X, Y)` finds all pairs of teammates. Our first result is $X =$ `kira`, $Y =$ `emily`, and the second $X =$ `emily`, $Y =$ `kira`.
- `?- plays(X, waterpolo), club(X, rosebowl)` binds X successively to each of the known Rose Bowl water polo players.

Prolog searches through the universe of possible facts to find query results, using backtracking internally. Occasionally, the programmer needs knowledge of this process in order to apply linguistic devices such as the **cut** (`!`) to drastically improve efficiency. Exploration of these devices is, alas, beyond the scope of this text.

16.2.3 K

K "is the executable notation at the heart of a high-performance programming platform. It is designed for analyzing massive amounts of real-time and historical data—ideal for financial modelling" [117].

K is an array-based language in the APL family (which also includes the tersely-named J and Q). Its designer, Arthur Whitney, had extensive APL experience at several companies, most notably at Morgan Stanley, where he worked on financial and trading applications and developed another APL-family language, A+. In 1993, Whitney left Morgan Stanley to form Kx Systems to commercialize the newer language, K.

Like APL, K's fundamental data type is the array, and nearly all operations apply to—or are easily extended to—arrays. Unlike APL, K programs are written in the ASCII character set. Let's take a look at the factorial function, defined as follows:

```
factorial:{*/1+!:x}
```

Working from right to left, as K does:

```
factorial 10
  = {*/1+!x} 10              -- call a function
  = */1+!10                  -- parameters are x, y, and z
  = */1+0 1 2 3 4 5 6 7 8 9  -- !n makes the array of 0..n-1
  = */1 2 3 4 5 6 7 8 9 10   -- scalar+array makes an array
  = 3628800                  -- op/array reduces (folds) the op
```

In K terminology, `+`, `*` and `*` are called **verbs**, as they apply to values, while `/` is an **adverb**, as one of its arguments is a verb. We may recognize the slash as the higher-order fold, or reduce operation, but K calls it OVER. Other adverbs include the apostrophe, called EACH, and \ called SCAN, similar to OVER but accumulates partial results. Let's run some examples in the REPL:

```
  factorial:{*/1+!x}
{*/1+!x}
  factorial' 3 8 0 4
6 40320 1 24
  +/3 8 0 4
15
  +\3 8 0 4
3 11 11 15
```

K is a proprietary language defined in [63], but an open-source version of the language, Kona, is available at https://github.com/kevinlawler/kona/. The Kona wiki has a number of example functions, a few of which we've duplicated here. Enjoy your self-study of K, and have fun puzzling out the inner workings of the following concise function definitions.

```
factorial:{*/1+!:x}
leapyear:{(+/~x!'4 100 400)!2}
pascal:{x{+':0,x,0}\1}
gcd:{:[~x;y;_f[y;x!y]]}
sort:{x@<x}
collatz:{(1<){:[x!2;1+3*x;_ x%2]}\x}
```

16.2.4 Io

Steve Dekorte's Io is another simple, "pure" language, whose programs consist entirely of objects sending messages. Dekorte created Io in 2002 "to refocus attention on expressiveness by exploring higher level dynamic programming features with greater levels of runtime flexibility and simplified programming syntax and semantics." [21]

Like JavaScript and Lua, Io is a prototype-based language. In Io, all computation, including assignment, conditionals, and loops, proceeds by sending messages. All objects are run-time accessible, as is the source code tree (yes, metaprogramming opportunities abound). The language provides actors, futures, and coroutines for concurrency. Io has much to offer and we can't do the language justice in this short section, but we encourage you to explore the examples at the book's companion website.

16.2.5 Factor

Although this book's main tour managed to visit a number of programming paradigms, we've up to now missed **concatenative programming**. Concatenative programing languages build subroutines for evaluation by composing functions.Factor is a modern concatenative language—PostScript, Joy, and Forth are three others—and like many, but not all, concatenative languages, it is also **stack-based**. It was created by Slava Pestov in 2003.

In Factor, as in other stack-based languages, each operation takes its implicit operands from a stack. Basic arithmetic has a postfix flavor:

```
9 7 * 2 + 13 neg /
```

Operands generate pushes. Operators pop values, compute, then push results:

1. Push 9, $Stack = \langle 9 \rangle$
2. Push 7, $Stack = \langle 9\ 7 \rangle$
3. Pop operands, push product, $Stack = \langle 63 \rangle$
4. Push 2, $Stack = \langle 63\ 2 \rangle$
5. Add, $Stack = \langle 65 \rangle$
6. Push 13, $Stack = \langle 65\ 13 \rangle$
7. Negate, $Stack = \langle 63\ -13 \rangle$
8. Divide, $Stack = \langle -5 \rangle$

The stack can hold code blocks, too, as in:

```
3 [ "Hello, world" print ] times
```

Developers and students of computer science will do well to study stack-based execution. There is a long history of virtual machines using stack execution, including the very popular Java Virtual Machine (JVM).

16.3 SCRIPTING

In a 1998 article [83], John Ousterhout identified a class of languages he termed **scripting languages**, remarking:

> Scripting languages assume that a collection of useful components already exist in other languages. They are intended not for writing applications from scratch but rather for combining components. For example, Tcl and Visual Basic can be used to arrange collections of user interface controls on the screen, and Unix shell scripts are used to assemble filter programs into pipelines. Scripting languages ...are rarely used for complex algorithms and data structures, which are usually provided by the components.

Today, the notion of what defines a scripting language is much more fluid, at times encompassing any language with sufficiently dynamic, high-level, expressive features. We're interested here in two classic exemplars: Bash and Perl, as well as the popular data science language, R. While each can be used for general-purpose computing, each definitely exhibit powerful and expressive features to excel in their target domains.

16.3.1 Bash

Today, operating system shells do much more than simply invoke and combine programs. Even the earliest of the Unix shells, the **Bourne Shell**, colloquially known as **sh**, provided variables, rudimentary control flow, and command substitution. Bash, created in 1989 as a replacement and extension of sh, added additional looping and conditional constructs, unbounded arrays, new forms of command substitutions, and support for in-process arithmetic. It competes with a number of other shell languages that have extended sh, including tcsh, ksh, and the default shell for macOS, zsh.

16.3.2 Perl

According to its designer, Larry Wall, Perl has two slogans: "There's more than one way to do it," (or TMTOWTDI) and "Easy things should be easy and hard things should be possible." [113] Created in 1987, Perl rose to fame in the early days of the World Wide Web due to its extraordinarily powerful text processing capabilities.

Perl continues to hold on to a fair share of popularity in the scripting world, where it competes with Python, Ruby, Tcl, and others. It is, to some old timers, the first language that comes to mind when hearing the phrase *regular expressions*. While Perl was not the first to use regular expressions, it likely popularized them like no other. Its extensive regex support inspired the popular PCRE (Perl Compatible Regular Expressions) library.

Perl has a rather unique culture in the programming language space. The language is very flexible and supports a great many programming styles. Wall's training as a linguist led to many of the language's highly expressive—and sometimes quirky—features, including various quoting mechanisms, a flip-flop operator, a spaceship operator, the diamond, and the tilde-tilde. Wall has even come up with a self-deprecating backronym for the language: *Pathologically Eclectic Rubbish Lister*.

16.3.3 R

R, designed by Ross Ihaka and Robert Gentleman (hence the name, R), is an implementation of the S programming language inspired by the lexical scoping semantics of Scheme.

R excels in statistical modeling applications thanks to its extensive set of core (and user-contributed) packages and direct support for matrix arithmetic. For a taste of the language, this small script hardcodes the number of bedrooms and price of seven homes, and displays a scatter plot with a linear regression analysis as well as a histogram of prices.

```
data <- data.frame(
  x = c(2, 2, 2.5, 3, 4, 5, 5, 5),
  y = c(90, 83, 125, 150, 144, 200, 250, 300)
)

model = lm(y ~ x, data = data)
plot(data$x, data$y, main = "Prices", xlab = "Rooms", ylab = "Price")
abline(model, col = "red")
hist(data$y, main = "Prices", col = "green")
```

16.4 THE ML FAMILY

While working on the theorem proving system LCF (short for Logic of Computable Functions), Robin Milner and colleagues in the early 1970s created a language for writing scripts to create proofs. Since LCF was itself a language for representing formulas in an underlying logic, the theorem-proving language was called the *Meta Language*, or **ML** for short. The original ML language soon evolved into a general-purpose programming language. Like Lisp, it featured first-class functions and pattern matching, and was expression-oriented and garbage-collected. Unlike Lisp, ML was statically typed, and was the first language to implement the Hindley-Milner type system and its associated type inference mechanism.

A number of ML dialects soon appeared, with the most popular family branches being led by Standard ML and OCaml. Several closely-related languages such as F#, PureScript, and Idris include the characteristic features of ML mentioned above that we consider them, at least for this book, to be part of the (extended) ML family.

16.4.1 Standard ML

In April of 1983, Milner authored *A Proposal for Standard ML* to advocate for a language that would consolidate the strengths of the multiple ML dialects created to that time. The new language appeared shortly after, with a highly sophisticated module system in which (1) **structures** define (parameterized) collections of types and values, (2) **signatures** provide interfaces into structures, often providing a restricted view of the structure to clients, and (3) **functors** can compute new structures, giving an interesting computational view into generic instantiation, among its other capabilities. A complete example with structures, signatures, and functors is available at the companion website.

16.4.2 OCaml

While Milner's efforts to consolidate the disparate ML dialects into a single standard had some success, one family branch—that of INRIA's CAML—did not join the party. By 1996, CAML had evolved into OCaml, also known as Objective Caml.

OCaml and Standard ML are very similar languages, inheriting all of the pattern-matching, type inference, and other powerful features of their ancestor, but differ greatly in syntax. OCaml does have better support for what we today call object orientation, and has turned out to be better known than its "standard" sibling. It has notably become a convenient language for authoring compilers—both Facebook's Hack and early versions of Rust were written in OCaml. Modern languages such as Elm, F#, Scala, Rust, and Grain have listed OCaml as one of their primary influences. The language is also known for high performance and safety, and has found uses in financial trading domains.

16.4.3 F#

Microsoft's F# first appeared in 2005 and was developed as an open-source, cross-platform CLI language, intended as a modern ML for the .NET world. Microsoft bills it as a "functional-first" language. While the similarities to ML languages such as OCaml and Haskell are unmistakable, F# contains an impressive list of features and support for imperative programming, asynchronous workflows, agents, object orientation, and parallelism, and is used in machine learning, business intelligence, analytics, and gaming.

16.4.4 PureScript

Phil Freeman began work on PureScript in 2013 when "I found myself using functional programming techniques that I had picked up in languages like Haskell, but wanted a more principled environment in which to apply them. Solutions at the time included various attempts to compile Haskell to JavaScript...but I was interested to see how successful I could be by approaching the problem from the other side—attempting to keep the semantics of JavaScript, while enjoying the syntax and type system of a language like Haskell."

PureScript is syntactically similar to Haskell, and shares several advanced features with its influencer, namely Hindley-Milner typing, algebraic datatypes, pattern matching, higher-kinded types, and typeclasses, while adding row polymorphism and extensible records. It differs form Haskell primarily in its emphasis on iteroperation with JavaScript (and other transpilation targets, including Go), its preference for strict (not lazy) evaluation, and its use of an `Effect` monad, rather than Haskell's `IO`.

16.4.5 Idris

Idris debuted in 2009 as a pure functional language featuring **dependent types**. Dependent types allow many aspects of a program's behavior to be specified within the type system itself. Common examples of dependent types include "the type of all vectors of length n," "the type of all pairs of integers in which the second element is greater than the first," and "the type of all functions returning the value seven." With computations performed within the type checker, at compile-time, the programmer can set guarantees that a program, when executed, will meet certain specifications. In fact, Idris borrows a number of features from interactive proof assistants such as Coq and Agda. While Idris can be used for theorem proving, its development has focused on general-purpose computation.

Syntactically, Idris looks a great deal like Haskell, with some superficial differences beyond those forms required to express dependent typing. Semantically, Idris is a pure functional language employing **algebraic effects** (in addition to monads) for managing side effects (networking, file I/O, exceptions, user interaction). Idris's creator, Edwin Brady, describes the theory and implementation of algebraic effects, and the means by which they solve various shortcomings of Haskell's monadic approach, in [12].

16.5 SYSTEMS LANGUAGES

It's often helpful to distinguish **application software** and **system software**. The former solves problems of interest to humans and is device and platform-independent; the latter is concerned with managing the resources of a computer system (memory, files, processes) and may be device-dependent. An **application programming language** provides rich data modeling constructs and assumes the existence of a large runtime system to perform memory management without programmer intervention. A **systems language** compiles "to the metal" with little or no runtime. C (Chapter 13) and Rust (Chapter 15), along with C++ and Zig, do without garbage collectors, while Go (Chapter 12), D, and Nim allow for garbage collectors while still providing low-level features suitable for systems programming. We've seen how the quintessential systems language C is unsafe-by-design systems language; each of the alternatives add, in varying degrees, features to make system programming safer. Let's look at four of these.

16.5.1 C++

In the 1970s, Bjarne Stroustrup implemented a simulator to study communication between programs in a distributed environment. The first version, written in Simula, was a "pleasure to write" but performance problems (80% of the run time was in the garbage collector) nearly doomed the project, so Stroustrup rewrote the simulator in BCPL (a predecessor of C). However, "the experience of coding and debugging the simulator in BCPL was horrible." [106] This experience led him to design a new language, a superset of C with Simula's classes, but without garbage collection, run-time type checking, and the impossibly low-level near-typelessness of BCPL. The language went by the name C with Classes and evolved, by 1983, into C++. With this new language, object orientation went mainstream.

As a superset of C, C++ necessarily permits manual memory management, but encourages memory-safe programming patterns such as **smart pointers** and the use of class constructors to allocate memory and destructors to deallocate, hiding management details from users of the class. C++ encourages all user-defined data types to act as if they were copied, but provides mechanisms to avoid copies via moves. Pointers and aliases (confusingly termed references in C++) are available when explicit sharing is desired.

C++ is a venerable, long-lived, and massively popular language used in enterprise applications, games, and performance-critical environments. It has evolved well: the so-called *Classic C++* has grown into *Modern C++*, a kind of language rebirth that began with C++ 11, and has continued through C++ 14, C++ 17, and C++ 20. Many of the programming patterns in Modern C++ replace the "old ways" of doing things. C++ programmers now enjoy extensive type inference, range-based `for`-loops, atomics, and anonymous functions (even supporting generics). The combination of expressive constructs and system-level memory operations has kept the language relevant and widely used.

16.5.2 D

Walter Bright at Digital Mars started the D language in 2001 at Digital Mars, as an improved version of C++. While keeping many of the low-level features of C++ (including the ability to inline assembly code) and remaining statically typed, D adds many expressive features common in dynamic languages, including array slices, function literals and closures, map and reduce. D even has automatic garbage collection, but under programmer control! The language provides several mechanisms to ensure memory safety around certain object lifetimes, though as some manual memory management is allowed, programmers have the ability to decorate areas of safe and unsafe code.

16.5.3 Nim

Like D, Nim is a systems programming language with plenty of modern features and tunable garbage collectors. It avoids many unsafe features of C such as pointer arithmetic. Created by Andreas Rumpf and first released in 2008, Nim 2.0 arrived in 2023 with reference counting as the default memory management strategy. Nim has a pleasant syntax (inspired by Python), an extensive macro system, a rich standard library, and mechanisms to interoperate with C and JavaScript.

16.5.4 Zig

Zig, created by Andrew Kelley and first released in 2016, is a pragmatic alternative to C that is relatively easy to learn due to its consistent and readable syntax. It has no garbage collector, but improves upon C's wild and unconstrained `malloc` and `free` calls by allowing **allocators** to be passed to functions that require dynamic memory. Error handling and null safety are notable and rather similar to Rust's mechanisms.

16.6 CONCURRENCY

Most modern languages feature concurrency support, but experienced developers working on mission-critical applications require a deep understanding of the fundamental principles of concurrent and distributed systems. This knowledge comes not only from day-to-day practice but also from a study of early seminal work such as Hoare's CSP and Milner's CCS, and the early experimental and research languages occam (featuring named channels, as adopted by Go), SR, and Linda (sporting a unique means of process coordination through a "tuple space").

Naturally, research in concurrent and distributed systems remains extremely active. We've spent time in previous chapters on a few principles of concurrency in our tours of Java, Clojure, Go, and Rust. A few languages have been created with concurrent and distributed in mind; let's explore a few.

16.6.1 Erlang

Organizations and personnel responsible for service availability like to talk about nines. "Five nines" means up and running 99.999% of the time, down only five minutes per year. Seven nines means down only three seconds per year (or two minutes every forty years). Can a *language* help its users (developers) achieve high availability? Erlang can, and does. It's almost entirely functional. Processes are lightweight and share nothing, communicating only via messages. Processes are distributed over machines transparently.

Joe Armstrong and others at Ericsson developed Erlang in the 1980s to implement highly reliable, long-lived applications in telephony to handle a large number of concurrent activities, meet real-time constraints, scale across large distributed networks, and are able to be modified while running. It lives on today in modern applications like CouchDB, RabbitMQ, WhatsApp, and Whisper. Like Java, Erlang arrived with its own virtual machine. And just as hundreds of languages have been targeted to Java's JVM, a number of up-and-coming languages (including Elixir, Joxa, and LFE) are designed to run on Erlang's virtual machine.

16.6.2 Elixir

Elixir was created by José Valim in 2012 as a modern Erlang, running on the Erlang virtual machine, with all of Erlang's process capabilities, but with a syntax influenced by Ruby. It adds to Erlang powerful metaprogramming facilities, with hygienic macros (operating on abstract syntax as in Julia and Clojure), protocols, reducers, and the convenient `|>` operator. Some of Erlang's more unusual syntactic forms and its representations of strings as lists of integers have been modernized. The Ruby-style syntax makes the authoring of domain-specific languages within Elxir quite pleasant, and the language has found use in voice, messaging, and communications applications.

16.6.3 ParaSail

Designed by Tucker Taft and first released in 2009, ParaSail, or *Parallel Specification and Implementation Language*, is a "parallel programming language designed to support the development of inherently safe and secure, highly parallel applications that can be mapped to multicore, manycore, heterogeneous, or distributed architectures." [108] The language has a familiar syntax and traditional object-oriented features such as classes, interfaces, and parameterized modules. Safety features are designed into the language itself and *enforced by the compiler*; there are no pointers, global variables, parameter aliasing, exceptions, or explicit locking.

16.6.4 Chapel

Chapel, developed at Cray Inc. and originating as part of DARPA's High Productivity Computing Systems (HPCS) program, aims to improve the programmability of parallel computers. It is strongly influenced by classic languages such as (High-Performance) Fortran, C, and C++ and their associated libraries, but has the look and feel of more modern, expressive languages such as Python. An important design goal has been to make the language expressive enough to formulate all parallel tasks with out the need to drop into a library. Architecturally, writes Brad Chamberlain, Chapel's principal engineer, "Chapel supports distinct concepts for describing parallelism ('These things should run concurrently') from locality ('This should be placed here; that should be placed over there'). This is in sharp contrast to conventional approaches that either conflate the two concepts or ignore locality altogether." [15]

16.7 MULTIPARADIGM LANGUAGES

An important domain targeted by some languages is the so-called **enterprise**, a term roughly meaning "a large collection of applications, custom and third-party libraries, and scripts for managing the mission-critical operations of a business or organization." In the business world, enterprise software manages data stores, processes bulk data from partner companies (often called *feeds*), services customers' browsers and mobile applications, and generates massive amounts of usage data, or analytics. In other domains, enterprise software may refer to the millions of lines of code running sophisticated aircraft or spaceships. Regardless of the domain, building systems with hundreds of thousands, or millions of lines of code requires language support for writing correct, reliable, robust, scalable, efficient code that is easy to maintain and extend.

16.7.1 CLU

As the first language to combine abstract data types and encapsulation, CLU was authored by a group led by Barbara Liskov beginning in 1973. Although not widely implemented, CLU was an enormously influential language. Its pioneering approach to information hiding has been adapted by hundreds of subsequent languages. It provided "linguistic support for data abstraction [and] was the first implemented language to do so. In addition, it contain[ed] a number of other interesting and influential features, including an exception handling mechanism, iterators, and parameterized types." [65] Its construct for modularity, the **cluster** (which gives the language its name), extended Simula's `class` by encapsulating, or hiding, all of the objects and functions not explicitly exported. The object and type systems were essentially copied to a tee by Python: values of all variables are references to

entities, and assignment and parameter passing therefore copies references. Liskov herself coined the term **pass-by-sharing** for this scenario, a term popularized by Python. In addition, Python's universal supertype, `object`, is exactly CLU's `Any`.

CLU's design and implementation came from years of research on programming abstractions and structuring mechanisms, and was guided by a number of principles including "minimality, simplicity, expressive power, uniformity, safety, and performance [65, p. 8], goals adopted by subsequent languages. Liskov writes:

> CLU was geared toward developing production code. It was intended to be a tool for "programming in the large," for building big systems (e.g., several hundred thousand lines) that require many programmers to work on them. As the work on CLU went on, I developed a programming methodology for such systems. CLU favors program readability and understandability over ease of writing, since we believed that these were more important for our intended users [65, p. 6].

16.7.2 Ada

Ada, named after Ada Lovelace, often credited as the first computer programmer, [94] was Jean Ichbiah's and CII Honeywell Bull's winning entry in a late 1970s competition sponsored by the United States Department of Defense to create a standard language to supersede the then hundreds of programming languages being used in the Department. It is still used to day in applications including air-traffic control and satellites, and defense systems

First appearing in 1983, Ada became the first internationally standardized object-oriented language [55] [10] Ada is a large language with several design goals, including:

- *Support for megaprogramming.* Ada was designed to support systems with millions of lines of code through various modular constructs split into interfaces and implementations, and can perform type-checking across separately compiled modules.
- *Support for real-time, concurrent, embedded systems.* Ada's representation clauses, tasks, and protected objects allow for hardware support with high-level constructs.
- *A laser focus on safety.* Ada prohibits virtually all type coercions, provides many constructs that hide all nonexported entities, and requires that generic types be explicitly declared. Ada has a reputation as a verbose language due to its favoring of explicitness over implicitness nearly everywhere.

Two of Ada's best-known features are (1) the **rendezvous**, a synchronization mechanism on which running tasks exchange information in a controlled fashion where the sender and receiver wait for each other to be ready to communicate, and data is transferred safely without race conditions; and (2) the **protected object**, a data-oriented mechanism for thread safety. Examples of each are included in the companion website.

Though not widely used outside its niche in safety-critical environments, Ada continues to evolve today. It has both influenced and has been influenced by a number of languages for whom safety is a critical motivating design factor.

16.7.3 C#

C# is the flagship enterprise language for the .NET framework [72], a popular software platform created by Microsoft, and the most widely used language on the platform. Although early versions of C# were notably similar to Java, C# has, through its evolution, kept

ahead of Java by supporting features that Java only later adopted, has yet to adopt, or may never adopt. These include nullable types, reified generic types,[3] partial methods, properties, functional programming support (e.g., lambda expressions), extension methods, anonymous types, and string interpolation.

C# sports many features useful in an enterprise language, including safety features such as static type checking, bounds checking, isolation of unsafe memory operations, and a prohibition against certain forms of error-prone shadowing. It prohibits compilation of code in which local variables are not assigned, but guarantees initialization of certain other variables, including instance variables. In addition, programmers find many conveniences such as type inference, properties (sugaring getters and setters, as in Swift), namespaces, list comprehensions, delegates, operator overloading, and metaprogramming capabilities. Fluent styles of programming are quite common. The following snippet of code features methods from the `System.Linq` ("(l)anguage-(in)tegrated (q)uery.") namespace:

```
lines
.SelectMany(line => wordRegex.Matches(line).Cast<Match>())
.Select(x => x.Value.ToLower())
.GroupBy(x => x)
.OrderBy(x => x.Key)
.ToList()
.ForEach(p => Console.WriteLine("{0} {1}", p.Key, p.Count()));
```

16.7.4 Scala

Scala (the *Sca*lable *la*nguage) was created by Martin Odersky at École Polytechnique Fédérale de Lausanne in the early 2000s as a functional, object-oriented language for the Java platform, and as such can be used anywhere Java is popular—infrastructure, APIs, etc. It added functional programming features to the JVM world years before lambdas were introduced to Java. It also was well ahead of Java in implementing type inference, string interpolation, class declarations with simple fields in a class header, and the ability to write scripts without an enclosing class. Though Scala is interoperable with Java and its threaded concurrency model, its preference for immutability and safety drove it to adopt the actor model as its main concurrency feature, though like Clojure, it also features software transactional memory.

16.7.5 Mojo

Mojo was created by Chris Lattner (of Swift and LLVM fame) and his team at Modular, Inc., primarily for systems programming and the creation of AI applications. It was made available to the world via browser-hosted notebooks in May of 2023 and to Linux-based machines in September of 2023. Designed as a superset of Python (to lessen the leaning curve), benchmarks exist that show certain Mojo applications running 35,000 times faster than their Python counterparts [18]. The speedup is due to the use of the LLVM compiler toolchain, the MILR (Multi-level Intermediate Representation) compiler infrastructure, and the compiler's ability to take direct advantage of parallel hardware. The complier is even able to autotune applications for even better performance.

[3] A reified generic maintains type information about the instantiated type at run time. Contrast this approach with Java's *type erasure*, which removes this knowledge at run time.

16.8 THE WEB

Given the ubiquity of the World Wide Web, it's not at all surprising that programming languages specifically targeted to the web as a platform would arise. The web is a **client-server** system, where multiple clients interact asynchronously with a stateless server using a protocol with a small, limited number of verbs, namely GET, POST, PUT, PATCH, DELETE, OPTIONS, HEAD, and TRACE. On the client side, web browsers have engines for directly running JavaScript or WebAssembly (WASM)—a binary format that just about any language can be compiled to for browser execution.

While just about any language can be—and has been—used on the server side of the web (e.g., Python, Ruby, Java, and many others come to mind), one, PHP, owes its very existence to its creator's desire for a server-side web application scripting program. On the client-side, JavaScript and TypeScript still live at the top of the most-popular lists, there's competition: hundreds of languages can be translated directly into JavaScript—the big list is at [9]. And with WASM gaining popularity, a fair number of browser-based games have been written in Rust and C++ without any JavaScript at all. We'll restrict our attention to those client-side languages that were designed specifically for web applications.

16.8.1 PHP

PHP is a server-side scripting language developed by Rasmus Lerdorf, who originally built it to maintain his **p**ersonal **h**ome **p**age. PHP stands for **P**HP **H**ypertext **P**rocessor, and is today an expressive, general-purpose programming language with plenty of features, including closures, null-coalescing, exception handling, generators, and variadic functions.

PHP does, however, remain dogged by some early design choices, including weak typing, inconsistent spelling (e.g., `gettype` vs. `get_class`); the left-associative conditional (`TRUE ? 1 : FALSE ? 2 : 3` is 2), and strangely named methods (e.g., `DateTimeImmutable::modify()`.) in its somewhat organic evolution. Ample good-natured criticisms of these design choices are documented in [25] and [114], among others. The language has, over the years, shed some of the early problematic features, but even with its remaining issues is able to stay high on the list of the world's most popular languages, powering a large number of the world's web servers.

16.8.2 Dart

Dart was created by Google to create scalable web applications, and is used today in products such as AdSense and GoogleFiber. Its flavor of object orientation, with interfaces, abstract classes, and generics are familiar to fans of Java and C#. While supporting client-side web applications and interoperability with JavaScript, Dart can manipulate files, sockets, and other I/O channels not available to typical browser-based runtimes.

Dart enables concurrency through **isolates**, a multithreaded message-passing system that compiles to the HTML5 Web Worker API [115] (or to the browser's event loop if web workers are unavailable). The language also defines hardware-friendly datatypes that pack multiple numeric values into a single unit for parallel execution, such as `Float32x4`, `Int32x4`, `Float32x4List` and `Int32x4List`, making Dart suitable for use with high-performance graphics and other computationally intensive tasks.

16.8.3 Elm

Evan Czaplicki created Elm in 2011 as a language for building browser-based applications in a functional programming style. Like TypeScript, Elm applications transpile to JavaScript. But while TypeScript is a superset of JavaScript, Elm is syntactically similar to Haskell. While it has the expected ML features of Hindley-Milner type inference, pattern matching, persistent data structures, and more, it replaces JavaScript's promise-based style with a subscription-based approach to interactivity.

16.8.4 Grain

Grain was created by Philip Blair and Oscar Spencer in 2017 as a statically-typed, high-level language targeted directly to the Web Assembly platform. Grain's built-in libraries provide direct access to the WebAssembly System Interface (WASI) for ease of building WebAssembly applications with multiple languages. Grain borrows several syntactic features from Rust (among them enums, the `mut` keyword), and borrows the type checker from OCaml, allowing programs be both short and quite readable.

16.9 QUANTUM COMPUTING LANGUAGES

When writing code for a quantum computer, one can either use a general-purpose programming language with libraries, frameworks, and software development kits to compose quantum algorithms, or use a language specifically designed for making quantum algorithms easier to express. Popular Python libraries include Google's Cirq or Xanadu's Strawberry Fields and Pennylane. Quantum languages include Q#, from Microsoft Research, and Quipper, a quantum circuit description language with powerful operators for assembling and manipulating circuits. Quipper features quantum data types, circuit transformers, and a number of quantum functions prepackaged into libraries. It runs as a domain-specific language embedded within Haskell. An extensive list of quantum libraries and languages can be found at [123]

16.10 ESOTERIC LANGUAGES

No tour of programming languages should omit the ever-growing family of **esoteric languages**, with members such as Ook!, Malbolge, Befunge, LOLCODE, Shakespeare, Chef, Chicken, Whitespace, and GOTO++. You won't see any of these powering any real-world applications; instead, their designers work outside the box to produce artistic, crazy, funny, themed, weird, or insightful creations that may push the boundaries of minimalistic computing models, look at computing in a whole new way, or simply amuse.

16.10.1 Brainf**k

Brainf**k is one of the better-known of the esoteric languages and one of the earliest. Urban Müller's 1993 creation grew out of an effort to design a Turing-complete[4] programming language implementable with the smallest possible complier (296 bytes at the time, with smaller compilers created since).

[4]Formally, as powerful as a Turing Machine; informally, capable of doing anything a Python program can do assuming infinite memory.

A Brainf**k program is a sequence of one-character commands operating on a single infinite array of bytes (indexed from zero) and two streams of bytes—standard input and standard output. The *current* memory cell is initially the leftmost. The eight commands are:

`,`	Read the byte from stdin into the current cell
`.`	Write the byte in the current cell to stdout
`<`	Move the current cell pointer left
`>`	Move the current cell pointer right
`+`	Increment the byte in the current cell
`-`	Decrement the byte in the current cell
`[`	If the byte in the current cell is zero, jump ahead to just past the matching `]` (Think of this as the beginning of a while loop: if the condition is false, you will skip the entire loop)
`]`	If the byte in the current cell is nonzero, jump backwards to just beyond the matching `[` (Think of this as the end of a while loop: if the condition is true, repeat the loop body again)

The following program:

```
+++++++[>+++++[>++>+++<<-]<-]>>++.>.
```

prints the string `Hi` to standard output. As all cells have an initial value of zero, the program first increases the first cell to 7, then the second to 5, then the next two cells to 2 and 3, with the fourth cell as our current cell:

```
7   5   2   3
            ^
```

We're inside a doubly-nested "loop" here. Our next instruction says to move backward two cells and decrement the value. If not yet zero, we'll continue our inner loop of incrementing the third and fourth cells by 2 and 3 respectively. After this is done five times, our memory becomes:

```
7   0   10   15
    ^
```

We're back in an "outer loop" now, and we're supposed to go back one cell to the left and decrement...we're being told to do what we just did seven times. This takes us to the following state:

```
0   0   70   105
^
```

We now need to move two cells to the right, bump the value by 2 to 72, the code point for Latin Capital Letter H and print it with the `.` command. Moving one cell to the right we find, happily, 105, the code point of Latin Small Letter I, which we print. And we're done.

You might want to now work through the classic Hello world program, as found on [118]:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>
---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Finally, do consider writing your own Brainf**k compiler. If you get stuck, consult the direct mappings from Brainf**k commands to fragments of C code online at [118].

16.10.2 Befunge

While Brainf**k was designed to make it easy to write compilers, and in fact small compilers, Chris Pressey designed Befunge as a challenge to would-be compiler writers. Befunge programs are written in a two-dimensional grid of instructions. Some instructions manipulate the **data stack** in the usual way, some do input and output (to/from the stack), and some instructions—`<`, `^`, `>`, and `v` change the direction in which the program is read. Even better, or worse, is the `p` instruction: it pops from the stack three values y, x, and v, then replaces the character at position (x, y) *in the code grid* with v. So the grid contains both instructions and data, allowing you to change code on the fly.

Here's a factorial function in Befunge (can you see the loop?):

```
&>:1-:v v *_$.@
 ^    _$>\:^
```

16.10.3 LOLCODE

A year after the massive popularization of the internet meme LOLcats, Adam Lindsay developed LOLCODE, a procedural language that adapted LOLcats' "kitty pidgin" speech to its own programming vocabulary. LOLCODE features all uppercase keywords such as `HAI`, `KTHXBYE`, and `O RLY?`. Underneath its amusing syntax, LOLCODE is actually a regular, imperative (though small) programming language, with loops, conditionals, and functions. Here's our recurring example script for computing Pythagorean triples:

```
HAI 1.2
  IM IN YR FIRST UPPIN YR Z TIL BOTH SAEM Z AN 41
    IM IN YR SECOND UPPIN YR Y TIL BOTH SAEM Y AN Z
      IM IN YR THIRD UPPIN YR X TIL BOTH SAEM X AN Y
        I HAS A W ITZ SUM OF PRODUKT OF X AN X AN PRODUKT OF Y AN Y
        BOTH SAEM W AN PRODUKT OF Z AN Z, O RLY?
          YA RLY, VISIBLE X ", " Y ", " Z
        OIC
      IM OUTTA YR THIRD
    IM OUTTA YR SECOND
  IM OUTTA YR FIRST
KTHXBYE
```

Cats don't write very well, so you won't see much punctuation in LOLCODE; we've only used the comma, which does nothing more than abbreviate a newline. And fixed-arity operators don't need parentheses (try proving this!). A few symbols do appear within YARN literals (YARNs are strings, get it?): `:)` represents the newline, `:>` is the tab character, and so on. The theme continues with well-chosen token names for breaks (`GTFO`), switches (`WTF?`), cases within switches (`OMG`), returns (`FOUND YR`), and the else branch of an `O RLY?` statement (`NO WAI`).

16.10.4 CJam

Code Golfing is a recreational pastime where participants attempt to solve programming challenges with the shortest possible source code. While one can golf in any language, dozens of languages [26] have been created expressly for golfing. Such languages tend to feature powerful one-character functions. CJam, created by aditsu in 2014, is a stack-based golfing language. The stack can contain numbers, arrays, strings, and even blocks of code. The stack is implicitly printed at the end of the program, so:

```
13 8 3 - + 2 3 * 2 / #
```

prints the evaluation of $(13+(8-3))^{(2*3)\div 2}$, namely 5832 (note that `#` is the exponentiation operator).

The language boasts nearly 100 predefined operators, many of which have multiple meanings depending on the type of the operands, and 26 variables (`A-Z`), all with initial values. Let's explore with a small script that prints the first twelve Fibonacci numbers to standard output, one per line:

```
0X{_2$+}A*]N*
```

The script begins by pushing 0 then 1 to the stack (X is pre-initialized to 1). Then a block of code is executed 10 times (`A*`). The block pushes (duplicates) the top of the stack (with `_`), then pushes the value two slots below the top (`2$`) and adds them. After 10 iterations, the entire stack is turned into an array with `]`. Then the array elements are joined (via the overloaded `*`) with a newline, the initial value of the variable N.

16.10.5 Pyth

Pyth, created by Izzy Grosof in 2014, is another golfing language, but rather than employing an implicit stack, Pyth programs translate rather directly into Python programs. All operators are prefix and have a fixed arity, rendering parentheses unnecessary. This means that since the subtraction operator is the hyphen character `-`, unary negation has to be something else; in fact, it's the underscore (`_`).

Like CJam, Pyth operators are overloaded; for example `+` will add numbers but concatenate strings and lists. The underscore `_` not only negates numbers but reverses sequences and inverts (swaps keys and values) of dictionaries. Control flow constructs use capital letters: F begins a for-loop, V is a for-loop over a range, and W a while loop. Many of Python's operators appear as single-character Pyth functions, e.g., m for map, f for filter, and r for reduce. Programs can be written as a single line with statements separated with semicolons. Opening quotes, parentheses, and brackets are automatically closed at the end of statement, since in code golfing, every character counts.

Here's the first-twelve-Fibonacci script from our section on CJam above, written in Pyth:

```
A(Z1;VhhTGA(H+GH
```

We begin with the tuple assignment operator A assigning to the implicit target (G,H), their initial values $(0,1)$. We then iterate (with V), 12 times (hhT, since T is 10 and h is the successor function) the following: evaluate G (for implicit printing), then assign $(G,H) \leftarrow (H, G+H)$.

Do visit the Pyth tutorial and documentation from [38], as well as the book's companion website for a collection of additional examples.

16.11 ASSEMBLY LANGUAGES

An **assembly language** describes a machine's execution, often with commands that map one-to-one to **machine instructions**. Even if you use only high-level languages, something about machines—machines of the past, present, and future—may be relevant to you. Uncle Bob tells us why: [69]

> Did you know, for example, that the `++` operator in C, C++, Java, and C# derived from the fact that there were special registers in early computers that automatically incremented their contents every time you accessed them? The compiler tried to make use of these auto-increment registers whenever you used a `++` operator.
> ...our languages are projections of the hardware. Some languages are more abstract projections than others; but the most popular languages, C, Pascal, C++, Delphi, Java, and C# are very closely related to the hardware. Even languages like Smalltalk, Python, and Ruby, while more abstract than the others, still show their hardware roots.

Machines can be quite interesting. While the average programmer with little exposure to assembly language may dread the number of instructions required to implement a `for`-loop, the fact is machines often contain relatively powerful instructions, combining comparisons and jumps, locking, exchanging regions of memory, and operating on dozens of data items in a single clock pulse.

We'll cover two short examples in the NASM assembly language, which targets the x86-64 architecture [54]. The first is a function computing the maximum of three integers. We have here only a function, not a complete program. After assembling and linking the code, the function can be called from any language that can be compiled to the same executable file format that the assembler produces.

```
; A 64-bit function that returns the maximum value of its
; three 64-bit integer arguments. The function has signature:
;
;   int64_t maxofthree(int64_t x, int64_t y, int64_t z)
;
; Note that the parameters have already been passed in rdi,
; rsi, and rdx.  We just have to return the value in rax.

        global  _maxofthree
        section .text
_maxofthree:
        mov     rax, rdi        ; result (rax) initially holds x
        cmp     rax, rsi        ; is x less than y?
        cmovl   rax, rsi        ; if so, set result to y
        cmp     rax, rdx        ; is max(x,y) less than z?
        cmovl   rax, rdx        ; if so, set result to z
        ret                     ; the max will be in rax
```

Here we see that functions accepting integers accept arguments in various registers, the first three of which are `rdi`, `rsi`, and `rdx`. The instructions for performing comparisons and conditional moves are explained in the comments.

And now let's look at one last example. The x86 architecture features hardware registers that can store, and operate on, multiple units of data at once, making **SIMD** (single-instruction, multiple-data) processing possible. The so-called XMM registers are 128-bits wide; the YMM registers are 256-bits wide; ZMM registers are 512-bits wide. Here's a very brief illustration: we'll add two 4-element arrays of floating-point numbers using the XMM registers:

```
; void add_four_floats(float x[4], float y[4])
; x[i] += y[i] for i in range(0..4)

        global   _add_four_floats
        section  .text

_add_four_floats:
        movdqa   xmm0, [rdi]     ; all four values of x
        movdqa   xmm1, [rsi]     ; all four values of y
        addps    xmm0, xmm1      ; do all four sums in one shot
        movdqa   [rdi], xmm0
        ret
```

A good compiler for a high-level language should be able to recognize when SIMD instructions can be used. And you may often see software libraries with functions that map directly to these instructions. But sometimes, you may wish to break out an assembler and experience the feeling of writing assembly language directly, feeling at one with the machine. Fortunately, several high-level languages *do* give explicit support for SIMD operations, including Dart and Mojo.

LANGUAGE TOUR WRAPUP

The question that opened the preface of this book, "Why would you want more than machine language?" was asked by none other than John von Neumann. History has shown that high-level languages matter. But somehow, it does seem rather fitting to finish our tour looking at...of all things, an assembly language. As long as our programs run on hardware, we'll need to be aware of the connections between our high-level languages and the machine and assembly languages that make the hardware run.

Von Neumann's name now labels one of the most prolific electronic computer architectures of the last century—one in which programs and data share memory, a program counter ticks away running sequential processes on each core, an arithmetic-logic unit (ALU) processes instructions, and a bus shuttles data to and from memory. But hardware engineers have been building alternative architectures for quite some time! Dataflow and reduction machines have been around for a while, and to some extent, functional and array programming languages really do show very little connection to the von Neumann architecture at all. Today we see research in a number of brain-inspired computer architectures, such as IBM's TrueNorth [71], and other neuromorphic [77] systems. Will we see inklings of these architectures showing up in future programming languages? Or will new architectures, no

matter how novel, be programmed in the languages of today, employing little more than new programming paradigms?

Speaking of hardware, it has been noted that since the 1950s, computer hardware has improved *several orders of magnitude* in speed, storage capacity, size, and weight [69]. Where are the orders of magnitude improvements in software? How much better are today's languages from the early versions of Fortran and Lisp? Are they anywhere near orders of magnitude better? What kind of revolutionary thinking would it take to design and implement a language that is?

We hope you've enjoyed the tour.

CHAPTER 17

Concepts

Hello again! If you've made it this far, you've been introduced to the study of programming languages through chapter-length overviews of fourteen languages and brief glances at a few dozen more. The intent was not to provide a comprehensive tutorial for any of the languages, but rather to use the overviews as a vehicle to introduce important *concepts* of programming language theory, design, and pragmatics.

Now it's time to revisit all that we've learned from another angle, this time with a focus on the concepts themselves. We will be bringing together and organizing the various linguistic elements around three major themes: (1) the **classification** of human-friendly units of information by structure and behavior into types; (2) the various linguistic **structures** that build a program up from entities (literals, variables, expressions, statements, functions, modules, and more), and (3) **execution**, the ways in which computations themselves progress over time and how the various computational units communicate and coordinate with each other.

While learning individual languages is certainly important, a study of language-independent concepts is a necessary step in advancing in the field. We need theory! One dictionary defines a theory as "the branch of a science or art consisting of its explanatory statements, accepted principles, and methods of analysis." [3] That is, a theory is an organized body of knowledge with explanatory and predictive powers. With a good theory, we have the vocabulary to communicate with others, reason about a discipline, make predictions, and generate new knowledge.

Armed with programming experience and proficiency in over a dozen programming languages, we can now organize our knowledge of language principles into a framework that will guide further study. Although this final chapter serves as a *summary* of concepts introduced during our language tour, we will slip in a few concepts *not* covered earlier, but that fit into the overall organization in a nice way. Take the opportunity to explore these popup concepts on your own.

Finally, because a Socratic approach can be fun at times, we will present the concepts in this chapter not with an outline, but with questions. The goal is to not only summarize, but to bring ideas close together to motivate ways of thinking about programming languages. We encourage the reader to not only look for answers to the questions posed here, but to also generate illustrative examples in their favorite languages.

17.1 CLASSIFICATION

Grady Booch has written:

> Scientists who work in material domains—physics, chemistry, astronomy, biology, geology, meteorology—study the cosmos and all that is in it, reflect on the patterns they observe, then try to reduce their understandings to the simplest principles.
>
> I am a computer scientist; I approach the cosmos though a very different path: I take the simplest of principles—information theory, logic, design—and from them create new worlds that are bound only by my imagination.

We can build new worlds, or descriptions of them (languages), from the building blocks of information: **bits**. We can use specific arrangements of bits to describe anything: numbers (bounded integers, unbounded integers, floating point numbers of different sizes, rationals, arbitrary-precision reals, complex numbers and more), booleans, and text. We call these human-friendly chunks of information **values** and classify them into **types**. Different languages have different type systems; we've seen many in previous chapters! To further our study and unify some of these concepts, we may ask, for a given language:

1. Is there exactly one numeric type or multiple numeric types? Are integer types divided across signed and unsigned types? Fixed size and unbounded? Modular and saturated? Are there separate types for 8, 16, 32, 64, and 128-bit fixed-size integers? Are floating-point types so divided? Is there an unbounded floating-point type?
2. Are rational, decimal, or complex types defined within the language, in a library, or completely unsupported? How about quaternions and octonions?
3. Are characters differentiated from strings of length one? Are strings just lists of characters or is there a distinct string type?
4. How do we access a string's (encoded) bytes, Unicode scalars (characters), custom codes, or graphemes?
5. Is the length of a string, or of any kind of collection, part of its type?
6. Is there a boolean type or must we use 0 and 1?
7. Are the basic types like numbers, booleans, and strings, wired so strongly into the language that reserved words define them? Or do they simply live in a standard library like all other user-defined types?
8. Does the language have a different set of static types from its dynamic types?
9. Can the programmer obtain the type of an expression via a function call, method call, or `typeof` operator? If so, is the response just a string naming the type, or a first-class type object?
10. Are types distinguished from classes?
11. Can types have supertypes? If so, more than one? If so, how is the inherited behavior determined in the presence of name conflicts?
12. Can we add new types?
13. Does the type system have top and bottom types?
14. Is there a language-intrinsic structure for a completely immutable "record" type?

15. Can we conveniently make both sum and product types? Are there tagged or labeled versions of each (i.e., regular vs. tagged unions, tuples vs. records)?
16. Are there explicit pointer types? If so, can we do arithmetic on instances?
17. Do function values have types? Is the return type part of the function type? What about the types of the parameters? Is the order of parameters significant in determining the function type?
18. Can we define parameterized types? If so, are they parameterized only by other types, or by arbitrary values?
19. If the language has a static type system, with type constraints on variables, can a variable's type be narrowed according to certain operators? According to a program's control flow?
20. What are the covariance, contravariance, and invariance rules for parameterized types?
21. What kind of type expressions are allowed in parameterization? Can we go beyond Java's `<? super T>` and `<? extends T>` mechanisms? Are we only allowed simple generic constraints? Are the `in` and `out` variance markers of TypeScript and Kotlin supported?
22. Are types first-class? Are type variables allowed? If so, can they be...assigned?
23. If types are objects, are all types instances of a single type (perhaps called `Type`)? Do type classes exist in the language? If so, can a type belong to more than one type class? Can a programmer define new type classes?
24. Do types have types? Do typeclasses have types? Does the language support kinds, or higher-kinded-types?
25. Can we create types that depend on the values of certain expressions (**dependent types**)? If so, what kind of expressions are allowed in dependent type specifications?

17.2 STRUCTURE

A language's type system allows us to chunk primitive bits into values, which can be numbers, text, data structures (made from sums and products), or even functions. But these values have to be manipulated in some way. A language has to give us a way to build, structurally, a program out of components. These components, or **entities**, vary from language to language; some of the more common and well-known ones are literals, variables, blocks, functions, statements, events, exceptions, types, interfaces, classes, fields, methods, modules, and threads. In this section, we'll consider questions that can help us build an ontology for concepts of program construction. Let's look first at the smaller parts of expressions—literals and variables:

1. For which types are literals allowed to be written? We expect literals for numbers, booleans, and strings, but are arrays, sets, dictionaries, optionals, and general objects also representable in literal form?
2. Do variable declarations have to be annotated with a type, or can they be inferred? What about function parameters?
3. Can the type constraint of a variable change?
4. Can a variable be marked as being mutable or immutable? If it can be marked immutable, does this mean that the variable may not be rebound, or that the object bound to the variable may not be mutated?

5. Are there restrictions on which values can be stored directly in a variable? That is, can only variables of certain types be stored directly while for others only references may be stored?
6. Can variables be bound only once or multiple times? Can they be bound via destructuring or pattern matching? Are they bound in parallel or sequentially?
7. How do we know, when looking at an expression, to *which* entities do each of the variables refer? In other words, how do we determine the **scope** of our bindings?
8. Can the same name refer to multiple entities within the same scope? Can an entity be referred to with multiple names in the same scope?
9. Are there lexical mechanisms for distinguishing local variables from global (or other nonlocal) variables? Or is this resolved by scope computation within a compiler?
10. Are variables allowed to shadow variables of the same name in outer scopes? If not, are name collisions prohibited or do they simply create new locals or refer to globals depending on context?
11. Which constructs define scope? Function bodies only? Certain compound statements but not others? Is there a block construct? `let`-expressions?
12. When a variable is updated, what happens to the "old" value?

Now for some questions regarding more complex expressions:

1. Are expressions evaluated when they are first encountered, or only when needed?
2. Is the type of every expression known at compile time (static typing) or not known until run time (dynamic)?
3. If the language detects that an operator is given operands of a type that it does not expect, does an error occur (strong typing) or is there an attempt made to convert the operands to new operands of an acceptable type (weak)?
4. Can we overload operators? Can we define new operators? If so, how do we specify precedence and associativity? Are all new operators given the same precedence, or can we choose our own?
5. Can we change the precedence of "built-in" operators?
6. Are operators built-in, or are they just functions from a standard library?
7. Are loops expressions (as opposed to statements)?
8. Can operators be extended to accept multiple arguments or does each operator have one and only one arity?
9. Which "magic" expressions does the language provide? Is there anything similar to JavaScript's `this`?
10. Can the programmer specify when dynamically allocated memory is freed? Is all garbage collection automatic, or does the runtime employ reference counting? Are there both strong and weak references? Does the language have any notion of ownership to help with memory management?
11. Must the operands within an expression be evaluated in a certain order (left-to-right, right-to-left) or is this left to an implementation? Is the implementation allowed to evaluate operands in parallel?
12. Does the language require that expressions be reduced in a particular fashion? For example, must `square(inc(8))` be reduced to `square(9)` or to `inc(8)*inc(8)`?

Moving up from the level of expressions, software systems need to have a structured, physical organization to be understandable. Often, a programming language may provide direct support for subsystems, rather than leaving it to "builder scripts" to cobble together the system. Whether they are called "modules," "packages," or "clusters," language designers can work with these in many ways. We may ask:

1. What kind of top-level constructs are defined by the language? Scripts and modules? Scripts only?
2. Can top-level modules be compiled separately? That is, are modules standalone entities that can be used in multiple scripts? Must a module's specification and implementation be separate?
3. How does one arrange to hide entities within a module from the outside world? Are they marked private? Or are entities private by default, requiring explicit export?
4. Can access permissions be changed at runtime?
5. Are the types of entities in the module retained when imported? (Don't be surprised at this question. In early languages, they were not!)
6. Can modules be nested within functions? Within other modules?
7. How does a module import entities from other modules? Implicitly? Via an explicit import statement? If a module is nested within an outer scope, are the outer entities automatically imported?
8. Do entities in a nested module shadow entities of the same name?
9. Are the modules first-class?
10. Is a special name given to higher-level modules (e.g., "crate," "package", "library")?
11. Are modules a logical concept that transcend multiple compilation units (files)? Or does the language expect one module per file?
12. Can modules be parameterized, so that new modules are created upon instantiation of a module template?

If you've ever found yourself in need of writing multiple constructs that all look about the same, you'd have certainly appreciated a language with good metaprogramming facilities. Metaprogramming, the art of writing programs that write other programs, or modify or extend the program that is currently running, is definitely one of the more fascinating areas of programming language design and pragmatics. It relies, of course, on reflection and introspection. What options do language designers have here?

1. Can a program ask which classes have been loaded?
2. Can a program ask which variables are global? Local? Properties of an object?
3. Can a program ask for the currently executing function, and perhaps, its caller?
4. Can we get a handle to the currently running thread? Or process? And its status (active, waiting, terminated, etc.)?
5. Given a function, can the program ask which parameters, and of which type, the function would accept as arguments? Can it ask for the return type of the function?

6. Given a class or module, can the program ask for its members, constructors, methods, initializers, etc? Its supertypes and subtypes?
7. Can a variable be read or set via its name (a string) only?
8. Can a function or method be invoked by its name (a string) only?
9. Can new variables or types or functions be created at run time, given only their name and some indication of how they should look?
10. Does the language have a macro system? If so, do macros generate strings of code which must then be scanned, parsed, and compiled? Or do they generate abstract syntax trees?
11. Is it possible to unquote within a macro?
12. Are macros hygienic? What syntactic devices are provided to explicitly capture in-scope variables, if any?

17.3 EXECUTION

In the previous section, we worked explored several "static" portions of a programming language ontology, focusing on how programs were built from components. Now we turn to questions of a more dynamic nature, asking what programs *do*. We'll start with attempting to organize knowledge about statements:

1. Are statements, expressions, and declarations distinct syntactic and semantic entities, or is everything—or perhaps *nearly* everything—an expression?
2. How do we express the ordering of evaluation or execution? Can we combine statements sequentially (s_1 `;` s_2), in parallel (s_1 `||` s_2), or even via some kind of nondeterministic choice mechanism (s_1 `|` s_2)? Or are we allowed to express sequencing only via composition of function calls?
3. In a multiway selection, must guards be evaluated in any order or a specific order? Do we evaluate only enough guards until we find one that is true, or must we evaluate them all?
4. In the case of a nondeterministic choice construct, what guarantees does the language place on its implementors to ensure, or to not ensure, fairness among choices?
5. Are short-circuiting boolean operators (such as `&&` and `||`, or their generalizations, `all` and `any`) available?
6. Is there a looping construct or must we rely on tail recursion?
7. Can we conveniently express looping mechanisms for all of the usual patterns: forever, n times, through a range of numbers (with an optional step), through a collection, or while a condition is true?
8. Can if-statements and while-statements be controlled by the success or failure of match, as opposed to just boolean test conditions?
9. Are there iterator objects? Or can iteration be implicit? Can we do both? What happens when a collection is changed during iteration?
10. Can we break out a loop? Continue to the next iteration? Start the current iteration over? Start the entire loop over?

11. Are exceptions relied upon for control flow, or are result objects (tagged unions with success and failure) or option objects favored?
12. Does the language provide a `goto` statement? If so, what restrictions, if any, are placed on its use?
13. Does the programmer have access to a timer, either for sleeping or delaying execution?

A function, or procedure, or subroutine, is named block of statements and expressions, possibly with parameters. How can we understand functions? After all, what could be simpler to understand than passing arguments to a function and waiting for the result to be computed? Quite a lot, it turns out! We can ask:

1. Must a function be called with a specific number of arguments? What if we pass too few arguments? Too many? Can we roll up many arguments into a single parameter?
2. Can we (or must we, or must we not) specify the names of the parameters in the call?
3. Can we arrange that certain parameters accept only arguments of specific types?
4. Can we change the value of a parameter within the function? If so, does this action also change the variable holding the argument (if any)? If it does, is the change made immediately or only at the point the function returns?
5. Is a parameter of the called function a distinct variable from its corresponding argument, or just an alias for the argument?
6. If a complex expression is used as an argument, is it evaluated once before the call? Or evaluated fresh every time the parameter is used inside the call?
7. If all arguments are evaluated before the call, in what order are they evaluated? Left to right? Right to left? Arbitrary order? In parallel?
8. How do we arrange for a function to return something? Through a special variable? A `return` statement? Implicitly as the last expression? Some other way?
9. Can a function return zero values? Must it return one? Can it return many?
10. Are there any restrictions on the types of things a function is allowed to return?
11. How do we (syntactically) distinguish between the function itself as a value, and the value resulting from a call of the function?
12. Can the function itself (not just its result) be assigned to a variable? Passed as an argument? Returned as a value from another function?
13. Can a function call itself? If so, are its variables shared among all invocations or does each invocation have a distinct set of variables?
14. Can a function access variables defined outside of itself? If so, must they be explicitly imported or are they available by default?
15. Can "local" variables (defined inside the function) be seen by outside code, or are local variables completely private to the function?
16. If a function can access nonlocal variables, does it use variable bindings in effect in its *caller*, or those in effect in the textually surrounding environment?
17. If a function can access nonlocal variables and is passed somewhere where those (nonlocal) variable names have different bindings, which bindings are used when the function is eventually called?

18. Is it possible to test two functions for equality? If so, what does it mean for two functions to be equal? Same function object? Same source code? Same input-output behavior?[1]
19. Is a function like a regular object—one that we can attach properties to?
20. Do functions have to have names?

Program execution is much more than running a sequence of operations consisting of basic statements and function calls. While sequential execution is necessary, software systems are made of *multiple* sequential threads of control. Describing a program in terms of independent actors doing their own thing concurrently with each other, and communicating and coordinating their activities, is an excellent way to model real-world activities. There are many known approaches to concurrent programming. Perhaps a few questions can help us with a conceptual ontology:

1. How does one decompose a program for concurrent execution?
2. Are there linguistic constructs for coroutines, threads, and processes, or are these "just functions"?
3. Can we define concurrent execution at different levels of granularity? Expression-level? Function-level? Program-level?
4. Does the language execution model consist of a single thread in which blocks of code are executed in response to events? Or can multiple threads exist?
5. If multiple threads can run concurrently, can they preempt each other? Or do threads run until a point in which they allow themselves to yield control?
6. Can mutable memory be shared? If so, do there exist constructs other than explicit locks that help to prevent race conditions? Is there support for transactional memory?
7. Is lock-free synchronization (e.g. atomic integers) available?
8. Are tasks spawned implicitly, at the time their enclosing block begins execution, or must they be explicitly started?
9. Does the runtime unceremoniously kill tasks when the enclosing block, or the spawning task, terminates? Or does the enclosing/spawning task wait for all internal tasks to terminate?
10. Are tasks spawned with or without a link to the task doing the spawning?
11. Are we able to keep track of task information at compile-time, so as to guarantee protection from race conditions?
12. If an asynchronous task is launched, how is a result obtained? Via a callback, promise, or some other mechanism?
13. What mechanisms exist to wait for a group of tasks to all finish? For the first of many to finish?
14. Is message passing between processes done by mentioning processes by name (or process identifier), or through specific channels, or are messages put into a global pool where they can be processed by any willing process?
15. If channels are used to wire processes for communication, must the wiring be fixed at compile time or can we rewire during execution?

[1] Readers familiar with the theory of computation may find this question of particular interest.

16. Is message passing only synchronous or only asynchronous? Can the programmer choose?
17. Can an asynchronous call be canceled if the other party does not connect within a given duration? Do we have a conditional synchronization mechanism (to connect *only* if the other party is ready and waiting)?
18. How, if at all, do we detect that a process has crashed?
19. What happens if a message is sent to a terminated process? Is an error generated or is the message ignored?
20. Do introspection mechanisms exist to get the state of a process (alive, dead, waiting, etc.)? If so, can we ensure that the state is still valid by the time we are ready to act upon it?

Finally, languages may have strong opinions about how programs should look and feel in terms of execution. Perhaps they are attempting to strictly follow a particular computational paradigm, or are trying to be accepting of multiple ways of doing things. The former approach tends to favor elegance and simplicity, as the language will have very few, easily recognizable building blocks. The latter favors pragmatism, since not every problem naturally fits into a particular programming style. Perhaps, too, details of the execution context—the real or virtual machine on which the program runs—may leak into language designs. Therefore, understanding execution platforms empowers you to write code that is not only more expressive, but more performant: you will be aware of how to make computations consume resources wisely. Let's approach these ideas through a series of questions:

1. Do programs in the language have a lot of mutable variables and statements? Or are virtually all variable immutable, with "control flow" emergent only by composition of functions operating on parameters and persistent data structures?
2. Does the language have a declarative feel, in which programmers state what they want and rely on a behind-the-scenes inference engine to obtain desired results?
3. Do programs in the language look like data processing pipelines?
4. Is "everything an object"?
5. Is "everything a function"?
6. Are there distinct ways of denoting variables stored "with the executable" (static), within a regular function (stack), or dynamically allocated (heap)?
7. Is there an operating system interface? That is, does the language allow you to create new processes, or read and write files, or make network requests?
8. Is there a way to embed assembly language or machine language? Is there a library of **intrinsic functions** that map directly to machine instructions?
9. Does the language advertise itself as *mutiparadigm*?
10. Are programs strings of text, or built from graphical blocks?

CHAPTER SUMMARY

In this chapter, we've laid out a collection of questions to ask about programming languages with the goal of framing language concepts into a rough ontology. We've grouped our questions into three main areas: (1) information representation and classification, specifically the classification of values into types; (2) program structure (architecture), considering

building blocks known as entities; and (3) execution, considering statements, functions, and concurrency mechanisms. We did not present an outline-oriented ontology, but we were able to show the breadth of the field and provide a very rough road map for organizing one's linguistic studies.

The previous chapters may have answers to many of these questions for our featured languages, but certainly not all of the questions have been answered, let alone asked. We encourage you to not only research answers to some of these questions but to create new questions of your own. Such exercises can help solidify knowledge gained throughout the tour, and provide a sense of how modern languages support multiple programming styles. Many of these concepts should carry over well into the study of new languages.

Do keep in mind that this book aims to expose the reader to linguistic theory and practice via *explorations* of various topics. Where this book lacks depth in both programming styles and theoretical foundations, other works can assist you in those areas. For the former, we recommend Cristina Lopes's excellent and comprehensive account of 37 programming styles and their implementation [66]; and for the latter, the works by Turbak and Gifford [111], Pierce [89], Harper [42], and Stump [107]. Theoretical studies will provide you with insights and skills into designing the languages of the future.

EXERCISES

17.1 Answer a number of the questions from this chapter for your favorite language(s), or for languages of your choice.

17.2 Add a question or two of your own to each section from the chapter.

Glossary

The lexicon of programming languages is quite large, worthy of collecting some of the more important terms in a single place, such as this very glossary. To keep it brief, you won't find any examples here—just bare-bones definitions. Here you'll find many of the terms encountered throughout the book, as well as quite a few new ones.

Abstract Class A class that cannot have any instances.

Abstract Method A method of an abstract class that has no implementation.

Activation Record See *frame*.

Actor A concurrent process that communicates by sending and receiving messages.

Agent An entity that acts on behalf of another entity.

Alias One of the names of a multiply-named entity. An entity to which multiple names are bound is said to be *aliased*.

Annotation A form of metadata attached to an entity.

Argument An expression passed to a (parameterized) entity during a call, instantiation, or other invocation.

Assignable A mutable storage location. Used in place of the term *variable* when emphasizing mutability.

Assignment The act of associating a value with a (mutable) variable.

Asynchronous A type of invocation in which the caller does not wait for the callee to complete.

Atom A named primitive entity whose value is, essentially, itself. Also known as a *symbol*.

Atomic Object An object that can be operated upon safely in a multithreaded environment without locks. Often the object holds a numeric value on which "compound" operations such as compare-and-set or get-and-add can be done atomically.

Automatic Reference Counting (ARC) A mechanism in which the compiler inserts code to release and retain blocks of memory so the programmer does not have to.

Binding An association of a name with an entity.

Borrow The *use* of a resource without taking full ownership.

Bottom Type The (concrete) type in a type system that has no values, and is hence a subtype of every type.

Callback A function f both passed to a function g and invoked within g.

Channel A conduit for messages in a distributed architecture.

Character A unit of textual information.

Character Set A set of characters together with a mapping from integers to these characters.

Class An entity from which objects are created. Objects instantiated from a class have the structure and behavior defined by the class, though in some languages instances are allowed to be customized.

Class Variable A property that belongs to a class, rather than to the instances of the class.

Closure A block or function with free variables that can use values from the environment in which it is defined.

Code Point The unique integer value assigned to a character in a character set.

Coercion An implicitly applied type cast.

Comprehension A composite value defined by means of an expression that generates the components.

Concrete Class A class that is allowed to have instances.

Concurrency The modeling and coordination of independent computing activities whose execution spans *may* overlap in time.

Continuation Roughly, a representation of "the rest of the program."

Contravariance The existence of a subtype relationship between `T<V>` and `T<U>`, where T is a parameterized type and U is a subtype of V.

Coroutine A line of execution that, under explicit programmer control, can yield to other coroutines and be resumed. Coroutines share a processing unit in the sense that only one coroutine is executing at a time.

Covariance The existence of a subtype relationship between `T<U>` and `T<V>`, where T is a parameterized type and U is a subtype of V.

Curried Function A function of one parameter that returns a function of one parameter, in contrast to a function that takes two parameters (or a pair of objects).

Dangling Pointer An in-scope pointer variable whose referenced memory has been freed.

Deadlock A situation in which two or more competing threads are blocked forever, each waiting for the other to finish.

Decimal Type A datatype that stores the decimal digits of numeric quantities.

Decorator An object providing additional behavior to another object.

Deferred Execution A block of code executed at some point in the future, generally in response to a triggering event.

Deep Copy A copy which recurses through an an object graph, copying all referenced values.

Dependent Type A type that depends on a value or constraint, such as the type of trees with height 8, or the type of all sorted lists.

Dereference The evaluation of the the object referenced by a (pointer) variable.

Destructor Code (generally a function) executed when an object is destroyed.

Dictionary A set of key-value pairs, with unique keys. Also known as a *map*.

Discriminated Union See *tagged union*.

Dynamic Language A language in which tasks such as modifying code and adding types can be done at run time.

Dynamic Typing The state of having types of some expressions not known until run time.

Eager Evaluation The evaluation of all of a function's arguments (or an operator's operand) before application of the function (or operator). Also known as *strict evaluation*.

Encapsulation The bundling of data elements into a single component, often in a way that the constituent elements cannot be directly accessed.

Encoding 1. The representation, in bytes, of an entity. 2. A specification of how certain entities should be encoded in bytes.

Enumeration A complete ordering of a given set of elements. The term may refer either to (1) a type whose constituent values are listed (an *enumerated type*) or (2) a sequence of values emitted by a generator.

Event An occurrence that can be detected and handled.

Exception An entity that is *thrown*, or *raised* to abort the current normal control flow and transfer control to the point at which the exception is *caught*.

Falsy Treated as false in a boolean context (e.g, when evaluating conditions).

First-class Able to be assigned to variables, passed as parameters, and returned from routines.

Fixnum An integer type represented with a fixed, finite, number of bits.

Frame A structure in the execution model of a program representing a particular activation of a routine. A frame records the parameters and local variables of the activation, as well as additional bookkeeping information. Also known as an *activation record*.

Function A possibly parameterized, and possibly named, callable block of code.

Functional Programming A programming paradigm in which computation proceeds entirely by evaluating side-effect-free expressions, generally involving function calls.

Future See *promise*.

Garbage Collection The reclamation of storage holding objects that are no longer needed.

Generic Able to be specified just once and "work in about the same way" in many different contexts.

Guard A boolean expression used to allow (when true) or disallow (when false) access to an entity.

Heap Storage The portion of memory in which new objects are allocated (and deallocated) at run time.

Higher-Order Function A function that takes functions as arguments or returns functions.

Homoiconicity The property of a language in which code is represented as a data structure and both have the same syntactic forms.

Hygienic Macro A macro whose expansion will not capture identifiers.

Immutable The quality of an entity whose state may not change.

Inheritance The property of a class or object acquiring the same structure or behavior of another entity, which can then be optionally specialized.

Instance An object (when referred to in reference to its class or interface).

Interface A programming structure that enforces certain behaviors on an object.

Interoperability The capability of two or more programming languages to interact and operate on shared data or invoke shared functions.

Interpolation The process of evaluating an expression with placeholders, generally a string literal, to yield a result in which placeholders are replaced with computed values.

Iterator An object that provides the capability to access the elements of a container in some order.

Keyword Argument An argument that is matched to a corresponding parameter using the parameter's name, rather than by its position in a call.

Lazy Evaluation An evaluation strategy for expressions in which evaluation is delayed until an actual value is needed.

Lifetime The time between an objects' creation and destruction. Also known as a *duration*.

Literal A representation of a value in source code.

Lvalue A storage location into which values can be assigned.

Macro A rule or pattern that specifies how an input should be mapped to a replacement output. Often used in the context of rewriting source code prior to compilation.

Mailbox A container for data into which one process writes data for another process to read (generally in an asynchronous fashion).

Match An object storing information regarding the successful location of a pattern in some data.

Memoization An optimization technique for repeatedly-called functions that caches results and returns cached results when the same inputs occur.

Metaclass A class whose instances are classes.

Method A function intimately associated with an object.

Module A language construct wrapping a collection of (usually encapsulated) related entities, providing an interface to other modules of a system.

Monad Roughly, a generic type $M\alpha$ supporting (1) an operation to "wrap" a value from the underlying type (e.g., $\alpha \to M\alpha$) and (2) an associative chaining operation of type $M\alpha \to (\alpha \to M\beta) \to M\beta$ for which the wrapper operation is an identity.

Monitor An object or module providing mutually exclusive access among its clients to achieve thread safety.

Null A value representing the absence of information.

Null Reference A reference without a referent.

Object 1. An entity in a program intended to be stored in memory. 2. In OOP, an instance of class.

Operator A function, generally, though not necessarily, named with nonalphanumeric symbols that computes well-known (often algebraic) and widely applicable operations.

Operator Overloading The allowance for a single named operator to have different implementations depending on its arguments. For example, the name `*` can overload integer multiplication, floating-point multiplication, string repetition, vector dot product, or vector scalar product.

Option Type A type whose values are wrappers that either contain or do not contain a wrapped value.

Parallelism The state of a running program in which multiple physical execution units (e.g. cores) are actively computing at the same time.

Parameter A variable to which arguments are passed.

Pass-by-Name An argument passing mechanism in which an argument is not evaluated prior to a call, but rather within the function using (essentially) the text of the argument expression.

Pass-by-Reference An argument-passing mechanism in which the parameter simply aliases the argument, so that mutations through the parameter are immediately applied to the argument.

Pass-by-Sharing An argument-passing mechanism in which only (a copy of) the object identifier of the argument is passed.

Pass-by-Value An argument passing-mechanism in which an argument is evaluated and then (a copy is) subsequently passed to the parameter.

Pass-by-Value Result An argument-passing mechanism in which an argument is evaluated before the call, passed by value to the corresponding parameter, and, after execution of the callee, any changes made to the parameter are copied to the corresponding argument.

Persistent Data Structure A data structure that retains the prior version of itself when "modified."

Pointer An object that references a location in memory.

Polymorphism The allowance for a single name to be bound to multiple entities at the same time, generally with the understanding that any invocation of the name will refer to the desired entity based on context.

Promise Roughly, a placeholder for an (initially) unfinished computation. A promise is immediately returned when an asynchronous computation is launched to be queried later or used in a callback. Also known as a *future*, though in some contexts, futures and promises are distinguished.

Prototype 1. An object from which other objects are derived. 2. An object to which property requests are forwarded by an object that does not own a property of a given name. 3. In C, a function signature.

Proxy An object that provides an interface to another object. The proxy takes requests on behalf of the proxied object and may modify or deny these requests before forwarding them to the proxied object.

Reactive Programming A programming style centered on streams of data that are managed asynchronously.

Record An object whose components are labeled values.

Reference A value that refers to an entity, called its referent. Often used to allow sharing of data. A reference may be *dereferenced* to obtain the referent. References are frequently implicit, but in some languages may be distinct objects in their own right. In C++, a reference is simply an alias.

Regex See *Regular Expression*.

Regular Expression A pattern that describes sets of strings according to a large number of widely-accepted mechanisms, and sometimes with programming-language-specific mechanisms.

Reserved Word A keyword that cannot be used as an identifier.

Runtime The period of time in which a program is being executed.

Rvalue A value that can be assigned, so named because traditional assignment statements are written with the destination storage location on the left (hence the term l-value) and the data to be assigned on the right (hence the term r-value).

Scope The region of source code in which a binding refers to a given entity.

Scripting Language 1. A language designed for running jobs, i.e., coordinating (or scripting) the activities of other programs. 2. A language low on "ceremony" but rich in expressive features, designed to allow its programmers to be productive.

Second-class An entity that is not allowed to be represented by a variable in an expression.

Semaphore A variable or object used to permit only a specific number of threads to access a shared resource at a time.

Shallow Copy A component-by-component copy of the values of the original object, in which references are not followed but are themselves copied.

Sigil A symbol affixed to a variable name that shows the variable's scope.

Signal A value that changes over time but that is processed as a stream.

Signature A specification of a function, method, or module that is completely free of implementation details, providing the minimum amount of information to use the entity.

Single Inheritance The restriction of a type or class to have only a single supertype or superclass.

Software Transactional Memory A mechanism for controlling updates in a concurrent environment in which all updates within a marked transaction either happen or are rolled back.

Splat A term used primarily in Ruby referring both to a function parameter that is to be unpacked and to a function argument spread. (See *spread*.)

Spread An operator that expands a single expression into multiple expressions, within the context of a collection literal, argument list, or similar construct.

Stack Storage Location in memory used to store frames.

Starvation A scenario in which a process or thread cannot proceed because it is continually denied requests to resources it requires.

Static Language A language in which a large number of rules are enforced by examining the source code prior to execution.

Static Storage The region of memory used to store data whose lifetime coincides with the entire program.

Static Typing The inference of and checking of types prior to execution.

Stream A sequence of objects made available over time.

Strict Evaluation An evaluation strategy in which all operands are evaluated before being given to an operator.

Subclass A class derived from another class, inheriting many of the original class's properties and optionally adding its own properties and behaviors.

Superclass A class that has been subclassed.

Symbol A named primitive entity whose value is, essentially, itself. Also known as an *atom*.

Synchronous A form of communication in which the caller must wait for the callee to be ready to accept the call and must wait for the callee to finish the requested service before proceeding.

Syntactic Salt Syntactic forms that tend to make programs verbose, or difficult to follow, even though they may have been added to the language with good intentions.

Syntactic Sugar A syntactic form that improves upon an equivalent primitive form by being more concise, expressive, powerful, or elegant.

Systems Language A programming language used for writing system software.

Tagged Union A union type whose members are each tagged with a label indicating the underlying type from which they came. Also known as a *disjoint union*, *discriminated union*, *disjoint sum*, or *discriminated sum*.

Thread A sequential flow of control, generally preemptible and generally able to share memory with other threads.

Top type The type in a type system containing all values, and is hence a supertype of all types.

Trait A set of methods, generally expected to be mixed in to a class to enhance its behavior or to increase its capabilities.

Transient The property of any element in a system that is not persisted or serialized.

Transpiler A translator between two high-level programming languages.

Truthy Treated as true in a boolean context (e.g., when evaluating conditions).

Type A set of values with common behavior enforced by the programming language.

Type Compatibility The property of a relationship between an expression e and type t in which e can be used in any context expecting a value of type t.

Type Inference Automatic deduction of the type of an expression.

Type Variable A variable that ranges over types.

Union Type A type whose set of values is the union of zero or more other types.

Value A unit of data.

Variable A storage location containing a value.

Volatile Variable A variable that must be made available to multiple threads and thus cannot be implemented with multiple divergent copies per thread.

Wildcard A character in a pattern meant to be replaced.

Zero Cost Abstraction A feature that looks complex but has zero overhead or performance penalty.

Bibliography

[1] Dave Abrahams. Protocol-oriented Programming in Swift. https://developer.apple.com/videos/play/wwdc2015/408/, 2015. Presented at Apple WWDC 2015.

[2] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49), 1996.

[3] American Heritage Dictionary of the English Language. Definition of "theory". https://ahdictionary.com/word/search.html?q=theory.

[4] Chris Angelico, Tim Peters, and Guido van Rossum. PEP 572—Assignment Expressions. https://peps.python.org/pep-0572/.

[5] Apple, Inc. Concurrency. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/.

[6] Apple, Inc. iOS. http://www.apple.com/ios/.

[7] Apple, Inc. Swift Standard Library Operators Reference. https://developer.apple.com/reference/swift/1851035-swift_standard_library_operators.

[8] Apple, Inc. The Swift Programming Language.

[9] Jeremy Ashkenas. List of Languages that Compile to JS. https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js.

[10] Ada Resource Association. Introducing Ada 95: First Internationally Standardized, Object-Oriented Language. http://www.adaic.org/ada-resources/standards/ada-95-documents/95intro/.

[11] Gary Bernhardt. WAT. https://www.destroyallsoftware.com/talks/wat, 2012.

[12] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. *SIGPLAN Not.*, 48(9):133–144, September 2013.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[14] CERT. MEM30-C. Do not access freed memory. https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory.

[15] Brad Chamberlain. Chapel: Productive Parallel Programming. http://www.cray.com/blog/chapel-productive-parallel-programming/, 2013.

[16] Catalin Cimpanu. Microsoft: 70 Percent of All Security Bugs are Memory Safety Issues. https://zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019.

[17] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[18] Abdul Dakkak. How Mojo Gets a 35,000x Speedup over Python—Part 1. https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1.

[19] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[20] Mark Davis. Unicode Text Segmentation. Technical Report 29, Unicode Consortium, 2015.

[21] Steve Dekorte et al. Io: A Programming Language. http://iolanguage.org/.

[22] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[23] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[24] Edsger W. Dijkstra, Fraser Duncan, Jan V. Garwick, C. A. R. Hoare, Brian Randell, G. Seegmueller, Władisław M. Turski, and Michael Woodger. News item—minority report. *Algol Bulletin*, 31:7–7, March 1970.

[25] Melissa Elliott. PHP Manual Masterpieces. http://phpmanualmasterpieces.tumblr.com/.

[26] ETH Productions. golfing-langs. https://github.com/ETHproductions/golfing-langs.

[27] Stuart Feldman. A conversation with Alan Kay. *ACM Queue*, 2(9):20–30, December 2004.

[28] Andrew Gerrand. Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating, 2010.

[29] The Go Authors. The Go Programming Language. https://golang.org/.

[30] The Go Authors. The Go Programming Language Frequently Asked Questions. https://golang.org/doc/faq.

[31] The Go Authors. The Go Programming Language Specification. https://golang.org/ref/spec.

[32] Google. Android. https://www.android.com/.

[33] James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. Sun Microsystems, 1995.

[34] Paul Graham. Fortran I. http://www.paulgraham.com/history.html.

[35] Paul Graham. Lisp Quotes. http://www.paulgraham.com/quotes.html.

[36] Paul Graham. Revenge of the Nerds. http://www.paulgraham.com/icad.html.

[37] Paul Graham. The Roots of Lisp. http://lib.store.yahoo.net/lib/paulgraham/jmc.ps, 2002.

[38] Isaac Grosof. Pyth Documentation. https://pyth.readthedocs.io/en/latest/index.html.

[39] Chet Haase. Google I/O 2019: Empowering Developers to Build the Best Experiences on Android + Play. https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html.

[40] James Hague. Puzzle Languages. http://prog21.dadgum.com/38.html, 2009.

[41] Paul Hallett and other PokéAPI contributors. PokéAPI: The Restful Pokemon API. https://pokeapi.co/.

[42] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016.

[43] Haskell Wiki Community. Monad Laws. https://wiki.haskell.org/Monad_laws.

[44] Haskell Wiki Community. Rank-N Types. https://wiki.haskell.org/Rank-N_types.

[45] Rich Hickey. Clojure. http://clojure.org/.

[46] Rich Hickey. Clojure Core Library Source Code. https://github.com/clojure/clojure/blob/master/src/clj/clojure/core.clj.

[47] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[48] C. A. R. Hoare. Communicating sequential processes, 1985.

[49] Tony Hoare. Null References: The Billion Dollar Mistake. *InfoQ*, August 2009.

[50] Allen Holub. Why Getter and Setter Methods are Evil. *Java World*, 2003.

[51] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 1985.

[52] IETF Trust. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, Internet Engineering Task Force, 2014.

[53] Stack Exchange Inc. StackOverflow Annual Developer Survey. https://survey.stackoverflow.co/, 2011.

[54] Intel Coporation. Intel 64 Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/microarchitecture/intel-64-architecture-general.html.

[55] ISO. Information Technology—Programming Languages—Ada. ISO/IEC 8652:2012, International Organization for Standardization, Geneva, Switzerland, 2012.

[56] JBoss. Javassist. http://jboss-javassist.github.io/javassist/.

[57] Kris Jenkins. What Is Functional Programming? http://blog.jenkster.com/2015/12/what-is-functional-programming.html, 2015.

[58] JetBrains. Kotlin Coroutines Guide. https://kotlinlang.org/docs/coroutines-guide.html.

[59] JetBrains. Kotlin Grammar. https://kotlinlang.org/docs/reference/grammar.html.

[60] Alan C. Kay. The Early History of Smalltalk. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 511–598. ACM, New York, NY, USA, 1996.

[61] Stein Krogdahl. The Birth of Simula. https://portablesimula.github.io/github.io/doc/HiNC1-webversion-simula.pdf.

[62] Kwang Yul Seo. TypeScript: Best Common Type. https://kwangyulseo.com/2015/06/12/typescript-best-common-type/.

[63] Kx Systems. *K Reference Manual*. Kx Systems, Inc., 1998.

[64] Angelika Langer. Java Generics FAQs. http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html.

[65] Barbara Liskov. A History of CLU. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 471–510. ACM, New York, NY, USA, 1996.

[66] Cristina Lopes. *Exercises in Programming Style*. Taylor & Francis, 2014.

[67] Lua.org, PUC-Rio. About Lua. http://www.lua.org/about.html.

[68] Lua.org, PUC-Rio. The Lua 5.4 Reference Manual. http://www.lua.org/manual/5.4/.

[69] Robert C. Martin. The Barbarians are at the Gates. https://blog.8thlight.com/uncle-bob/2011/12/11/The-Barbarians-are-at-the-Gates.html, 2011.

[70] Yukihiro Matsumoto. Ruby 1.9, Google Tech Talk. https://www.youtube.com/watch?v=oEkJvvGEtB4, 2008.

[71] Paul Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–672, August 2014.

[72] Microsoft. .NET. http://www.microsoft.com/net.

[73] Microsoft. The TypeScript Handbook—Narrowing. https://www.typescriptlang.org/docs/handbook/2/narrowing.html.

[74] Microsoft. The TypeScript Handbook—tsc CLI Options. https://www.typescriptlang.org/docs/handbook/compiler-options.html.

[75] Microsoft. TypeScript Wiki—TypeScript Design Goals. https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals.

[76] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 12 1978.

[77] Don Monroe. Neuromorphic computing gets ready for the (really) big time. *Communications of the ACM*, 57(6):13–15, June 2014.

[78] Natasha the Robot. Swift 2.0: Why Guard is Better than If. https://www.natashatherobot.com/swift-guard-better-than-if/.

[79] Michael Nielsen. Lisp as the Maxwell's Equations of Software. http://www.michaelnielsen.org/ddi/lisp-as-the-maxwells-equations-of-software/.

[80] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[81] Oracle. Java Tutorials: Lesson: Generics. https://docs.oracle.com/javase/tutorial/java/generics/.

[82] Oracle. Outline of the Collections Framework. https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/doc-files/coll-reference.html.

[83] John K Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, March 1998.

[84] OW2 Consortium. ASM. http://asm.ow2.org/.

[85] OWASP. Buffer Overflow. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

[86] Palo Alto Research Center, Inc. Milestones: Xerox PARC History. http://www.parc.com/about/.

[87] Tim Peters. PEP 20—The Zen of Python. https://peps.python.org/pep-0020/.

[88] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[89] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[90] Rob Pike. Concurrency is not Parallelism. https://talks.golang.org/2012/waza.slide, 2012.

[91] Pivotal Software, Inc. Spring Framework. http://projects.spring.io/spring-framework/.

[92] Jonas Platte. turbo.fish. https://turbo.fish.

[93] Rob Pooley. An Introduction to Programming in Simula. https://portablesimula.github.io/github.io/doc/SimulaTextBook.pdf.

[94] Maria Popova. How Ada Lovelace, Lord Byron's daughter, became the world's first computer programmer. *Brain Pickings*, December 2014.

[95] Pressler, Ron. JEP 445: Unnamed Classes and Instance Main Methods (Preview). https://openjdk.org/jeps/445.

[96] Python Software Foundation. Displays for lists, sets and dictionaries. https://docs.python.org/3/reference/expressions.html#displays-for-lists-sets-and-dictionaries.

[97] Python Software Foundation. Python. http://python.org.

[98] Red Hat, Inc. Hibernate. http://hibernate.org/.

[99] The Rust Community. The Rust Programming Language. https://www.rust-lang.org/.

[100] The Rust Project Developers. The Rust Programming Language. https://doc.rust-lang.org/book/.

[101] Jean Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall series in automatic computation. Prentice-Hall, 1969.

[102] Clay Shirky. Ontology is Overrated: Categories, Links and Tags. http://www.shirky.com/writings/ontology_overrated.html, 2005.

[103] sigpfe. You Could Have Invented Monads! (And Maybe You Already Have.). http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html.

[104] Michele Simionato. The Python 2.3 Method Resolution Order. https://www.python.org/download/releases/2.3/mro/.

[105] Daniel Steinberg. Daddy, Are We There Yet? A Discussion with Alan Kay. *OpenP2P*, April 2003.

[106] Bjarne Stroustrup. A history of C++: 1979–1991. *SIGPLAN Notices*, 28(3):271–297, March 1993.

[107] Aaron Stump. *Programming Language Foundations*. Wiley, 2013.

[108] S. Tucker Taft. The ParaSail Programming Language. http://parasail-lang.org.

[109] The Internet Society. HTTP Over TLS. RFC 2818, Internet Engineering Task Force, 2000.

[110] The Internet Society. The Base16, Base32, and Base64 Data Encodings. RFC 4648, Internet Engineering Task Force, 2006.

[111] Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.

[112] Bret Victor. The Future of Programming. http://worrydream.com/dbx/, 2013.

[113] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, Third Edition*. O'Reilly Media, Inc., 3 edition, 2000.

[114] Eric Wastl. PHP Sadness. http://www.phpsadness.com/.

[115] Web Hypertext Application Technology Working Group. Web Wokers (HTML Living Standard). https://html.spec.whatwg.org/multipage/workers.html.

[116] Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Messenger*, 1(1):7–87, August 1990.

[117] Arthur Whitney. K. *Vector*, 10(1):74–79, July 1993.

[118] Wikipedia. Brainfuck.

[119] Wikipedia. Grace Hopper.

[120] Wikipedia. Heap's Algorithm.

[121] Wikipedia. Hindley-Milner Type System.

[122] Wikipedia. Programming Paradigm.

[123] Wikipedia. Quantum programming.

[124] Wikipedia. Simula.

[125] Wikipedia. UTF-16.

Index

A-0, 336
abstract class, 70
abstract method, 131
abstract type, 169, 173
access modifier, 69
action, 201
activation record, 248
actor
    Swift, 310
Ada, **349**
address, 271
adverb, 341
agent, 201, 344
algebraic data type, 224
    Haskell, 214
algebraic effects, 345
Algol, xiii, xv, **337**
Algol 68, xiii
aliasing, 6
alignment, 273
allocated storage duration, 279
ALU, 357
ancestor, 115
APL, **338**, 340
application software, 345
applicative, 230
ARC, *see* automatic reference counting
argument, 24
argument label, 289, 290, 293
arguments, 46
    keyword, 93, 172
    positional, 172
arity, 9
array decay, 277
arrow function, 49
assembly language, 356
assertion, 20
assertion function, 77
associativity, 9
asynchronous, 41, 55
atom, 4
attack surface, 13
attribute, 86, 330
autoboxing, 133
automata theory, 3
automatic reference counting, 289, 307
automatic semicolon insertion, 41
automatic storage duration, 278

Backus, John, xiii
Barton, Bob, 338
BCPL, 346
BDFL, 83
behavior, 338
Bernhardt, Gary, 11
best common type, 66
Big Data, xiii
billion dollar mistake, 127
binary, 9
binding, 47
bits, 360
bivariant, 137
block, 104, 111
    Factor, 342
    Ruby, 111
borrow checker, 315, 323
borrowing, 315, 322
bounded type parameter, 136
Bourne Shell, 343
Brainf**k, **352**
broadcast, 171
bus, 357

C, xiii, **267**
C++, xiii, 337, **346**
C#, **349**
callable, 86
callback, 56
canonical composition, 43
canonical decomposition, 43
case expression, 227
cast, 278
CCS, 347
channel, 237

buffered, 258
Chapel, **348**
characters, 8, 21
class, 8, 338
Clojure, 196
client-server, 351
Clojure, xiii, **189**
evaluator, 193
reader, 193
closure, 14, 28, 294
closure expressions, 294
CLU, xiii, **348**
cluster, 348
COBOL, xiii, **336**
CODASYL, 336
code point, 21
code unit, 126
coercion, 23, 46
coercions, 8
Common Lisp, xiii
companion object, 162
compatibility decomposition, 43
compile-time error, 66
complexity theory, 3
comprehensions, 86
computability theory, 3
computed goto, 336
concatenative programming, 342
concrete type, 70, 173
concurrency, 139, 257, 347
conditional types, 79
conjoin, 195
cons operator, 216
constant, 5
context, 229
contravariant, 137
Control flow analysis, 76
cooperative multitasking, 34
coroutine, 10, 33
cosmic supertype, 255
covariance, 74, 137
crash, 256
crate, 316
Cray Inc., 348
cross-cutting concerns, 97
CSP, 347
curried, 215
custom operators, 300
cut, 340
daemon, 144, 260
Dahl, Ole-Johan, 337
dangling pointer, 248, 278
data class, 153, 159
data race, 331
dataflow machine, 357
decimal type, 7
decorated, 97, 125
deep binding, 27
deep copy, 29
default implicit initialization, 305
default package, 127
deferred operations, 237
definitive initialization, 305
deinitializer, 307
delayed evaluation, 100
delegation, 50
dependent types, 345, 361
dereference, 246
dereference operator, 276
designator, 332
destructuring, 296
destructuring assignment, 47
dimension, 177
directives, 284
discriminated sums, 224
disjoint sums, 224
distributed computing, 183
double free, 282
dynamic polymorphism, 110
dynamic type, 8
dynamic types, 65
dynamically scoped, 27

effects, 9, 211
Eiffel, xiii
eigenclass, 110
Elixir, **347**
Elliott, Melissa, 11
encapsulated, 108
enterprise, 348
entities, 4, 361
entity, xv
enumeration, 296
escape analysis, 248
escape sequences, 22
escaped entities, 248
esoteric languages, 352
Ethernet, 338
Euclid, xiii, 337

event emitters, 10
event handler, 56
event queue, 56
event-driven paradigm, 107
events, 10, 56
exception
    catching, 10
executor service, 141
exit status, 268
export, 11
expression, 9
expression evaluation, 9
expression problem, 202
extension, 289, 303

f-strings, 84
F#, **344**
Factor, **342**
falsy, 46, 58
fill, 93
filter function, 2
fixed formatting, 336
fixity, 9
fizzle, 232
FLOW-MATIC, 336
format specifier, 17
forms, 193
Forth, 342
Fortran, xiii, **335**
frame, 248
free, 6
free variables, 27
function, 10
functional interface, 132
functional paradigm, 107
functional programming, 45
functors, 230
future, 183

garbage, 6
garbage collection, 14
garbage collector, 6, 248
generator expression, 2, 96
generators, 28, 96
generic, 8, 72
generic function, 175, 179
generic type, 297
generic types, 133
global environment, 30
global object, 45
global variables, 45
Go, **237**
Goldberg, Adele, 337
goroutine, xiv
goroutines, 237, 245, 257, **257**
    termination, 257, 261
gradually typed, 67
grapheme clusters, 21
graphemes, 8, 21
guards, 217

Haskell, **211**
header file, 280
higher-order functions, 24, 45
Hindley-Milner type sytem, 220
Hindley-Milner typing, 211
Hoare, Tony, 337
homoiconic, 204
Hopper, Grace, xiii, 336
hygiene, 183
hygienic macro, 169, 181

immediately invoked function expression, 28
immutable, 5, 87
imperative paradigm, 107
implementation blocks, 324
implicit return, 294
import, 11
index signatures, 71
infix, 9
information hiding, xiii
Ingalls, Dan, 337
inherit, 88
inheritance, 31, 295
inherited properties, 50, 58
initializer
    convenience, 295
    designated, 295
insecure, 28
instance, 32, 52, 86
instance method, 303
instance property, 303
instance variables, 108
instantiated, 72
interface, 8, 237, 251
    adoption, 301
    conformance, 301
    Java, 131
intermediate operations, 138

interns, 196
interpolation, 40
intersection type, 68
intrinsic functions, 367
introspection, 118
invariant, 137
Io, **341**
IO action, 212
iPadOS, 289
IS-A, 53
isolates, 351
items, 319
iterable, 95
iterator, 29, 95, 171
iterators, 316

Java, xiii, **123**, 337
Java platform, xiv
JavaScript, xiii, **39**
Joy, 342
Julia, **169**

K, 2, **340**
Kay, Alan, xvi, 107, 337
kind, 8
kitty pidgin, 354
Kotlin, **151**

lambda expression, 132
language evaluation, 11
language tag, 13
language theory, 3
Lattener, Chris, 289
lazily, 220
lazy, 192
lexical closure, 28
lifetime, 248, 315, 323
Linda, 347
line numbers, 335
Liskov, Barbara, 348
Lisp, xiii, xv, **336**
list comprehension, 212
literal, 4
little computers, 2, 338
LLVM, 289
local binding, 195
logic paradigm, 107
LOLCODE, **354**
Lovelace, Ada, 349
Lower bounded wildcards, 136
Lua, **15**

machine instructions, 356
macOS, 289
macro, 316
    Clojure, 191
    Julia, 180
macros, 284
main thread, 140
map function, 2
mapped types, 79
Martin, Robert C., *see* Uncle Bob
matrix, 177
McCarthy, John, xiii
memoization, 98
memory, 6, 338
memory leak, 282, 308
Mesa, xiii, 337
message, 10, 104, 338
metaprogramming, 117, 341
metatable, 31
method, 49, 107, 325
method reference, 132
method resolution order, 89
methods, 10, 108, 175, 179, 247
    object-specific, 110
mixed in, 114
ML, xv, 344
Modula, xiii, 337
modular arithmetic, 7, 305
module, 10, 84, 113
Mojo, xiii
Monad, 230
monad, 230, 345
monitor, 142
Morgan Stanley, 340
move, 321
MRO, 89
multimethod, 203
multiple dispatch, 169, 179
multiple inheritance, 83, 88, 131
mutable, 5

namespace, 11, 86
    Clojure, 196
narrow, 76
narrowing conversion, 127
neuromorphic system, 357
nonarrow, 49
nondaemon, 144

nondeterminacy, 140
nonstrict semantics, 211
normalization, 42
null pointer, 278
null reference, 7
nullable type, 163
nullish, 44, 58
Nygaard, Kristen, 337

object, 5, 104
object orientation, xiii
Object Pascal, xiii
object pool, 87
object-oriented paradigm, 107
Objective-C, xiii
objects, 126, 338
OCaml, **344**
offsets, 276
opaque type, 280
operator overloading, 31
operator sections, 218
option chaining, 299
option values, 10
optional chaining, 44
override, 295
own properties, 50, 58
ownership, 315

package, 10, 127
panic
    Go, 255
parallel, 139
parallel programming, 183
parameterized types, 133
parameters, 24, 46
parametric, 8
parametric methods, 180
parametric types, 169, 175
ParaSail, **348**
pass-by-sharing, 92, 349
path copying, 198
pattern, 18, 47
pattern matching, 224
PCRE, 343
Perl, xiii, 343
persistent, 198
persistent data structures, 14, 189
PHP, xiii, **351**
PL/I, xiii
platform thread, 140
platform threads, 10
pointer, xiv, 276
pointers, 246
positional, 93
postfix, 9, 342
PostScript, 12, 342
precedence, 9
precomposed, 291
predeclared identifier, 254
preempt, 10
preempted, 10, 140
preemptive multitasking, 34
prefix, 9
preprocessor, 284
primitive types, 126
private, 54
Private methods, 116
procedural macros, 332
procedure, 10
process, 10
product types, 176
programming paradigm, 107
Prolog, xv, **339**
promise, 56
propagated, 25
protected call, 25
Protected methods, 116
protected object, 349
protocol, 202, 289, 301
prototype, 49, 341
prototype chain, 89
public methods, 116
pure, 211
pure function, 211
pure functions, 228
puzzle languages, 13
Python, **83**

qualified names, 126
queries, 340
quoted, 180

R, xiii
range, 84
raw type, 296
raw values, 296
receiver, 49, 110
recursion point, 206
reduce function, 2
reduction machine, 357

reference, 5
reference count, 307
reference cycle, 309
reference semantics, 292
reference types, 126
referent, 6
refs, 200
register, 357
remote call, 183
remote reference, 183
rendezvous, 349
REPL, xvii
rest parameter, 46
result values, 10
routine, 10
Ruby, **103**
runes, 240
Rust, **315**

safe language, 305
saturate, 7
Scala, xiii
Scheme, 11
scope, 26
scripting language, 342
Self, xiii
sequence, 4, 195
settles, 57
sh, 343
shadow, 26, 91
shallow binding, 27
shallow copy, 29
shared resource, 140
Shirky, Clay, 12
short variable declaration, 239
shorthand argument names, 294
side-causes, 228
side-effects, 9, 228
SIMD, 357
Simula, xiii, **337**
singleton class, 110
slice, 40, 177, 240, 250
slice literals, 250
Smalltalk, xv, 337, **337**
smart pointers, 321, 322, 346
software crisis, xiii
software transactional memory, 200
sparks, 232
special forms, 191
splicing, 180
spread, 43
spread operator, 46
SR, 347
stack, 248
    Factor, 342
stack-based, 342
Standard ML, **344**
state, 108
statement, 9
static storage duration, 278
static type, 8
static types, 65
statically scoped, 27
storage duration, 278
stream, 138
strict, 220
strict evaluation, 220
string, 4, 8
string literal, 282
string terminator, 270, 282
strong references, 309
strong typing, 24
strongly typed, 8
structured programming, xiii, 245
subclass, 88
subroutines, 10
sum types, 176
superclass, 88, 105
suspendable function, 165
Swift, **289**
switch statement, 296
symbol, 4, 106
synchronous, 144
synchronous queue, 145
syntactic sugar, 53
syntax quote, 205
system software, 345

table, 15, 28
tagged union, 224
tainted, 212
task, 142, 310
template literal types, 79
temporal dead zone, 55
terminal operation, 138
ternary, 9
thread, 34
thread pool, 142
threads, 10, 140
throwing functions, 306

TMTOWTDI, 343
trailing closure, 156
trait, 329
trait objects, 330
transaction, 200
transient data structures, 198
transients, 189
TrueNorth, 357
truthy, 46, 58
tuple, 85
turbofish, 318
tvOS, 289
type, 7
type alias, 67
type annotations, 64
type constraints, 8
type erasure, 137, 150
type extension, 202
type guard, 76
type hint, 91, 210
type inference, 294
type parameter, 8, 133
type predicate, 77
type property, 303
type signature, 215
type variables, 221
typeclass, 8, 211, 222
types, 360
TypeScript, **63**

unary, 9
unbounded wildcards, 137
unboxing, 133
Uncle Bob, 356
uncurried, 215
undefined behavior, 278
Unicode scalars, 21
unify, 221
unit type, 214
unowned references, 309
unquote splicing, 206
upper bounded wildcard, 136
utility types, 79

Valim, José, 347
value semantics, 292
values, 4, 360
van Rossum, Guido, 83
var
    Clojure, 196
variable, 5
vector, 177
verb, 341
Victor, Bret, 2, 93
virtual thread, 141
visionOS, 289
von Neumann, John, 357

wait group, 261
wall clock time, 232
walrus operator, 90
watchOS, 289
weak reference, 309
weak typing, 8, 24, 39, 46
Whitney, Arthur, 340
widening conversion, 127
wild pointer, 278
wildcards, 136

yield, 33, 96, 100

zero cost abstractions, 321
zero value, 241