Ray Toal, Loyola Marymount University
Rachel Rivera, Pandora
Alex Schneider, Security Innovation

Programming Language Explorations

Contents

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: Industry learns that hard-to-read code prevents many projects from ever completing. The *structured programming* revolution begins. Languages providing *information hiding*, such as Modula, CLU, Mesa, Turing, and Euclid attempt to address the "software crisis." Structured and modular features are bolted on to earlier languages.
- 1980s: *Object-orientation* (OO) takes over the world. Though begun in the 1960s with Simula and refined in the 1970s at Xerox PARC with Smalltalk, OO—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: We now have the World Wide Web 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, and Swift challenge C and C++ for native applications.

Each of the thirty-two 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 solutions in a more efficient manner. Some introduce big ideas that give us new ways to *think about* programming. 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 in use today. For each, we'll provide a short but example-filled tour of what makes the language unique, what it does well, and what it does not do well. In doing so, we'll introduce many *fundamental concepts* transcending multiple languages, providing you a foundation for more effectively using your current favorite languages—and for learning new ones too.

ORGANIZATION

After the obligatory introductory chapter outlining the book's goals and objectives, we present 15 chapters covering 15 languages. We introduce each with a common trio of example programs, take a brief tour of its basic elements, and describe its type systems, functional forms, scoping rules, concurrency patterns, and sometimes, metaprogramming facilities.

We've carefully chosen the order of languages to make a story out the book. We start with JavaScript because it is extremely popular and full of interesting features, including first-class functions and prototypes. It has influenced hundreds of successors, of which our second language, CoffeeScript, may be the most popular.

The next three languages, Lua, Python, and Ruby, are general-purpose scripting languages. They are followed by Julia, whose treatment of object-oriented programming via generic functions and multimethods is a nice contrast to Ruby's classes.

We then take a quick look at Java, a popular language in the enterprise computing space, and our first statically-typed language. Java was introduced along with the *Java platform*, which includes a virtual machine and a very extensive set of powerful libraries for just about every computing task imaginable. Hundreds of languages are targeted to this platform. We cover two of the most popular: Scala and Clojure. Next up is Erlang, a language designed for building large-scale, highly reliable systems, and, like Clojure, is a dynamic functional language. We return to static typing with Elm, where we'll encounter a very powerful type inference mechanism.

Our last four languages are all designed to produce native applications. First we see C++, a near-superset of the popular C language, famous for constructs (such as destructors) that simplify memory management. Rust strives to be C++ without the memory leaks and dangling pointers; these errors are elimi-

nated by fascinating concepts such as ownership and borrowing of references. Go compiles to the metal while still supporting garbage collection. We close with Swift, which like its predecessor, Objective-C, avoids both completely manual memory management and garbage collection, instead managing memory with a technique known as ARC.

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.—do not appear. Our goal is to focus on languages that are used today. After all, the influencers have modern descendants: Clojure is a modern Lisp; Ruby borrows much from Smalltalk; Elm is the latest of the ML family. Fans of Haskell will find much to admire in Elm. C++, while over 30 years old, has been modernized over the years: today we use C++14. Recent languages, such as Julia, Rust, and Swift, collect many of the best ideas of the last fifty years; we've selected them as a way to illustrate the breadth of the field.

Notice that we just mentioned *ideas* rather than *features*. Alan Kay [20] 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/1, 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 styles in addition to features. And while our main focus is on languages of today, many of the great classics, along with a few other modern languages, are discussed in appendices. We've even added appendices on a few intermediate and assembly languages.

AUDIENCE

This book is targeted both to professionals looking to expand the range of languages and programming patterns they can apply in their work, and to advanced college undergraduates, perhaps as a supplement to a programming languages or software engineering course. We've paid special attention to modern programming practice, covered some cutting-edge languages and patterns, and provided a number of runnable examples, so developers will find new skills to apply in their craft. In addition, we've slipped in a bit of academic terminology, and at times real computer science, because foundational principles are required to take the leap from a hobbyist programmer to a professional

able to build large, scalable, efficient systems. However, the book remains unapologetically an exploration of high-level language practice, rather than a theoretical treatise based on formal syntax or semantics.

PREREQUISITIES

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 more tour than tutorial. We will be covering many interesting (and often powerful) features, and will do so with code that is often very dense, sometimes cryptic, even when presenting unfamiliar paradigms for the first time.

You should also know your way around your file system and be skilled in using a command line interface to create and delete files, manage directories, and launch programs. You should also be pretty good with a text editor. Finally, you should be able to find and install software as needed to build and run the examples, and use a REPL or playground to practice. We're not going to tell you how or where to run our examples, since this information is a short web search away (and subject to change frequently).

SUPPLEMENTARY MATERIAL

All of the code in the book can be found at `https://github.com/rtoal/polyglot`. Errata, additional exercises, and bonus material can be found at `http://rtoal.github.io/polyglot/`.

ACKNOWLEDGEMENTS

We'd like to thank Loren Abrams, Facebook; David Pedowitz, Friendbuy; Andy Won, Amazon; Saturnino Garcia, University of San Diego; Caskey Dickson, Google; B.J. Johnson, Claremont Graduate University; Matt Brown, UCLA; and Craig Reinhardt, California Lutheran University for their careful readings of early drafts and many constructive comments. Zane Kansil, Zoey Ho, Juan Carrillo, Stephen Smith, Andrew Akers, and Ed Bramanti also reviewed the text and wrote much of the sample code. We are also grateful to the staff at Taylor & Francis, including Randi Cohen, Senior Acquisitions Editor, and Melisa Sedler, Project Coordinator, without whose hard work this book would not have been possible.

CHAPTER 0

Introduction

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 to convince you in this chapter that programming languages are worth studying, and we'll give you tips for how to get the most out of your study.

0.1 WHY STUDY PROGRAMMING LANGUAGES

Learning new programming languages will enable you to think about, and solve, problems 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? In Swift, we can use the array *elements* directly:[1]

[1] Yes, there are fancier ways of doing this problem in Swift, but we're moving incrementally for now.

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

Still unsatisfying. The `for`-loop and its nested `if`-statement somewhat obscure the fact we are operating on an entire array, and gives the code a one-element-at-a-time feel. We could instead:

1. *Select* the even elements from the array,
2. *Map* the square operation over the selected elements, then finally
3. *Reduce* 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 (using the term *filter* for selection):

```
(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(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 lets you express a mapped and filtered sequence directly with a *generator expression*:

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

Concise `for`-loop alternatives barely scratch the surface of the things you may learn. Consider assignment. What happens after executing `x = y + z`? The variable x gets updated on the spot 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 isn't just assigned to once, but will automatically update whenever y and z are subsequently changed. There are even languages that prohibit assignment altogether!

You'll also encounter lists that aren't physically stored but produce their elements on demand. You'll encounter models of the world where every piece of a 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. You will even find languages in which you never code step-by-step algorithms; instead, you simply state the properties you expect of a solution, and let a built-in inference engine find the solution for you.

Exposure to these novel ways of thinking about and solving problems in new languages can help you better write code in the languages you *do* use every day, as you discover how to simulate features that your language lacks. As you learn the names of these features, you'll begin to understand obscure error messages from your compiler. You may even gain a sense of performance implications of doing things one way over another—no matter what language you will encounter.

Understanding language-independent concepts allows you to learn new languages more easily, and even design your own language. Although making a splash with a complete general purpose language you design yourself may bring fame and fortune, remember that "design" also applies at small scales: you can create little languages to control robots or paintbrushes, invoke commands in an adventure game, describe formulas to input into a calculator or spreadsheet, or specify questions to ask of a search engine or database.

There's more: learning many programming languages may actually make you feel smarter, as you find yourself casually discussing the finer points of pattern matching, type inference, closures, prototypes, introspection, instrumentation, just-in-time compilation, annotations, decorators, memoization, traits, streams, monads, actors, mailboxes, comprehensions, continuations, wildcards, promises, regular expressions, proxies, and transactional memory.

Steve McConnell articulates these benefits as well as anyone [27]:

> ...Mastering more than one language is often a watershed in the career of a professional programmer. Once a programmer realizes that programming principles transcend the syntax of any specific language, the doors swing open to knowledge that truly makes a difference in quality and productivity.

0.2 HOW TO STUDY PROGRAMMING LANGUAGES

In order to become proficient in multiple languages (Smalltalk, Ruby, Python, C#, JavaScript, Go, Lua, Fortran, Io, Java, ML, and Elixir, to name a few) you must learn fundamental language concepts (binding, scope, substitution, sequencing, conditional execution, iteration, recursion, functional decomposition, modularity, synchronization, and metaprogramming, again to name a few). Similarly, learning multiple languages enables you to better master these concepts. Both directions of influence are equally important. In this book, we've chosen to proceed language-by-language rather than concept-by-concept. As we present each language, we'll introduce concepts as needed, and take the time to compare and contrast with languages we've seen before.

Keep in mind that a language is more than a list of its features. 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* [2], Melissa Elliott's *PHP Manual Masterpieces* [10], and Eric Wastl's *PHP Sadness* [39]. But most languages 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
Algol 68	General-purpose computation
Pascal	Teaching structured programming
Modula	Modular programming
C	Systems programming
SQL	Database applications
Scheme	To be a simpler alternative to Lisp
Prolog	Expert systems; Natural language processing
Smalltalk	Personal computing
MATLAB	To be an alternative Fortran for numeric computation
Ada	Megaprogramming; Embedded systems
C++	Simulation
ML	Theorem proving
Perl	Scripts, with a focus on text processing
Erlang	Massively scalable, available, soft real-time systems
Python	Scripting, with an extensible language
Ruby	To be the language Matz[2] wanted, and to be better than Perl
Lua	General-purpose scripting

[2]Yukihiro Matsumoto, the designer of Ruby

Java	Compact downloadable executable components
C#	Enterprise computing on the .NET platform
JavaScript	Client-side scripting for web applications
PHP	Server-side scripting for web applications
Postscript	Formatting of printed documents
XSLT	Hierarchical document transformation
Haskell	To bring together a number of existing functional languages
Io	Dynamic language experimentation
Scala	To support both functional and object oriented programming
F#	To solve complex problems in a simple way
Clojure	To be a modern Lisp dialect for the JVM
CoffeeScript	To be a cleaner JavaScript
Dart	Modern web applications
Go	Large distributed, interconnected systems
Rust	Large, safe, network clients and servers
Elm	Browser-based apps in a functional, reactive style
Elixir	To modernize and extend Erlang
Julia	High-performance scientific computing in a dynamic language
Hack	Web applications in a safe variant of PHP
Swift	iOS development with modern language features

It's probably a good bet that a language designed expressly for the purpose of formatting printed documents might not be good for writing web servers. So what is the point of arguing whether PostScript is better than Go? The correct retort to "*A* is better than *B*" is "For what, exactly?"

In addition to understanding the kinds of problems a language was designed to solve, we need to understand *how* a language solves these problems. We can then classify languages in a way that enables us to compare and contrast. 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*. [31] When there is no hierarchy, we employ *tags*. Here are a few tags we can use to label languages[3]:

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

[3]These are defined in the glossary in Appendix A.

A given language will generally have quite a few tags. Java, for example, was introduced to the world as meeting the following goals: simple, object-oriented, familiar, robust, secure, architecture neutral, portable, high-performance, interpreted, threaded, and dynamic. [14] Scala, by design, is both functional and object-oriented. Understanding the various tags, or categories, provides important contextual information for your language study.

Finally, it is crucial to study concepts and languages which are not "mainline." In [37], 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 instead of procedures, and are represented spatially rather than in text strings. 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.

0.3 ELEMENTS OF PROGRAMMING LANGUAGES

In order to compare and contrast programming languages in a meaningful way, we should have a working vocabulary. In this section, we'll introduce a few terms that capture some of the very basic elements that most languages must denote.

A **value** is a unit of data. We have numeric values (e.g., *three*, π, *ninety-seven point eight*), values for truth and falsity, text values, values containing other values (whose internal values may be named or numbered), and values that indicate missing or unknown information.

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

- `95` is a literal for the value *ninety-five*. So, in many languages, is `0x5F` (`0x` is a common prefix for hexadecimal numerals) 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.
- `"Hello, how are you?"` is a literal representing a common English greeting.
- `[0, true, 98.6]` is 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. Alternate forms for this function include `{%0 / 2}`, `#(/ % 2)`, `lambda x: x / 2`, and `function (x) {return x / 2}`.
- `null` (sometimes `nil` or `None`) is a literal used to indicate 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." In languages without this special literal, `null` or `nil` ambiguously stands for both (1) intentionally missing and (2) unknown information.

A **variable** is a name that refers to a value. Do not confuse variables and values: variables either *stand for* or *hold* values, they are not themselves values. In some languages, a variable is simply a name bound to a value; in others, a variable is a container into which different values can be placed at different times during program execution. Variables of the latter kind might more properly be called **assignables**. [15] Regardless of the kind of variable, please note a *value never changes* (*five* is always *five*), but *which* value is bound to or held by a variable may vary.

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; some, like `&&` ("and then") and `||` ("or else"), do not. An example of expression evaluation, assuming 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
```

A **routine** is a bundle of code, designed to run as a unit. Routines designed to run from the beginning until completion are called **subroutines**; those that can yield to other routines in the middle of execution, and be resumed where they left off, are called **coroutines**. Many languages use the term **function** in place of subroutine; others use the term only for subroutines that return a value (or values), and use **procedure** for a subroutine that returns no values.

A **type**, roughly, is 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 [29]. 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, say, `true` to the boolean type, `0.3` to some kind of numeric type, and `"Hello"` to the string type, but exactly how these types are defined can vary significantly by language. Some languages use different types for integer and non-integer numeric values. Some arrange types into supertype-subtype relationships, where a value of a subtype can be used wherever a value of a supertype is expected. You might find that some languages group all functions into a single type, while others type their functions based on the types of arguments or permitted return values (see Table 0.1).

Language	Literal	Type
Lua	`function (x) return x * x end`	`function`
JavaScript	`x => x * x`	`Object`
Elm	`\x. x * x`	`number -> number`
Python	`lambda x: x * x`	`function`
Julia	`x -> x * x`	`Function`
Scala	`(x: Int) => x * x`	`Int => Int`
Go	`func (x int) int {return x*x}`	`func(int) int`

Table 0.1 The type of the square function across different languages

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.

A **statement** is code that performs an action. Common types of statements include (1) *declaration statements* to bring variables into existence, (2) *expression statements* to evaluate expressions, (3) *assignment statements*, (4) *invocation statements* to run subroutines or start coroutines, (3) *conditional statements* (`if`, `unless`, `switch`, `match`) to execute code only under certain conditions, including perhaps the resumption of one of several coroutines based on the state of various guarded expressions, (4) *iteration statements* (`while`, `do while`, `for`, `repeat`) to execute code repeatedly, and (5) *disruptive statements* (`break`, `continue`, `retry`, `throw`, `return`, `yield`, `resume`) for changing the normal control flow when needed.

Finally, many languages provide a means for *throwing* or *raising* an **exception** when something has gone wrong, and a means to *catch*, or *handle*, thrown exceptions. Languages without an exception facility will often have a means to allow certain operations to return two values, one indicating success or failure, and the other for the expected value on success.

0.4 EVALUATION OF PROGRAMMING LANGUAGES

We haven't mentioned one of the most important advantages of studying multiple programming languages: being able to choose the most appropriate language for a given task. But what makes a language the most suitable for a given task? On what criteria might we evaluate languages?

We may, for example, 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**. Few people are willing to undertake a huge learning curve for "difficult" languages.
- **Expressiveness**. Isn't it lovely when `A=B+C` just works—when A, B, and C are ... matrices? Or when we can avoid writing type expressions because the compiler infers the types for us? Or when we have high-level constructs to make concurrent programming safe without us having to remember to acquire and release locks? Or when we can simply express *what* we want and have the system figure out *how* to get it for us?
- **Guidance**. It can be very frustrating when a programmer uses a value of the wrong type, or 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.
- **Efficient compilation, efficient execution, or both**. Who wants to sit around waiting for the code to compile,[4] or run?

But we may end up choosing a language for other, "softer," reasons:

- It might be the only language suitable for a specific problem.
- You just like it. We all have our own preferences. As they say, "there's no accounting for taste."
- You need to hire enough people that already know it. Face it, some languages are very popular; some are not.
- There is a wealth of development tools (IDEs, fast compilers), or resources (books, blog posts, activity on programming Q&A sites) for it.
- A big government, or a big company, created or backed it (IBM created PL/I, Sun–Java, Google–Go, Facebook–Hack, and Apple–Swift).
- Everyone else is using it. There's an old saying that goes: "No manager ever got fired for choosing Java."
- The boss made you use it.

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

- 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.

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 informed 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, polymorphic type systems, functions as first-class values, higher-order functions, and closures (we'll talk about these later) saves development time but often leads to slower-running code.
- 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. A garbage collector might decide to free up space while your code is performing a time-sensitive complicated maneuver in a fighter plane 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 to have so many choices. The next fifteen chapters will take you through fifteen modern languages and the choices made by their designers. We'll see design tradeoffs time and time again. Have fun.

CHAPTER 1

JavaScript

We'll begin our tour with JavaScript, because it is both simple and sophisticated. It is also, by some measures, one of the most popular programming languages in the world.

First appeared	1995
Creator	Brendan Eich
Notable versions	ES3 (1999) • ES5 (2009) • ES2015 (2015)
Recognized for	First-class functions, Weak typing, Prototypes
Notable uses	Web application clients, Asynchronous servers
Six words or less	"The assembly language of the web"

JavaScript was designed and implemented in ten days in 1995 by Brendan Eich, then at Netscape Communications Corporation, with the goal of creating an amateur-friendly scripting language embedded into a web browser. The syntax of the language was strongly influenced by C, with curly braces, assignment statements, and the ubiquitous `if`, `while`, and `for` statements. Semantically, however, JavaScript and C are worlds apart. JavaScript's influence here was Scheme. Functions are *first-class values*: they can be assigned to variables, passed to functions, and returned from functions.

The goal of allowing novice programmers to write small scripts in web page markup led to some well-loved design choices, including array (e.g., `[10, 20, 30]`) and object (e.g., `{x:3, y:5}`) literals. Yet the attempt to keep the language simple led to several notorious features as well. **Weak typing**, where expressions of the wrong type are automatically coerced to "something that works," and **automatic semicolon insertion**, where the language will figure out where your statements begin and end when you are not explicit, save typing but can result in utterly surprising behavior. Douglas Crockford [7] has catalogued these and a number of other "Bad Parts" and

"Awful Parts," while at the same time praising JavaScript as "[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." [7, p. 2].

Though originally targeted to beginners, the language has grown a great deal over time and has been used in thousands of successful, and sophisticated, applications. Every major web browser comes with a JavaScript engine. Running applications written in languages other than JavaScript in a browser is done by first translating, or compiling, to JavaScript. For example, C and C++ applications can be compiled into the intermediate language LLVM (see Appendix A.3) then translated into JavaScript by the compiler *emscripten* [11]. But JavaScript's success is not limited to the browser; the language is used on powerful servers running applications supporting thousands of concurrent users.

As the first language on our tour, we'll use JavaScript to introduce several aspects of functions: argument passing, scope, the handling of free variables, closures, and anonymous and higher-order functions. We'll show how prototypes are used to create sets of similar objects, and look at one of JavaScript's more interesting features: the all-purpose `this`-expression, which allows code to behave differently in different contexts. We'll continue with a brief look at how large programs are built with modules. We'll close the chapter with a short overview of asynchronous programming, one of many approaches to concurrent programming we'll encounter in this book.

1.1 HELLO JAVASCRIPT

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

```
for (let c = 1; c <= 50; c++) {
  for (let b = 1; b < c; b++) {
    for (let a = 1; a < b; a++) {
      if (a * a + b * b == c * c) {
        console.log('%d, %d, %d', a, b, c);
      }
    }
  }
}
```

We're terribly sorry for the flashback to high school math, but this code works pretty well as an opener, and is certainly more interesting than "Hello, world." It lists all of the right-triangle measurements with integer values up to size 50 (see Figure 1.1 for examples). You can run it directly in your browser's developer console, in an online JavaScript Runner, or with Node.js. [19] Your output should be:

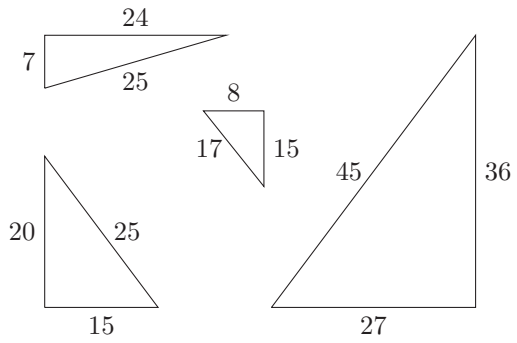


Figure 1.1 Integer right-triangle lengths

```
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
9, 40, 41
27, 36, 45
30, 40, 50
14, 48, 50
```

As mentioned in the preface of this book, we're expecting you to have programming experience, so you can probably figure out the for-loops and the if-statement. Note the spelling of the equality operator: `x === y` is true iff x and y have the same value and the same type. We don't use JavaScript's `==` operator, as it only computes whether two objects are *similar* to, rather than equal to, each other, and sometimes behaves unexpectedly (e.g., `null == undefined` and `false == " \t "`). The `log` method of the `console` object writes a line of formatted text to standard output; common format specifiers include `%d` for numbers, `%s` for strings and `%j` for JSON data. [18]

Our second program accepts a **command line argument**, writing its permutations (possible orderings of characters) to standard output:

```
function swap(a, i, j) {
  [a[i],a[j]] = [a[j],a[i]]
}

function generatePermutations(a, n) {
  if (n === 0) {
    console.log(a.join(''));
  } else {
    for (let i = 0; i <= n; i++) {
      generatePermutations(a, n - 1);
      swap(a, n % 2 === 0 ? i : 0, n);
    }
  }
}

if (process.argv.length !== 3) {
  console.error('Exactly one argument is required');
  process.exit(1);
}
let word = process.argv[2];
generatePermutations(word.split(''), word.length - 1);
```

Let's run this script with Node.js. Store your script in the file *anagrams.js* and enter the following on the command line:

```
node anagrams.js rat
```

Your output should be:

```
rat
art
tar
atr
tra
rta
```

The program first checks that there is exactly one command line argument, and if not, terminates with an error message (written to standard error, *not* standard output) and non-zero exit code. In Node, `process.argv` is an array of *all* the tokens entered on the command line. We check for a value of 3, because the first two tokens specify the script to run:

```
process.argv[0] === 'node'
process.argv[1] === 'anagrams.js'
process.argv[2] === 'rat'
```

The script then calls the recursive `generatePermutations` function of two arguments. We won't describe how the algorithm works; it's *Heap's algorithm*, which you can look up on Wikipedia. We will, however, note the script's use of `split` and `join`. JavaScript strings are **immutable**: you cannot make a string longer or shorter or change any of its constituent characters. Implementing this algorithm requires us to split the string's characters into an *array*, which *is* mutable. We repeatedly rearrange the array in place, each time applying the `join` operation to get a new string for output.

Let's move to to a more complex example. We'll read text from standard input and produce a report with the number of times each word appears:

```
import split from 'split';
import {XRegExp} from 'xregexp';

let counts = Object.create(null);

process.stdin.setEncoding('utf8');

process.stdin.pipe(split()).on('data', line => {
  let wordPattern = XRegExp("[\\p{L}']+", 'g');
  (line.toLowerCase().match(wordPattern) || []).forEach(word =>
    counts[word] = (counts[word] || 0) + 1
  );
});

process.stdin.on('end', () =>
  Object.keys(counts).sort().forEach(word =>
    console.log('%s %d', word, counts[word])
  )
);
```

We'll use Node.js to run this script, too. Our script requires functionality from two external modules; you'll need to fetch these from an online repository.[1] For fun, download the plain text version of *War and Peace* from Project Gutenberg (`http://www.gutenberg.org/cache/epub/2600/pg2600.txt`) and call the file *warandpeace.txt*. Store the script in *wordcount.js* and run:

```
node wordcount.js < warandpeace.txt
```

This script introduces too many JavaScript features to explain in detail at the moment, but let's look at some highlights before we begin our language overview in earnest in the next section.

1. We bind variables `split` and `XRegExp` to values from external modules. We're importing the *default* value from the `split` module, so its import statement lacks braces.

[1] Run `npm install split; npm install xregexp` in the folder you are using for your source code.

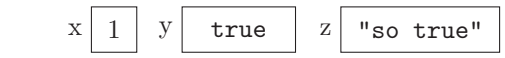
2. We initialize an empty **object** that we will fill with words and their counts. JavaScript objects map properties (in this script, our words) to values (in this case, each word's count).
3. We tell the standard input reader, `process.stdin`, how it should interpret the bytes that it reads. UTF-8 is explained in Appendix D.
4. We enhance the standard input reader to emit a line at a time. We arrange that whenever a line is ready (i.e., the reader has fired a `data` event), we will lowercase it, break it up into words, and increment the count for each word. The `xregexp` module allows us to define patterns that can be matched; here we state that words are sequences of one or more (`+`) letters (`\p{L}`) or apostrophes (`'`).[2]
5. We arrange that when we get the signal that our input has been fully read, we will write out each word, and its count, in sorted order.

1.2 THE BASICS

A JavaScript program is made up of modules (covered in Section 1.6), each containing a sequence of statements and function declarations. Statements include variable declarations, assignment and function calls, and simple conditionals and loops, among others. Values have one of exactly 7 types:

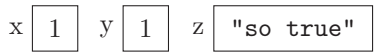
- The type containing the sole value `undefined`.
- The type containing the sole value `null`.
- **Boolean**, containing the two values `true` and `false`.
- **Number**, the type of all numbers, including -98.88, 22.7×10^{100}, `Infinity`, `-Infinity`, and, strangely enough, `NaN`, the number meaning "not a number."
- **String**, roughly, the type of character sequences, but technically the type of sequences of UTF-16 code points (details in Appendix D). String literals are delimited by either single or double quotes.
- **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.[3]

The first six types are **primitive types**; Object is a **reference type**. The difference is best explained by looking at how variables work. Think of variables as boxes containing values. The *declarations* `let x = 1; let y = true; let z = 'so ' + y` create variables with initial values as follows:



[2]Complete details regarding patterns are beyond the scope of this text.
[3]The properties of an array include `0`, `1`, `2`, and so on, and `length`.

After a variable is declared, an **assignment** puts a new value in its box; for example, `y = x` will *copy* the value currently in `x` into `y` (overwriting the contents of `y`):



Values of a primitive type are written directly inside the variable boxes, but object values are actually *pointers*, or *references* to entities holding the object properties. This is best explained by analyzing the following script, and its visualization in Figure 1.2:

```
let a = {x: 3, y: 5};  // creates an object
let b = a.y;           // simply puts 5 into b
let c = null;          // simply puts null into c
let d = {x: 3, y: 5};  // creates an object
let e = d;             // does not create an object
```

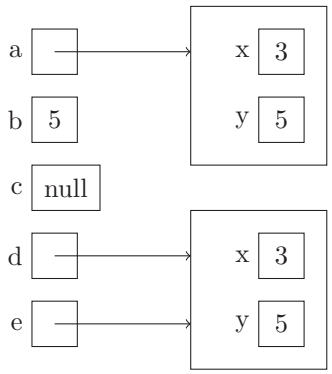


Figure 1.2 JavaScript Primitives and References

References allow multiple variables to refer to the same object. In our figure, `d.x` and `e.x` are **aliases** of each other—an assignment to one updates the other. References lead to multiple interpretations of what it means to copy: when nested objects exist, will copying only the references suffice (a **shallow copy**), or must *all* of the data be copied (a **deep copy**)? To perform a shallow copy, iterate through the properties of an object and assign their values to properties in the copy, or, for arrays, use JavaScript's `slice` method (see Figure 1.3):

```
let a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

let b = a;           // copies the reference, nothing more
let c = a.slice();   // makes a SHALLOW COPY of array elements
```

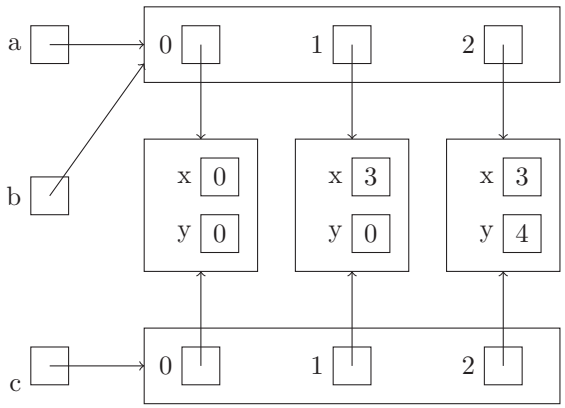


Figure 1.3 A Shallow Copy

To make a deep copy, iterate through the components of an object, copying primitives and recursively creating deep copies of objects.

JavaScript is a **weakly-typed language**, because more often than not, values of one type can appear where values of other types are expected.[4] For example:

- 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, primitives act as you would expect: `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" (e.g., `"3.5"` acts as `3.5`) or `NaN` if the string does not look like a number. To use an object x in a numeric context, JavaScript evaluates x`.valueOf()`.

1.3 FUNCTIONS

We all know what functions are (abstractions for expressions), how they work (you *call* them, passing **arguments** to **parameters**) and why we need them (code reuse). However, when we think more deeply about how functions must

[4]Contrast this with a strongly-typed language, in which using values of the wrong type more often than not generates an error.

work, many questions arise. Are arguments evaluated before the call or during the execution of the function? What if too many arguments are passed? Too few? Can a function access or modify variables in the caller's context? These questions don't even scratch the surface of what we'd like to know, but they provide a starting point for the study of JavaScript functions.

1.3.1 Frames

Let's begin with the answers to the above questions: 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 parameters without arguments start off as `undefined`. Each time a function is called, a brand new **frame**, or **activation record**, is allocated to hold parameters and **local variables** for this call only. Local variables are those **declared** with `var` or `let` inside the function. Variables used but not declared inside the function are called **free variables**. Local variables and parameters cannot be seen from outside the function.

Because parameters are always freshly allocated at call time, they are distinct variables from the arguments, so changing a parameter will not affect its corresponding argument. But keep in mind that since object values are just references, you can change the *properties* of a passed object through the parameter. Let's see how this works with a concrete example:[5]

```
import assert from 'assert';

let g = 100;                        // Note declared outside of f
let x = [1,2,3];                    // Will pass this to a
let y = [4,5,6];                    // Will pass this to b

function f(a, b) {
  assert(g === 100);                // We can see enclosing scope
  g = 200;                          // Change var in enclosing scope
  assert(g === 200);                // See it changed
  a = 300;                          // Change a parameter
  assert.deepEqual(x, [1,2,3]);     // But argument still intact!
  b[1] = 400;                       // Change property
  assert.deepEqual(y, [4,400,6]); // See the change!
}

f(x, y);
```

[5]Throughout this book, we'll be illustrating concepts using assertions. An assertion either silently succeeds or generates an error, and allows for both convenient documentation and automated testing. You can find information about Node's built-in `assert` module in [**?**].

For a visual take, we'll revisit the permutations script from Section 1.1. Consider running the script with argument `rat`. The script calls `generatePermutations(['r','a','t'], 2)`, generating a new frame with local variable `i`. This activation makes a call to `generatePermutations(a,1)`, which calls `generatePermutations(a,0)`. This activation outputs `"rat"`, then returns to the previous activation, where its `i` is incremented and `swap` is called. Figure 1.4 shows a snapshot of program execution at this point. Because each frame has its own set of local variables, we are able to swap characters at the proper times. Because only a reference to the array is passed each time, we save the overhead of making copies of (potentially large) objects.

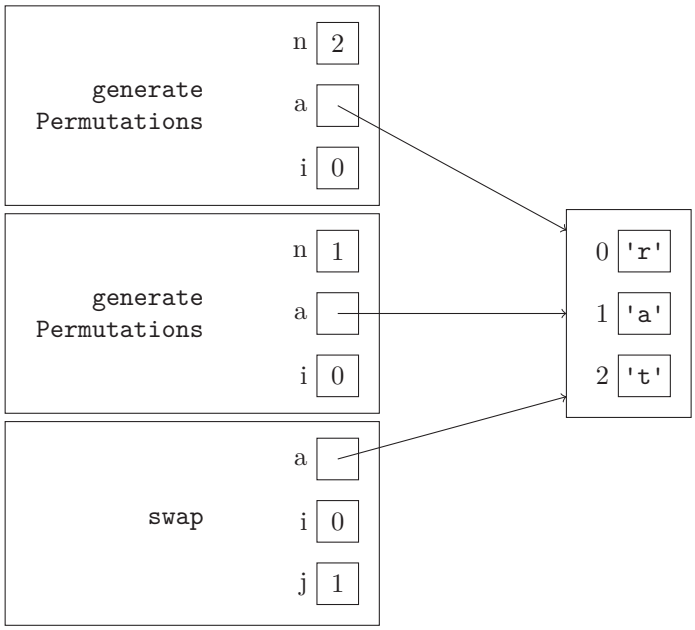


Figure 1.4 Functional Call Execution Snapshot

Variables declared outside of any function or module (see Section 1.6) are called **global variables** and are implemented in JavaScript as properties of the **global object**. Because they are properties, they really aren't variables at all: they live inside an object, not a frame! Initially, the global object properties include `isFinite`, `String`, `Object`, and `Date`, as well as a property that references the global object itself, generally called `global` or `window`, depending on the environment. The programmer can create additional global variables with `var`- or `let`-declarations outside of any function or module. Interestingly, using an undeclared variable throws a `ReferenceError`, while referencing a missing property produces `undefined`.

```
import assert from 'assert';

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

1.3.2 Anonymous and Higher-Order Functions

JavaScript functions are objects, so they can be assigned to variables, passed to functions, and returned from functions. They may or may not have **names**; functions without names are called **anonymous functions**. The following script shows functions in action:

```
import assert from 'assert';

// We don't have to name functions to call them
assert((x => x + 5)(10) === 15);

// We can assign function values to variables
let square = x => x * x;
let odd = x => Math.abs(x % 2) === 1;
let lessThanTen = x => x < 10;
let twice = (f, x) => f(f(x));

// We can pass function values
assert(twice(square, -3) === 81);
assert(twice(x => x + 1, 5) === 7);

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

// Several built-in array functions accept functions
let 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]);
```

Functions accepting a function as a parameter, or returning a function, such as `twice` and `compose` above, are called **higher-order functions**. Note the

behavior of the higher-order array functions at the end of the script (`every`, `some`, `filter`, and `map`). Do you see how these free you from having to write explicit loops?

1.3.3 Closures

All of the functions we have seen so far operate only on parameters and local variables, so let's see how JavaScript handles free variables.

```
let x = 'GLOBAL';
function second() {console.log(x)}
function first() {let x = 'FIRST'; second();};
first();
```

The variable x within `second` is free. Does x take on the value from its caller, function `first`? Or from the global x? Languages that use caller's values for free variables are **dynamically scoped**; those that look outward to textually enclosing scopes are **statically scoped**. JavaScript is statically scoped.

Things become even more interesting when functions are nested inside other functions. Consider this script:

```
function second(f) {
  let name = 'new';
  f();
}

function first() {
  let name = 'old';
  let printName = () => console.log(name);
  second(printName);
}

first();
```

The function assigned to `printName` in `first` has a free variable; it is passed to `second` (as f) and then executed. What gets logged? 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 JavaScript 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, so the function, together with its bindings, is called a **lexical closure**, or **closure** for short.

Closures are commonly used to make **generators**. A generator is a function that produces a "next" value each time it is called. For example, a generator for a sequence of squares would produce 0 on its first call, 1 on its second, then 4, then 9, then 16, and so on. While we could write a simple `nextSquare` function to increment a global variable then return its square, such code would be **insecure** because other parts of the code could change the global variable and disrupt all future calls to the generator! Fortunately, we can use the fact that a function's local variables are completely hidden from the outside:

```
let nextSquare = () => {
  let previous = -1;
  return () => {
    previous++;
    return previous * previous;
  }
}();

import assert from 'assert';
assert(nextSquare() === 0);
assert(nextSquare() === 1);
assert(nextSquare() === 4);
```

The value assigned to `nextSquare` is the result of calling an anonymous function; we call this value a **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 (`nextSquare`) can see this value, but no other parts of the code can. The generator is secure.

1.3.4 Methods

An object can have properties whose values are functions:

```
let 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}
};

import assert from 'assert';
assert(circle.area() === 100 * Math.PI);
circle.expand(2);
assert(circle.circumference() === 40 * Math.PI);
```

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**.[6] Note that we've used the alternate syntax for function values: if we define the method value with the `function ( params ) { body }` syntax, the special expression `this` refers to the receiver. Functions defined with `( params ) => { body }` do not get a special `this`. You may wish to adopt the convention of using the `function` syntax for methods and the arrow notation in all other cases. If you prefer, there's a shorthand notation for the `function` syntax inside of object literals:

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

The purpose of the special `this` expression is to allow *context-dependent code*. In the case of methods, `this` takes on the value of the method's receiver as determined at runtime. For example, if we copy a method defined in object A to object B and call the method through B, `this` will be B, not A. This **late binding** can be quite flexible, as we'll see in the next section. Before moving on, though, let's note that we can *force* the value of `this` to take on the value of our choosing via `call`, `apply`, and `bind`:

```
function talkTo(message, suffix) {
  console.log(message + ', ' + this.name + suffix);
}

let alice = {name: 'Alice', address: talkTo};
let bob = {name: 'Bob'};

alice.address('Hello', '.');               // Hello, Alice.
alice.address.call(bob, 'Yo', '!');        // Yo, Bob!
alice.address.apply(bob, ['Bye', '...']);  // Bye, Bob...
alice.address.bind(bob)('Right', '?');     // Right, Bob?

bob.greet = alice.address;
bob.greet('Oh', ' :(');                    // Oh, Bob :(
```

[6] It is probably more correct to say that `area` is a *message* sent to the receiver; the actual *method* is the body of the `area` function.

1.4 PROTOTYPES

Let's turn now from functions to objects. How do you efficiently create a bunch of similar objects? How would you define dozens, thousands, or millions of points, or of circles, or people, or votes, or airports, or web page index entries?

In JavaScript, we start with an initial (prototypical) object, then *derive* additional objects from it. These new objects have the original object as their **prototype**. What is a prototype? When we encounter the expression $q.x$, we look for an x property in q. If found, we produce the corresponding value; if not, we'll look in q's prototype (if it has one), 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`.

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

let c1 = Object.create(unitCircle);
c1.x = 3;
c1.color = 'green';

let c2 = Object.create(unitCircle);
c2.radius = 5;

let c3 = Object.create(unitCircle);
```

The expression `Object.create(`p`)` creates a new object whose prototype is p. Our script creates a black circle of radius 1, centered at the origin, as the prototype of three other circles. Because of the way JavaScript **delegates** property lookup, we have `c1.x === 3` and `c1.color === 'green'` (obviously), as well as `c1.y === 0` and `c1.radius === 1`. 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. In particular, each circle inherits all of the methods from the prototype. Every circle computes its area and circumference the same way, so it would be wasteful to store copies of these functions in each circle.

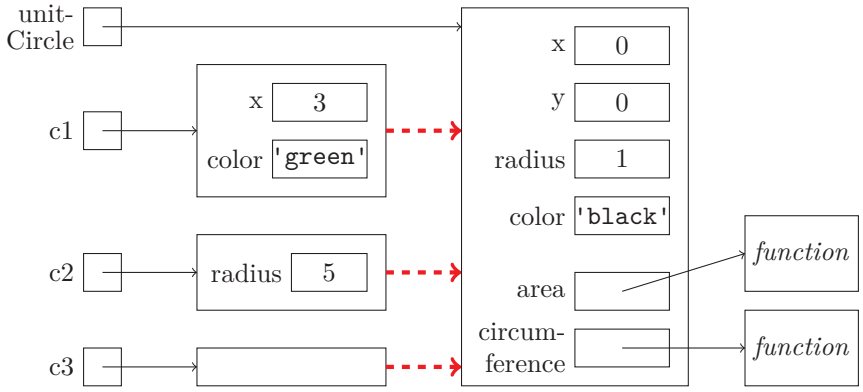


Figure 1.5 Objects sharing a prototype

Figure 1.5 shows the four circles and the prototype links to `unitCircle`. Our figure actually omits the prototype links from `unitCircle` to its prototype, and from the two functions to their prototype. You will explore these prototypes in the exercises.

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:

```
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;
};

import assert from 'assert';
let c = new Circle(1, 5);
assert.deepEqual(c, {x:1, y:5, radius:1, color:'black'})
assert(c.area() === Math.PI);
```

This simple looking script illustrates a lot of JavaScript magic. Every JavaScript function has two properties, `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. That's a lot to take in, so study Figure 1.6.

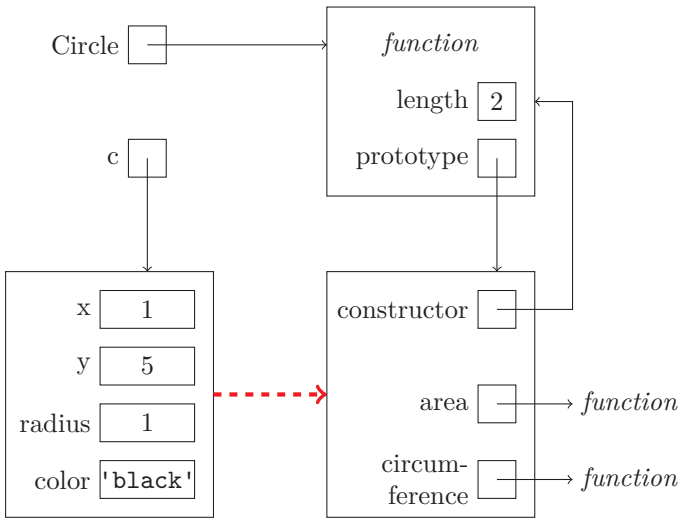


Figure 1.6 Construction of Objects using `new`

The declaration `let c = new Circle(1, 5);` creates a new object with prototype `Circle.prototype`, passes it to the function `Circle` as `this`, which fills in the properties of the new object and finally binds the object to the variable `c`. The area and circumference methods stored as properties in the prototype, just as in our earlier example. By the way, we've taken the opportunity here to introduce **default parameter values**—parameters initialized to specified values, rather than `undefined`, when no argument is supplied.

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:

```
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;}
}
```

```
import assert from 'assert'
let c = new Circle(1, 5);
assert(c.circumference() === 2 * Math.PI);
assert(typeof Circle === 'function');
```

Note, that the `class` keyword *does not* create a "class object"—`Circle` is still a function! The class notation is simply **syntactic sugar**, a syntax that makes the standard form easier to read. It does no more and no less (in this case) than defining the function and assigning methods to the prototype.

1.5 SCOPE

A **binding** is an association of a name with an entity. The **scope of a binding** is the region of code where a particular binding is active. Let's take a look at two ways we can introduce bindings in JavaScript: `let` and `var`. Bindings introduced with `var` are scoped to the innermost function, and bindings introduced with `let` are scoped to the nearest block:

```
import assert from 'assert';

let a = 1, b = 2;

function main() {
  assert(a === undefined);   // the local `a` is in scope
  assert(b === 2);           // we see the global `b`

  if (true) {
    var a = 100;             // scoped to whole function!
    let b = 200;             // scoped only inside this block
    let c = 300;             // scoped only inside this block
  }
  assert(a === 100);         // it's been initialized
  assert(b === 2);           // global, because local used `let`

  assert.throws(() => c);    // there's no `c` out here at all
}

main()
```

Reading a `var`-declared variable in its scope but before its declaration produces `undefined`, as shown above. Reading a `let`-declared variable in its scope but before the `let` throws a `ReferenceError`.

Given that there are several other ways to create bindings (`const`, function declarations, class declarations, function parameters, etc.), the complete set of rules that determine scope would fill many pages. In general, we will not be detailing the complete scoping rules for any language in this book, though we

will make time to show some of the more interesting design choices surrounding scope.

1.6 MODULES

Nontrivial applications are often partitioned into several reusable **modules**, responsible for things such as report formatting, encryption, parsing, database interaction, calendar computation, and flying drones. JavaScript's module system is pretty sophisticated, so we have time only for the briefest of examples. Let's build a little module for generating unique strings. Call this file `string_generator.js`:

```
let counters = Object.create(null);

export default function (prefix) {
  counters[prefix] = (counters[prefix] || 0) + 1;
  return prefix + counters[prefix];
}

export function countOf(prefix) {
  return counters[prefix];
}
```

This module defines three entities. The first is not exported; it is hidden, or **encapsulated** within the module. Because we have both a default and a regular export, our import syntax merits some attention:

```
import generateString, {countOf} from './string_generator.js';
import assert from 'assert';

assert(generateString('v') === 'v1');
assert(generateString('v') === 'v2');
assert(generateString('f') === 'f1');
assert(generateString('v') === 'v3');
assert(countOf('v') === 3)
```

We've specified the `string_generator` module source as a file path, beginning with `./`, since our module is a file we've written ourselves. For built-in modules (such as `assert`) or modules downloaded from a remote repository (such as `split` and `xregexp` from the beginning of the chapter), no file-system path specification is required.

1.7 ASYNCHRONOUS COMPUTATION

Programmers often have to deal with uncertainties surrounding time. Applications may have **users**—people who, without any warning, click buttons, drag fingers across a surface, press and release keys, and move cursors in and out of regions on a display. Applications may read and write files, use databases, and exchange information with programs running on different machines. These operations might take several seconds to a few minutes or more to complete. When an application stops and waits for external operations to complete, we call its behavior **synchronous**; if its stays busy doing other things, we call it **asynchronous**.

An asynchronous architecture consists of code for *firing* and *responding to* **events**. Some events originate outside the program (button clicks, a cursor entering a canvas, data becoming ready from a file read) and some from **event emitters** that you write yourself. An interactive application contains instructions to the system saying "When event e happens, call function f with the data provided by e". Let's see how this looks in a browser. The following script creates a little canvas you can sketch in:

```
(function () {
  var canvas = document.createElement('canvas');
  var ctx = canvas.getContext('2d');
  var drawing = false;
  canvas.style.border = '2px solid green';
  canvas.onmousedown = function (e) {
    drawing = true;
    ctx.moveTo(e.clientX, e.clientY);
  };
  canvas.onmousemove = function (e) {
    if (drawing) {
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    }
  };
  canvas.onmouseup = canvas.onmouseout = function () {
    drawing = false;
  };
  document.body.appendChild(canvas);
}());
```

The functions stored in the "on" properties are called by the browser whenever the desired event is emitted. The functions are called **event handlers**, or **callbacks**, and assignment to the special properties is called **registering** the handler. When an event occurs, the browser passes an **event object**, containing data about the event, to your handler. Mouse events will contain

the cursor position in the `clientX` and `clientY` properties. Touch events for phones and tablets work in a similar fashion.[7]

To see how things work on the server side, let's revisit our word count script from the beginning of the chapter:

```
import split from 'split';
import {XRegExp} from 'xregexp';

let counts = Object.create(null);

process.stdin.setEncoding('utf8');

process.stdin.pipe(split()).on('data', line => {
  let wordPattern = XRegExp("[\\p{L}']+", 'g');
  (line.toLowerCase().match(wordPattern) || []).forEach(word =>
    counts[word] = (counts[word] || 0) + 1
  );
});

process.stdin.on('end', () =>
  Object.keys(counts).sort().forEach(word =>
    console.log('%s %d', word, counts[word])
  )
);
```

Here `process.stdin` is a file **stream** representing standard input. In Node, streams are event emitters; `stdin` will fire a `data` event when data is ready to be read, and an `end` event when there is no more data to be read. We call the `on` method on the stream to register the function for the system to call when the event fires.

Node.js comes with a number of built-in modules that contain event emitters; we present a few of these in Table 1.1.

JavaScript's preference for emitters and asynchronous callbacks is one approach to managing **concurrency**. We will see others throughout the book, including threads, coroutines, actors, and functional reactive programming.

1.8 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.

[7] You can find a version of this script using touch events instead of mouse events at the companion website for this book.

Module	Emitter	Events
—	process	exit, beforeExit, uncaughtException
`stream`	Readable	readable, data, end, close, error
	Writable	drain, finish, pipe, unripe, error
	Transform	finish, end
`fs`	ReadStream	open
	WriteSteam	open
	FSWatcher	change, error
`net`	Server	listening, connection, close, error
	Socket	lookup, connect, data, end, timeout, drain, close, error
`dgram`	Socket	message, listening, close, error
`http`	Server	request, connection, close, checkContinue, connect, upgrade, clientError
	ServerResponse	close, finish
	ClientRequest	response, socket, connect, upgrade, continue

Table 1.1 A sampling of built-in event emitters in Node

- JavaScript expressions manipulate values of exactly seven types: Undefined, Null, Boolean, Number, String, Symbol, and Object. The first six are primitive types; Object is a reference type. Arrays and functions are objects. Values of reference types are actually pointers, so assignment of these values creates aliasing, or sharing.
- JavaScript is weakly-typed, meaning that in most situations a value e of type t can be used in a context in which a value of type t' is expected. The runtime will find a coercion of e to a roughly-equivalent expression e' of type t'.
- Values that coerce 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.
- 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. Parameters and local variables live only during the function activation and are invisible to outer scopes.
- When functions with free variables are passed into, or copied into, different scopes, the free variables retain continue to refer to the variables in their originally enclosing scopes. Functions taking advantage of this ability are called closures.

- JavaScript is statically scoped and uses deep binding. It supports both function-scoped (`var`) and block-scoped (`let`) entities.
- 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. It refers to (1) the global object when used in a global context, (2) the receiver of a method in a method call, provided the method is *not* defined with the fat arrow, (3) the newly created object when used with operator `new`, or (4) a object designated by the programmer in certain methods, such as `apply`, `call`, and `bind`.
- The keyword `class` provides sugar for defining a constructor function and methods that populate a prototype object.
- JavaScript's module system provides for encapsulation and both regular and default exports.
- The design of JavaScript facilitates asynchronous, event-driven programming with callbacks.

To continue your study of JavaScript beyond the introductory material of this chapter, you many wish to find and research the following:

- **Language features not covered in this chapter**. Symbols, regular expressions, property attributes, default arguments, destructuring, generators, rest parameters, spreads, template strings, typed arrays, proxies, `for-of`, and the built-in objects. Although many of these topics were not covered here, we will see them later in the context of other languages.
- **Open source projects using JavaScript**. Studying, and contributing to, open source projects is an excellent way to improve your proficiency in any language. Of the thousands of projects using JavaScript, you may enjoy jQuery (`https://github.com/jquery/jquery`), d3 (`https://github.com/mbostock/d3`), Backbone (`https://github.com/jashkenas/backbone`), and Underscore (`https://github.com/jashkenas/underscore`).
- **Reference manuals, tutorials, and books**. Strictly speaking, JavaScript is an implementation of the language ECMAScript, whose official specification is [9]. Douglas Crockford's *JavaScript: The Good Parts* [7] is a popular text covering many aspects of the language and its usage, and *NodeJS in Action* [4] will help you use the popular NodeJS and its ecosystem to build complex applications.

EXERCISES

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

1.2 If you have not already done so, install Node.js on your machine. Try out the REPL (do a web search for "node repl" if you do not know what one is). Evaluate simple expressions such as `2+2` and `true || false`.

1.3 Evaluate the following JavaScript expressions in the Node REPL: (a) `"16" == 16`, (b) `16 == "0x10"`, and (c) `"0x10" == "16"`. Were the results surprising? Why or why not? If you said no, why might a reasonable person be surprised? (Hint: Consider transitivity.) What happens when you replace the `==` operator with the `===` operator?

1.4 Learn about standard output and standard error (Wikipedia has information at [42]). Why is it so important to write error messages to standard error rather than standard input?

1.5 Why is it so important that processes produce a return code of 0 on success, and non-zero on failure?

1.6 For the first three examples in this chapter, identify their literals, variables, operators, expressions and statements. (A precise understanding of this vocabulary is essential to being able to design, analyze, and implement languages.)

1.7 Write a function to produce a deep copy of an array of points, where each point is an object with three properties (x, y, and z) with numeric values.

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

1.9 We skimped a bit in our explanations of 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.

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

```
// Rookie mistake: alerts 10 for every button.
for (var i = 0; i < 10; i++) {
  button = document.createElement("button");
  button.innerHTML = i;
  button.onclick = function () {alert(i);};
  document.body.appendChild(button);
}
```

What happens when each button is pressed? Why? Make each button alert its *own* label by assigning a one-argument IIFE to `button.onclick`.

1.11 Research the issue of static vs. dynamic scoping, and list the advantages and disadvantages of each. Do any modern languages use dynamic scoping? If so, which ones?

1.12 JavaScript 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?

1.13 `Let`-declarations in `for` loops are special: there's a new binding for *each execution* of the loop. Given this JavaScript fact, try to determine 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 you're analysis was correct.

1.14 Rewrite the IIFE for `nextSquare` on page 23 so that the local variable `previous` is a parameter, and is initialized via the argument in the invocation. Why does this alternative implementation work? Do you find it version more or less readable than the original?

1.15 It turns out that JavaScript has a more direct way of implementing generators, using the keyword `function*` and the `yield` statement. Research the topic of generator functions and rewrite the `nextSquare` example to use this mechanism.

1.16 Research the details of the `call`, `apply`, and `bind` methods that we mentioned, but did not explain, in this chapter. Explain the behavior of each in your own words.

1.17 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."

1.18 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?

1.19 Complete Figure 1.5 to show the prototypes of the unit circle object and its two methods. Show the prototypes of the prototypes where they exist.

1.20 In the example at the end of Section 1.4, we see a constructor function designed to be called with operator `new`. What happens if it is called without `new`?

1.21 Using the `lodash` module from npm, write a function that outputs a shuffled deck of (traditional) playing cards. Represent the deck as an array of 52 cards, each with one of thirteen ranks and one of four suits. (Hint: look for a method called `shuffle` in lodash.)

1.22 Write a module that exports a class representing a deck of playing cards. (You will need to look up the syntax `export default class`. Supply the following methods for a deck object: (1) Retrieve the card at a given index position (0...51), (2) Retrieve the position of a given card, (3) Shuffle the deck, and (4) Move the top card of the deck to the bottom.

1.23 (Challenge) We have included the apostrophe in the set of "word characters" in our word count example from this chapter, allowing us to correctly pick up as words the following from the *War and Peace* text file from Project Gutenberg: `who'll`, `i've`, and `zdrzhinski's`. However, the following are picked up as "words" too: `'did`, `already'`, and the lone `'`, since the text uses apostrophes for quoted text within quoted text. Do some research on the topic of **regular expressions** and rewrite the script so that the pattern picks up apostrophes as word characters *only* if they are immediately preceded and followed by a letter.

1.24 Explore the introductory JavaScript course at Khan Academy. [21] Create and share an animation of your own design.

CHAPTER 2

CoffeeScript

CoffeeScript is "an attempt to expose the good parts of JavaScript in a simple way." [12]

First appeared 2009
Creator Jeremy Ashkenas
Notable versions 1.0 (2010) • 1.9 (2015)
Recognized for Transpiling to JavaScript, Expressiveness
Notable uses Web application clients
Six words or less "It's Just JavaScript"

CoffeeScript appeared in late 2009 with the goal of making web application clients easier to write. Because web browsers run JavaScript natively, CoffeeScript code is simply translated into JavaScript for execution. In fact the **golden rule of CoffeeScript** is "It's just JavaScript."

CoffeeScript exposes JavaScript's Good Parts only, and introduces a handful of features that were not present in JavaScript when CoffeeScript was first written, among them destructuring assignment, default parameter values, splats, and the `class` syntax for creating constructors and assigning prototype properties. It also features a few concepts not (yet?) in JavaScript, including the existential operators, the `do` construct, and comprehensions. It is an **expression-oriented language**: where JavaScript has `while`, `for`, and `if` *statements*, these constructs are *expressions* in CoffeeScript.

CoffeeScript, by design, removes a lot of the "noise" in JavaScript. It does away with curly braces for bracketing (preferring indentation), parentheses for arguments in function calls (though you can use them if you like), and allows object literals to forego braces and use indented lines for each property. Many of the messy parts of JavaScript have no translation at all. It wisely throws

out JavaScript's `==` completely: writing `==`, or its synonym `is`, **transpiles** (translates at the source level) to JavaScript's `===`.

In this chapter, we will treat CoffeeScript as a language in its own right, but because of its golden rule, we'll make extensive comparisons with the language of the previous chapter.

2.1 HELLO COFFEESCRIPT

We begin with the same three introductory programs that began Chapter 1. First up is a script to generate integer right triangle measurements.

```
for c in [1..50]
  for b in [1...c]
    for a in [1...b]
      console.log "#{a}, #{b}, #{c}" if a * a + b * b is c * c
```

CoffeeScript expresses program structure by indentation rather than braces. The `for-in`-loop iterates through the *values* of an array[1] Two dots give a range an inclusive upper bound; three an exclusive range. `#{...}` evaluates expressions inside a string delimited with double quotes. The `if`-clause may follow the code to be conditionally executed.

Let's translate our command line permutations example from JavaScript:[2]

```
swap = (a, i, j) ->
  [a[i], a[j]] = [a[j], a[i]]

generatePermutations = (a, n) ->
  if n is 0
    console.log a.join ''
  else
    for i in [0..n]
      generatePermutations a, n-1
      swap a, (if n % 2 is 0 then i else 0), n

if process.argv.length isnt 3
  console.error 'Exactly one argument is required'
  process.exit 1
word = process.argv[2]
generatePermutations word.split(''), word.length-1
```

[1] In JavaScript `for-in` will iterate through the *indexes* of an array, while `for-of` iterates the values.

[2] This script is designed to be run with Node.js; running `npm install -g coffee-script` will get you a command line interpreter and program runner.

The function expression `(`*params*`) ->` *body* is equivalent to JavaScript's `function (` *params* `) {` *body* `}`. A destructuring assignment simplifies swapping array elements. And like it or not, you don't always need parentheses in function calls.

Now for our third introductory program:

```
split = require 'split'
{XRegExp} = require 'xregexp'

counts = Object.create null

process.stdin.setEncoding 'utf8'

process.stdin.pipe(split()).on 'data', (line) ->
  wordPattern = XRegExp("[\\p{L}']+", 'g')
  for word in (line.toLowerCase().match(wordPattern) or [])
    counts[word] = (counts[word] or 0) + 1

process.stdin.on 'end', ->
  for word in Object.keys(counts).sort()
    console.log "#{word} #{counts[word]}"
```

The operator `or` can be used in place of `||` (and as you probably guessed, `and` for `&&` and `not` for `!`).

Did you notice one other difference between CoffeeScript and JavaScript that we had not mentioned: the lack of a `var` or `let` keyword? Keep this in mind, we'll have a *lot* to say about it in an upcoming section.

2.2 THE BASICS

Although CoffeeScript can be considered a dialect of JavaScript, the two languages "look" rather different. JavaScript is a *curly-brace language*, using `{` and `}` to express code structure, while CoffeeScript uses indentation. Indentation gives structure not only to compound statements, but also to objects:

```
circle =
  radius: 3
  center:
    x: 5
    y: 4
  color: 'green'
  fillColor: 'pink'
  thickness: 2
  lineStyle: 'dashed'
```

Since the line breaks and the number of spaces matter in determining the structure of a script, we say CoffeeScript has a **significant whitespace** syntax, as opposed to a **free-form** syntax. The moniker "free-from" refers to the way that multiple statements can be appear on a line, or a single statement can be broken up across multiple lines, with very little, if any, constraints.

CoffeeScript employs whitespace-awareness to facilitate writing long strings, a task that many languages make surprisingly clumsy. Strings delimited by apostrophes or double quotes are allowed to span multiple lines; each line will be joined by a *single* space, even though you've kept your code pretty with proper left margins. For strings that *must* contain line breaks, use triple quotes (''' or """); CoffeeScript will suppress the initial whitespace on each line to keep everything clean:

```
example =
  multi: "This is
    really a one line
    string"
  block: """
    One
    Two
      TwoPointFive
    Three
    """

console.log example.multi
console.log example.block
```

This script outputs:

```
This is really a one line string
One
Two
  TwoPointFive
Three
```

CoffeeScript expressions, types, and variables mimic their JavaScript counterparts. Functions do too: function values can be assigned to variables, passed as parameters to other functions, and even called directly without being named:

```
square = (x) -> x ** 2
squares = [1..5].map(square)  # pass function by name
result = ((x) -> x * 5) 16    # call anonymous function

assert = require 'assert'
assert.deepEqual(squares, [1, 4, 9, 16, 25])
assert result is 80
```

Similarly, arguments are passed by value, extra arguments are ignored, and extra parameters start off `undefined`. It's also possible to specify a default value for a parameter when no corresponding argument is supplied:

```
f = (x, y=1, z=0) -> x * y + z

assert = require 'assert'
assert f(2) is 2           # 2*1+0 = 2
assert f(3, 5) is 15       # 3*5+0 = 15
assert f(2, 8, 1) is 17    # 2*8+1 = 17
```

You can even pass multiple arguments to a function, but have them all *packed* into a single parameter, using a **splat**:

```
average = (a...) ->
  (a.reduce ((x, y) -> x + y), 0) / a.length

assert = require 'assert'
result = average 7.5, -10, 50.5
assert result is 16
assert isNaN average()
```

Within the function body, the splat `a` is just an array. We can also use splats in calls to *unpack* an array before passing:

```
medianOfThree = (x, y, z) ->
  return x + y + z - (Math.max x, y, z) - Math.min x, y, z

assert = require 'assert'
numbers = [80, 20, 55]
middle = medianOfThree numbers...
assert middle is 55
```

Note the parentheses in the body of `medianOfThree` above. Without parentheses the call to `Math.max` would have parsed as follows:

```
Math.max(x, y, z - Math.min(x, y, z))
```

Be especially careful when nesting function calls, as invocations slurp as many values as they can for arguments. Writing:

```
console.log 'The average is ', average 89, 91, 'so far.'
```

displays `The average is NaN so far` since `average` grabs three arguments. Use parentheses to limit the number of arguments and do the right thing:

```
console.log 'The average is ', average(89, 91), 'so far.'
```

Moving on, CoffeeScript has a `this` keyword, which operates exactly like JavaScript's `this`. You have two nice shorthands in CoffeeScript: `@` for `this`, and `@`*property* for `this.`*property*:

```
circle =
  radius: 10,
  area: -> Math.PI * @radius * @radius
  circumference: -> 2 * Math.PI * @radius

assert = require 'assert'
assert circle.area() is 100 * Math.PI
assert circle.circumference() is 20 * Math.PI
```

This script works because functions defined with the thin arrow (`->`) set the value of `this` to the receiver when used as methods. But CoffeeScript also has JavaScript's fat arrow (`=>`), which does *not* set the value of `this`. This is useful for those cases where we have functions nested inside of methods, and we do not want the value of `this` in the nested functions to be hijacked.

The following example should help to illustrate the difference between the thin and fat arrows. A person wants to say hello after a one second delay. The global `setTimeout` function takes a function and a duration in milliseconds, and executes the function after the duration is passed. Note that we wish the value `this` within the function we pass to `setTimeout` to refer to the person, *not* the receiver of the delayed function. Our example shows both that (1) the thin arrow for the `setTimeout` callback produces an unexpected result (due to `this` being hijacked), and (2) since the fat arrow does not take a `this` of its own, it produces the desired message:

```
person =
  name: 'Alice'
  tryToSayHelloButFail: (delay) ->
    setTimeout (() -> console.log "Hi from #{@name} :("), delay
  sayHello: (delay) ->
    setTimeout (() => console.log "Hi from #{@name} :)"), delay

person.tryToSayHelloButFail(1000)           # Hi from undefined :(
person.sayHello(1000)                       # Hi from Alice :)
```

2.3 NO SHADOWING?!

In programming language theory, the **scope of a binding** is the region of the code where the binding (of a name to an entity) is in force. JavaScript and CoffeeScript both use lexical scoping: scopes are determined by looking only at the source code and not relying on any runtime behavior. JavaScript, as

we saw in the last chapter, uses the `let` or `var` keyword to explicitly create a new variable in an inner scope:

```
// JavaScript illustration of local variables
var a = 0, b = 1, c = 2;
() => {
  var a = 100;   // Local, shadows the global a
  b = 200;       // Forgot var, overwrites global b!
  var d = 300;   // Local, will not exist after return
}();

import assert from 'assert'
assert.deepEqual([a,b,c], [0, 200, 2])
assert.throws(() => d, ReferenceError)
```

Here the local variable *a* **shadows** the global variable *a*. The two variables are distinct. As long as you remember to explicitly declare the inner variable with `var` or `let`, the outer variable is safe. If you forget the declaring keyword, you'll clobber an outer variable of the same name if one is present, or throw a `ReferenceError` if one is not present.[3] CoffeeScript, though, *never* shadows: writing `a = 100` inside a function will simply create a local variable *a* unless there is a global *a*, in which case it will assign to the global variable!

```
[a, b, c] = [0, 1, 2]
(() ->
  a = 100      # Overwrites global: there's an a out there
  b = 200      # Overwrites global: there's a b out there
  d = 300)()   # Local, because there's no d out there

assert = require 'assert'
assert.deepEqual([a,b,c], [100,200,2])
assert.throws (-> d), ReferenceError
```

This design decision is controversial: a CoffeeScript function cannot, without resorting to a little hackery, declare a truly local variable. Adding a new top-level variable to a file can turn a local variable non-local! While this is unlikely to happen when good coding standards are in place, (e.g., small modules, few globals, good naming conventions), the design has generated much discussion in the CoffeeScript community.[4]

Interestingly, parameters, unlike "local" variables, *do* shadow, sp you can force the equivalent of a JavaScript-style local variable with an IIFE. Another "trick" is to use the `do` keyword, which sugars an IIFE. Let's see both tricks in action:

[3] In earlier versions of JavaScript, you could run scripts in a "non-strict mode" in which the attempt to write to a non-declared variable created a new global variable. This behavior has been identified as a cause of errors in real-world systems [16].

[4] `https://github.com/jashkenas/coffeescript/issues/712`

```
[a, b, c] = [1, 2, 3]      # globals

bad = ->
  a = 10                   # clobbers global a

okay = ->
  ((b) -> b = 20)()        # writes to local b

better = ->
  do (c=30) ->             # writes to local c

bad(); okay(); better()
assert = require 'assert'
assert.deepEqual([a,b,c], [10,2,3])
```

2.4 PROTOTYPE CHAINS

CoffeeScript's objects are just like JavaScript's: you have `{}` literals, `Object.create`, the `new` operator, and prototypes. The expression `this` (and its alias `@`) works the same way. So we *could* write:

```
# Note: this is NOT idiomatic CoffeeScript.

Circle = (x=0, y=0, radius=1, color='black') ->
  [@x, @y, @radius, @color] = [x, y, radius, color]

Circle.prototype.area = ->
  Math.PI * @radius * @radius

Circle.prototype.circumference = ->
  2 * Math.PI * @radius

circles = [(new Circle 3, 5, 10, 'blue'), (new Circle)]
```

but CoffeeScript can generate constructors and prototypes via its **class** syntax, which comes with a slick way to handle property initialization. Here is the idiomatic version of the previous script:

```
class Circle
  constructor: (@x=0, @y=0, @radius=1, @color='black') ->
  area: -> Math.PI * @radius * @radius
  circumference: -> 2 * Math.PI * @radius
  expand: (scale) -> @radius *= scale

circles = [(new Circle 3, 5, 10, 'blue'), (new Circle)]
```

```
assert = require 'assert'
assert circles[0].color is 'blue'
assert circles[1].circumference() is 2 * Math.PI
```

As in JavaScript, the CoffeeScript "class" is *just a function*, with the parameters of the constructor. The declarations inside the class—with the exception of the constructor—are properties of the associated prototype. And the additional syntactic sugar of tagging a constructor parameter with an `@` allows you elide the oft-seen (boilerplate) property initialization within constructor bodies.

The class syntax makes it convenient to express **IS-A** relationships, which we can illustrate with some animals. All animals have a name, and when told to speak, they say their name and make a species-specific sound. Animals come in three kinds: cows, horses, and sheep. A language should make it convenient to capture what is common to all kinds of animals (e.g., having a name, speaking) in one place, and isolating the kind-specific behavior (the particular sounds made by each species). Study the CoffeeScript solution:

```
class Animal
  constructor: (@name) ->
  speak: -> "#{@name} says #{@sound()}"

class Cow extends Animal
  sound: -> "moooo"

class Horse extends Animal
  sound: -> "neigh"

class Sheep extends Animal
  sound: -> "baaaa"

s = new Horse "CJ"
console.log s.speak()
c = new Cow "Bessie"
console.log c.speak()
console.log new Sheep("Little Lamb").speak()
```

Read the `extends` keyword as "is a"—that `Horse` extends `Animal` means a horse *is an* animal. Since since an animal can speak, so can a horse. This works because the class extension mechanism makes `Animal.prototype` the prototype of `Horse.prototype`. The implementation is shown in Figure 2.1.

2.5 COMPREHENSIONS

Programming languages vary in their ability to express values. Consider the common array. Sometimes we must write code to create an empty, mutable,

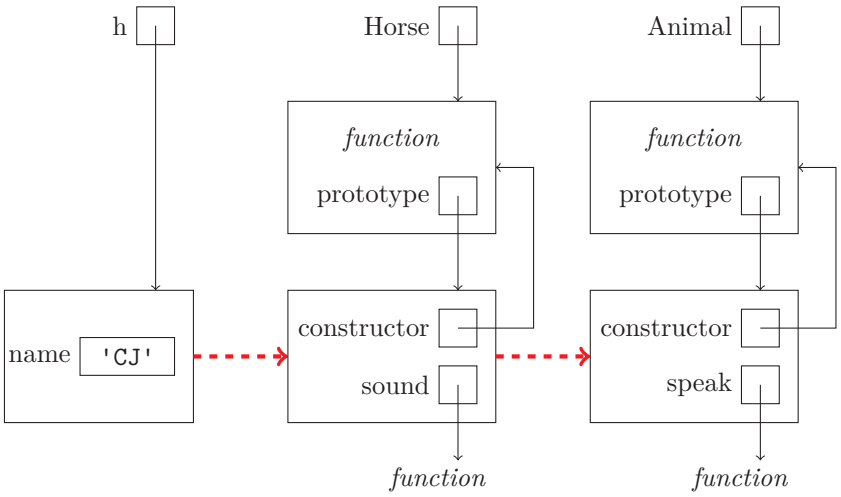


Figure 2.1 Implementation of IS-A in CoffeeScript

array, then fill it by looping through a collection of items, selecting the elements to add with an `if`-statement.

In CoffeeScript, we express such an array directly, using an array **comprehension**. Here's how we can get the names of all employees earning a salary above some threshold:

```
employees = [
  {name: 'alice', salary: 85000}
  {name: 'bob', salary: 77500}
  {name: 'chi', salary: 58200}
  {name: 'dinh', salary: 99259}
  {name: 'ekaterina', salary: 105882}]

assert = require 'assert'
highEarners = (e.name for e in employees when e.salary>80000)
assert.deepEqual(highEarners, ['alice','dinh','ekaterina'])
```

The comprehension is an expression that produces an array. No statements are required.

2.6 DESTRUCTURING

How do names take on values in a program? For many novice programmers, the answer "Assignment statements!" is 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**. Once you begin to think in generalized terms, you may start asking questions such as "Why does assignment go right to left (`x=3`) instead of left-to-right (`3=x`)?" Or "Why can't I assign multiple values at once?" You might even combine those two questions and realize that binding might just be carried out via a generalized pattern match. Consider the following expression, from a hypothetical programming language:

```
{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. CoffeeScript doesn't go quite so far as binding variables on both sides, but it does allow expressive *patterns* 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:

```
[x,y] = [10,20]              # x gets 10 and y gets 20
[x,y] = [y,x]                # yes, a swap!
{a, b} = {a: 5, b: 3}        # a nice idiom

{place: {name: mountain, loc: [lat, lon]}} = {
    place: {name: 'Everest', loc: [27.9881,86.9253]}}

assert = require 'assert'
assert.deepEqual [x, y, a, b, mountain, lat, lon],
  [20, 10, 5, 3, 'Everest', 27.9881, 86.9253]
```

This mechanism is called **destructuring assignment** because the language is breaking up the complex objects for you. Contrast this with the more verbose *explicit* destructuring:

```
# NOTE: This is NOT idiomatic CoffeeScript!
# Use destructing assignment instead!

destination = {place: {name: 'Everest', loc: [27.9881,86.9253]}}
mountain = destination.place.name
lat = destination.place.loc[0]
lon = destination.place.loc[1]

assert = require 'assert'
assert.deepEqual [lat,lon,mountain], [27.9881,86.9253,'Everest']
```

2.7 EXISTENTIAL OPERATORS

CoffeeScript, again like JavaScript, uses `null` for the intentional absence of information, and `undefined` to indicate no information is known. All other values are *known values*, and can act like objects—in fact, trying to access properties of `null` and `undefined` are among the few places a `TypeError` can ever be thrown:

```
assert = require 'assert'
assert 4.toFixed(2) is '4.00'
assert true.toString() is 'true'
assert 'abcde'.length is 5
assert [5,3,9,4,6].indexOf(3) is 1
assert.throws((-> null.toString()), TypeError)
assert.throws((-> undefined.toString()), TypeError)
```

The postfix operator `?` reports whether an expression is known, that is, not `null` and not `undefined`:

```
assert = require 'assert'

assert 78.8? is true
assert false? is true
assert []? is true
assert undefined? is false
assert null? is false
x = 9;
assert x? is true
```

The question mark combines with several operators to do some slick things. The most common use is

```
c = employee?.supervisor?.city?.name
```

which is a much nicer way to write:

```
c =
  if employee?
    _supervisor = employee.supervisor
    if _supervisor?
      _city = _supervisor.city
      if _city?
        _city.name
      else
        undefined
    else
      undefined
  else
    undefined
```

We also have the forms `a?[]` and `a?()`. And there's a `?=` which is a bit different than `or=`, and sometimes more useful. The expression `x ?= 1` assigns 1 to x only if x is `undefined` or `null`, whereas `x or= 1` will assign 1 if x is any falsy value, including 0.

The various uses of the existential operator are examples of **idioms**. Until you know what they mean, you might find them cryptic; however, once learned they are quite powerful and reduce one's cognitive load in understanding code. They also are perfectly sensible. Consider `supervisor?.name`: if the supervisor is undefined (unknown), then so is the supervisor's name.

2.8 COFFEESCRIPT WRAP UP

In this chapter we were introduced to CoffeeScript and its concise syntax. We learned that:

- CoffeeScript is "just JavaScript." In fact, the language is defined by how its constructs are translated to, or *transpiled to*, JavaScript. What is truthy (or falsy) in one language is truthy (or falsy) in the other. CoffeeScript purposely transpiles only to a subset of JavaScript, intentionally avoiding some of JavaScript's messier parts.
- CoffeeScript does not avoid all of JavaScript's bad parts. For example, it takes JavaScript's entire type system, and is therefore weakly-typed.
- CoffeeScript is a significant-whitespace language, using indentation to show structure. It provides excellent support for multiline strings.
- Thin arrow functions provide the value of the receiver (if any) to the `this`-expression, while fat arrow functions do not. The symbol `@` is an alias for `this`, and `@`x aliases `this.`x.
- In an extremely controversial design decision, CoffeeScript local variables do not shadow variables in outer scopes. To avoid potential bugs, programmers should adopt good practices such as keeping modules short, and using different naming conventions for top-level and local names. Only immediately-invoked function expressions (IIFEs) and the `do` keyword provide a means to force a variable to be "local."
- CoffeeScript has a `class` mechanism to simplify the writing of constructors and methods.
- Array comprehensions, destructuring assignment, and existential operators are among the features that make CoffeeScript a relatively concise language.

To continue your study of CoffeeScript beyond the introductory material of this chapter, you many wish to find and research the following:

- **Language features not covered in this chapter**. Symbols, regular expressions, property attributes, block comments, generators, `unless`, the operators `**`, `//`, and `%%`, `super`, the switch statement, and chained comparisons.
- **Open source projects using CoffeeScript**. Studying, and contributing to, open source projects is an excellent way to improve your proficiency in any language. You may enjoy the following projects written in CoffeeScript: atom (`https://github.com/atom/atom`), dynamics.js (`https://github.com/michaelvillar/dynamics.js`), Brunch (`https://github.com/brunch/brunch`), and Hubot (`https://github.com/github/hubot`).
- **Reference manuals, tutorials, and books**. CoffeeScript's home page, describing the entire language, and containing links to the implementation itself is at `http://coffeescript.org`. The page also contains a *Try CoffeeScript* section where you can type CoffeeScript and see the translation to JavaScript in real time. Two noteworthy books are Trever Burnham's *CoffeeScript: Accelerated JavaScript Development* [3] and Alex McCaw's *The Little Book on CoffeeScript* [26].

EXERCISES

2.1 Read the CoffeeScript documentation page at `http://coffeescript.org`.

2.2 What is the golden rule of CoffeeScript? What is the motivation for this rule?

2.3 Enter and execute several little scripts in the "Try CoffeeScript" window at the CoffeeScript home page. Pay attention to the JavaScript translations.

2.4 Install Node.js and the Node CoffeeScript module to your machine. Experiment with the CoffeeScript REPL. Execute the complete scripts from this chapter on the command line.

2.5 Research the `in` and `of` operators, and demonstrate their use in a small script of your own. How do they differ from `in` and `of` in JavaScript?

2.6 Evaluate the CoffeeScript expressions `'#{2+2}'` and `"#{2+2}"` and explain the difference.

2.7 CoffeeScript can iterate through key-value pairs of an object like so:

```
caps =
  'ME': 'Augusta'
  'VT': 'Montpelier'
  'NH': 'Concord'
  'MA': 'Boston'
```

```
  'RI': 'Providence'
  'CT': 'Hartford'

for state, capital of caps
  console.log "The capital of #{state} is #{capital}."
```

Why did we not apply this technique in our word count example near the beginning of this chapter?

2.8 JavaScript does not have a 100% free-form syntax, because of something known as *automatic semicolon insertion*, or ASI. This means that although many statements are required to end with semicolons, a language processor will insert them for you in certain cases where the language rules think you might have omitted them. Research the ASI rules. Have the ASI rules been considered a success? Give four examples of cases in which the ASI rules produce non-intuitive statement separation.

2.9 In the median-of-three-program in this chapter, what would be output if the parentheses were removed in the return statement of the function? Why?

2.10 Argue for or against this claim: "Splats are simply a syntactic device and offer no real benefits in expressive power."

2.11 Research and then summarize the arguments for and against CoffeeScript's choice to reject shadowing. See if you can find a published anecdote of a local variable inadvertently becoming global in an actual software project.

2.12 What does CoffeeScript produce when the `typeof` operator is applied to a "class"? What, then, is the proper way to understand the `class` keyword? (Feel free to compare and contrast CoffeeScript "classes" with the classes of Smalltalk, Ruby, or similar languages.)

2.13 Create a little script, modified on the animals script in this chapter, for shapes. The basic shape class should have a constructor for initializing the shape's color, and a method to produce a string stating the the shape's area and perimeter. The actual shapes should be circles and rectangles with their own constructors and implementations of area and perimeter. You will have to research CoffeeScript's `super` keyword to do a nice job.

2.14 In the expression `highEarners = (e.name for e in employees when e.salary > 80000)` we evaluate a comprehension and assign the resulting array to the variable `highEarners`. Suppose the parentheses were omitted in this expression. What would the meaning of this modified expression be?

2.15 What does the following CoffeeScript expression produce? Why?

```
[x*y for x in [1..12] for y in [1..12]]
```

2.16 Create an example to show how splats can be combined with destructuring assignment.

2.17 Improve the word count script at the beginning of this chapter by using the `xregexp` module (from npmjs.org) to allow words to contain any Unicode letter rather than just the Basic Latin letters.

2.18 For CoffeeScript practice, translate by hand the browser-based sketching program on page 30. Compare your hand-translation to the translation produced by the translator at `http://coffeescript.org`. Given that you should find some differences in translation, think about how human and machine translations are likely to be inherently different.

CHAPTER 3

Lua

Lua is a "powerful, fast, lightweight, embeddable scripting language." [23]

First appeared	1993
Creators	Roberto Ierusalimschy, Waldemar Celes, Luiz Henrique de Figueiredo
Notable versions	5.0 (2003) • 5.1 (2006) • 5.2 (2011) • 5.3 (2015)
Recognized for	Tables, Interoperability with C
Notable uses	World of Warcraft, Angry Birds, 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) with lightweight and fast implementations. Its designers created a classic **scripting language**: typically an application's logic is written in Lua, with portions, such as time-critical or device-specific code, written in the host language.

The language features a small number of types and operators, a means for defining functions, and only *one* data structure—the **table**—that does double duty representing both traditional arrays as well as key-value objects. 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. Lua's functions are first-class objects, providing all the power of functional programming. It can interoperate with other languages, including C, C++, Java, Fortran, Smalltalk, and Erlang.

In this chapter, we will introduce Lua. Because the language is small, we will give pretty good coverage of its basic elements, including a full list of its operators and types. We will cover functions and scope in some detail. We will

then see how Lua is able to work with only a single data structure, the table. We'll explain metatables, and how they provide a mechanism for user-defined types nearly identical to JavaScript's prototypes. We'll see how Lua allows its operators to take on new meanings under programmer control. We'll close with an overview of coroutines.

3.1 HELLO LUA

Welcome to Lua. Our traditional first program, listing some right-triangles, is:

```
for c = 1, 50 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
```

Here we've introduced the numeric `for`-loop (note that *both* bounds are inclusive) and `string.format` for nice output. Our second example introduces **tables**, used in Lua both for lists (indexed starting at 1) and for key-value pairs:

```
function generatePermutations(a, n)
  if n == 0 then
    print(utf8.char(table.unpack(a)))
  else
    for i = 1, n do
      generatePermutations(a, n-1)
      swapIndex = n % 2 == 0 and i or 1
      a[swapIndex], a[n] = a[n], a[swapIndex]
    end
  end
end

if #arg ~= 1 then
  io.stderr:write('Exactly one argument required\n')
  os.exit(1)
end
word = {utf8.codepoint(arg[1], 1, utf8.len(arg[1]))}
generatePermutations(word, #word)
```

Here the `arg` table contains the command line arguments, `#` is the length operator, and `~=` is the not-equal operator. As in previous chapters, we write a

message to standard error if we do not get exactly one command line argument. Strings are immutable, so we put our characters into a (mutable) table in order to generate the permutations by the sequence prescribed by Heap's algorithm.

Now let's get word counts from standard input:

```
counts = {}
for line in io.lines() do
  line:lower():gsub('[a-z\']+', function(word)
    counts[word] = (counts[word] or 0) + 1
  end)
end

report = {}
for word, count in pairs(counts) do
  table.insert(report, string.format('%s %d', word, count))
end
table.sort(report)
for _, line in ipairs(report) do
  print(line)
end
```

We've had to do a little more work than in previous chapters, because Lua tables can only be sorted over their integer-indexed keys, and counting words required a table keyed on words. We had to build a second, list-like, table with the output lines in order to produce sorted output. On the plus side, we can iterate through the lines of a file using the builtin `io.lines` function, without resorting to an external module as we needed to do in JavaScript and CoffeeScript.

3.2 THE BASICS

All values in Lua belong to one of eight types: nil, boolean, number, string, function, thread, userdata, and table. The first four are primitive types, and the latter four are reference types. Primitive and reference have the same meanings as in JavaScript: primitives are immutable; references allow access to objects through multiple variables simultaneously. Lua doesn't do as many implicit type conversions as JavaScript but it does do some: anything can be coerced to a boolean, strings and numbers are coercible to each other[1], but `true` and `false` are not coerced to strings or numbers. You can use a variable without declaring it; its value in this case is just `nil`. Unlike JavaScript and CoffeeScript, the *only* things that are falsy in Lua are `nil` and `false`.

[1] However the attempt to coerce a non-numeric-looking string to a number generates an error, not `NaN`.

Strings are sequences of 8-bit values (not 16), so the length of the string "café" as computed with the `#` operator, is 5, not 4, since it is counting the bytes in the UTF-8 encoding. Use `utf8.len` to count characters.

```
assert(x == nil)              -- Does not fail! x is nil

s = "caf\u{e9}"
assert(#s == 5)               -- counts bytes
assert(utf8.len(s) == 4)      -- counts characters

function firstFewPrimes()
  return {2, 3, 5, 7, 9, 11, 13}
end

assert(type(4.66E-2) == "number")
assert(type(true and false) == "boolean")
assert(type('message') == "string")
assert(type(nil) == "nil")
assert(type(firstFewPrimes) == "function")
assert(type(firstFewPrimes()) == "table")
assert(type(coroutine.create(firstFewPrimes) == "thread"))

assert(0 and "" and 0/0)      -- all of these are truthy!
assert(not(false or nil))     -- only false and nil are falsy
```

Lua (as of version 5.3) has 25 operators across 12 precedence levels. From highest to lowest precedence, they are:

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

There are a few differences from JavaScript here. Lua adds an exponentiation

operator, a length operator, a floor division operator (i.e., `5 // 2 == 2` but `5 / 2 == 2`), and wisely uses different operators for string concatenation and numeric addition. This means that Lua avoids the classic surprise in which `"The answer is " + x + y` reports an "answer" of 22 even when x and y both have a numeric value of 2. Lua does not have a conditional operator (e.g., `_?_:_`), nor any assignment operators! Assignment in Lua is a statement, not an expression. This may be a Good Thing: assignment has side-effects, and its appearance in the middle of an expression may be confused (by some) with an equality test.

In an assignment statement such as `x, y = 0, 0` we have a *name list* on the left and an *expression list* on the right. An expression list isn't an actual list object, nor is it a tuple. It's simply a list of expressions that get evaluated and then assigned to the variables on the left side. "Extra" expressions on the right are ignored; extra names on the left become `nil`. Expression lists also appear in function return statements, since a function may return multiple values:

```
function computeThreeThings()
  return 5, 6, 7
end

a, b = computeThreeThings()         -- extra results ingnored
c, d, e, f = computeThreeThings()   -- extra vars get undefined
g, h, i = 4, computeThreeThings()   -- right-hand-side is 4, 5, 6, 7

assert(a == 5 and b == 6)
assert(c == 5 and d == 6 and e == 7 and f == nil)
assert(g == 4 and h == 5 and i == 6)
```

Lua's error-handling mechanism does not use the ubiquitous `throw` and `try` statements, and there is no special exception type. However, errors, whether from adding booleans, calling `nil` as a function, or explicitly calling the `error` function, 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 an error message, if applicable. 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, result = pcall(add, 5, false)
assert(success == false)
```

3.3 SCOPE

Lua's visibility rules are fairly straightforward: (1) Variables are global unless defined using `local`; (2) shadowing happens; and (3) quoting the reference manual, "The scope of a local variable begins at the first statement after its declaration and lasts until the last non-void statement of the innermost block that includes the declaration." [24] Blocks are the bodies of `do`, `if`, `while`, `repeat`, and `for` statements, as well as function bodies. Rule 3 means that we can use a name prior to its declaration in a block, and even in the initializing expression of its declaration; it this case, it will refer to the nonlocal entity currently bound to the name:

```
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, local gone
```

There is another interesting consequence of rule 3: 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)
print(reason)                         -- your output will vary
```

Because the local f is not in scope until after its declaration is complete, we can make a function recursive by declaring it on one line, then assigning to it on the next:

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

Because recursion is not uncommon, Lua provides an equivalent sugared form:

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

3.4 TABLES

Like JavaScript and CoffeeScript, Lua provides a single structure that can be used to hold both named and numbered properties. Lua's structure is called a **table**. Here is a contrived example to show how tables are created and iterated over. The functions `pairs` and `ipairs` produce

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

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
```

Keys are generally enclosed in brackets, except when the key is a simple name (letters, digits, and underscores, not beginning with a digit). Unspecified keys are auto-generated as the integers 1, 2, 3, and so on.[2]

Keys are not restricted to strings and numbers, but they cannot be `nil`. Values can't be `nil`, either, since `nil` turns out to be the response to a request for a value at a nonexistent key. This treatment of `nil` is consistent with Lua's evaluation of an undeclared variable as `nil`. A table value of `nil` would be indistinguishable from the key not being present in the table. Therefore, to remove an element from a table, set the value at the desired key to `nil`.

```
colors = {'red', 'blue', 'green'}
dog = {name = 'Lisichka', breed = 'G-SHEP', age = 13}

-- Length operator counts number of integer keys only
assert(#colors == 3)
assert(colors[1] == 'red')

-- Need our own function to count all pairs!
```

[2] Yes, they start at 1, not 0.

```lua
function number_of_pairs(t)
  local count = 0
  for _, _ in pairs(t) do count = count + 1 end
  return count
end

-- Assignment of nil removes a key pair
assert(number_of_pairs(dog) == 3)
dog.age = nil
assert(number_of_pairs(dog) == 2)
```

Global variables are actually kept in a table called the *global environment*. This table contains

- A string at index `_VERSION`;
- Functions, including `assert`, `pairs`, `pairs`, `print`, `pcall`, `tonumber`, `tostring`, `require` among others; and
- 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.

3.5 METATABLES

Recall that in JavaScript and CoffeeScript, every object has a prototype (unless explicitly created, by the programmer, without one) to which it delegates the search for missing properties. Lua supports delegation, too: a table may have a **metatable**, though metatables do much more than simply extend the search for properties!

To attach metatable m to a table t, invoke `setmetatable(t,m)`.[3] The metatable will contain zero or more of the following properties, 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

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

for subtraction, `__unm` for unary minus, `__concat` for `..`, `__len` for `#` and so on.[4]

The `__index` property allows delegation of property lookup: its value can be a function that returns the value for the missing property, or a table holding the missing property. Let's recast our JavaScript delegation example from Chapter 1 into Lua:

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

Now let's use metatables to create a "type" of two-dimensional vectors useful in graphics applications. For simplicity, we'll keep our code light and implement only construction, vector addition, vector dot product, a magnitude function, and a conversion to string. Here is one implementation:

```
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
  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)
```

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

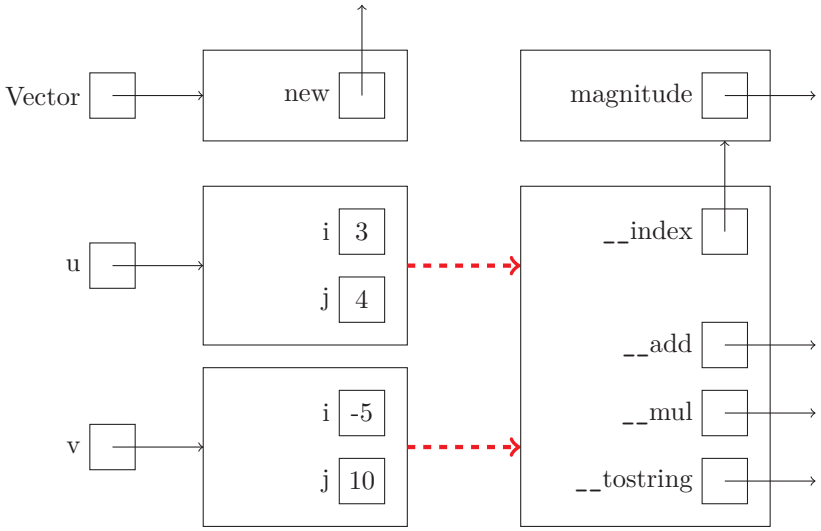


Figure 3.1 Vectors in Lua

Each call to `Vector.new` produces a vector instance, a table with `i` and `j` components, and a metatable containing operator implementations and a reference to a table with the shared `magnitude` property. Lua doesn't have the magic `this`-expression of JavaScript, so the magnitude function must explicitly be defined to take an argument. This means an invocation can be written:

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

However, Lua allows `v:magnitude()` for this expression, which is better because it evaluates v only once.

3.6 COROUTINES

A function runs until it terminates (via a normal return or an error), then returns to its caller. A **coroutine** can do the same, but also **yield**, in which case it can be *resumed* to continue where it left off. Coroutines make writing on-demand sequences fairly easy:

```
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
```

Note that `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 running or has yielded and 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.

Coroutines allow us to structure an application into multiple threads of control, each written rather independently of each other. A game might contain coroutines for players, volcanos, tornados, waterfalls, and an event manager to listen for user input. Only one coroutine runs at a time, so each must contain explicit `yield` and `resume` calls, and not take up too much time between these calls. This architecture is known as **cooperative multitasking**; contrast this with **preemptive multitasking**, where the operating system can preempt a thread at anytime and give control to another thread.

3.7 LUA WRAP UP

In this chapter we were introduced to Lua, a high-performance, embeddabe, lightweight, scripting language. We learned that:

- Lua is a relatively small language, with only a few statements, operators, types, and only one data structure.
- The only falsy values are `false` and `nil`.

- In Lua, unlike many other languages, assignment is a statement and not an expression.
- Lua has no conditional operator.
- Lua functions can return zero or more values. There is no tuple type in Lua; functions actually can return multiple values.
- Lua variables can have global scope or local scope. Local variable 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 the programmer must take care in creating local recursive functions.
- Tables are used for both classic arrays (1-based sequences of numerically indexed values) and dictionaries (sets of key-value pairs). Both numeric and non-numeric indexed values can be mixed in a single table; it is up to the language implementation to keep the "array-part" access efficient.
- Like JavaScript, Lua's standard library is actually a collection of objects (e.g., `io`, `os`, `math`, etc.)
- 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 19 of the operators. In addition, metatables can be used to build the same prototypal inheritance mechanism found in JavaScript.
- Lua directly supports coroutines: calling `coroutine.create` on a function object produces a coroutine (of type `thread`). Coroutines can yield and be resumed, and their status can be queried.

To continue your study of Lua beyond the introductory material of this chapter, you many wish to find and research the following:

- **Language features not covered in this chapter**. Chunks, environment variables, garbage collection metamethods, weak tables, the auxiliary library, the standard library, and interoperability with C (the API).
- **Open source projects using Lua**. Studying, and contributing to, open source projects is an excellent way to improve your proficiency in any language. Of the many projects using Lua, you may enjoy Luvit (`https://github.com/luvit/luvit`), CorsixTH (`https://github.com/CorsixTH/CorsixTH`), termtris (`https://github.com/tylerneylon/termtris`), and PacPac (`https://github.com/tylerneylon/pacpac`).
- **Reference manuals, tutorials, and books**. Lua's home page is `http://www.lua.org/`. It's Reference Manual (as of Version 5.3) is freely available at `http://www.lua.org/manual/5.3/`. A good selection of Lua books can be found at `http://www.lua.org/docs.html#books`.

We recommend the most recent version of *Programming in Lua* by Lua's creator, Roberto Ierusalimschy. You may also like one of the few books covering Lua's use in Game Programming.

EXERCISES

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

3.2 Practice with the Lua REPL.

3.3 What do you think about Lua's choice of allowing you read undeclared variables without an error? Do you find this error-prone, or useful? Why?

3.4 Contrast Lua's support for assignment with that of JavaScript and CoffeeScript. Consider assignments such as

```
{a: y, b: [_, c]} = {b: [10, true], a: 9, c: 5}
```

Does this work in JavaScript? CoffeeScript? Lua?

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

3.6 To compensate for the lack of a conditional expression, some suggest simulating JavaScript's x`?`y`:`z with the Lua expression x `and` y `or` z. Is the simulation correct? If not, how exactly does it differ from the intent of the conditional?

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

3.8 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);
}
```

3.9 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?

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

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

3.12 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?

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

3.14 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.