SAMPLE CHAPTER

CoffeeScript IN ACTION

Patrick Lee

MANNING

CoffeeScript in Action

by Patrick Lee

Chapter 2

brief contents

2 Simplified syntax

This chapter covers

- Basic syntax and structure
- Expressions and operators
- An introduction to strings, arrays, objects, and functions
- How to run CoffeeScript programs

Before going to a country where you don't speak the language, you might spend some time listening to the language to get a feel for it or maybe learn some essential canned phrases to help you get around. Think of CoffeeScript in the same way. As you're immersed in the syntax of CoffeeScript, you'll start to build your understanding.

In this chapter, you'll learn about expressions, operators, and statements, as well as how they work in CoffeeScript and how they're related to the JavaScript equivalents. You'll explore fundamental building blocks of CoffeeScript programs with strings, arrays, comments, and regular expressions. You'll begin with a program.

Click here to open the source code for this chapter.

2.1 Your first program

Imagine a small program that controls a coffee machine (called the Barista). This coffee machine serves different styles of coffee, some with milk and some without, but it doesn't serve any coffee styles containing milk after midday. Now imagine you find an existing implementation of this CoffeeScript program, as shown in the following listing.

Listing 2.1 The coffee machine

```
houseRoast = null

hasMilk = (style) ->
  switch style
    when "latte", "cappuccino"
      yes
    else
      no

makeCoffee = (requestedStyle) ->
  style = requestedStyle || 'Espresso'
  if houseRoast?
    "#{houseRoast} #{style}"
  else
    style

barista = (style) ->
  time = (new Date()).getHours()
  if hasMilk(style) and time > 12 then "No!"
  else
    coffee = makeCoffee style
    "Enjoy your #{coffee}!"
```

A function to determine if a style of coffee has milk

A function to make the coffee; returns a string

A function that coffee is requested from

You don't yet fully grasp CoffeeScript, but already you can get a feel for the basic structure. An equivalent JavaScript program has a very similar structure but slightly different syntax, as you'll see in listing 2.2. Compare these programs side by side and you'll begin to understand the syntax features that CoffeeScript removes and appreciate *why* it removes them.

Why remove?

Claude Debussy is quoted as saying that "music is the space between the notes." CoffeeScript syntax is defined as much by what is missing as by what is present. Part of the thought experiment behind CoffeeScript is to take patterns written frequently in JavaScript, look at them, and ask, "How much of this is necessary?"

2.1.1 Comparing CoffeeScript to JavaScript

The CoffeeScript implementation of the coffee machine appears side by side with an equivalent JavaScript implementation in the next listing. See how many differences you can spot.

Listing 2.2 Comparing CoffeeScript to JavaScript

CoffeeScript

```
hasMilk = (style) ->
  switch style
    when "latte", "cappuccino"
      yes
    else
      no

makeCoffee = (style) ->
  style || 'Espresso'

barista = (style) ->
  now = new Date()
  time = now.getHours()
  if hasMilk(style) and time > 12
    "No!"
  else
    coffee = makeCoffee style
    "Enjoy your #{coffee}!"

barista "latte"
```

Top level, not indented

Indentation inside function

Indentation inside if

Indentation inside else

JavaScript

```
var hasMilk = function (style) {
  switch (style) {
    case "latte":
    case "cappuccino":
      return true;
    default:
      return false;
  }
};

var makeCoffee = function (style) {
  return style || 'Espresso';
};

var barista = function (style) {
  var now = new Date();
  var time = now.getHours();
  var coffee;
  if (hasMilk(style) && time > 12) {
    return "No!";
  } else {
    coffee = makeCoffee(style);
    return "Enjoy your "+coffee+"!";
  }
};

barista("latte");
```

JavaScript requires var declaration

Returns either "No!" or "Enjoy your latte!" depending on the time of day

When you compare the CoffeeScript and JavaScript in listing 2.2, you'll notice four key things that CoffeeScript is missing. It has no `var` statements, semicolons, `return` statements, or curly braces.

VAR STATEMENTS

In JavaScript, unless you're in strict mode it's possible to accidentally assign variables on the global object, in the global scope. If you don't know what that means, for now just know it's a bad thing. CoffeeScript always defines new variables in the current scope, protecting you from this unfortunate feature in JavaScript.

SEMICOLONS

In CoffeeScript there are no semicolons.[1] Although they're allowed, you never need them, and they shouldn't be used.

[1] Sure, ECMA-262 says that JavaScript parsers should do automatic semicolon insertion, but the potential for errors in JavaScript has resulted in frequent advice to always use them.

RETURN STATEMENTS

The `return` keyword is absent from CoffeeScript. In CoffeeScript the last expression in the function is returned without any `return` statement. This is called an *implicit return*. Although required occasionally, explicit `return` statements are rarely used in CoffeeScript.

CURLY BRACES

In CoffeeScript there are no curly braces used to mark out blocks of code. In order to remove the need for curly braces and other block-delimiting characters, newlines and the indentation level for each line of code are meaningful. This is referred to as *significant whitespace* or sometimes as the *offside rule*.[2] Indentation matters in CoffeeScript programs, so you must be careful to indent consistently. Always use either two, three, or four spaces in CoffeeScript. Two spaces is the most common. If you use mixed indentation (sometimes using two, sometimes using five, sometimes using four), then your CoffeeScript might still compile, but it will almost certainly not do what you want it to. Almost all CoffeeScript programs you'll find in the wild use spaces. Using tabs is not recommended.

As you now turn your attention to expressions and some basic language features of CoffeeScript, keep the rules about `var`, semicolons, `return` statements, and curly braces in the back of your mind until you encounter them again. Unfortunately, reading about basic language features is a little bit like reciting the alphabet or playing musical scales. On the upside, once you see these basic features, you're better prepared for the real fun stuff.

2.2 Simple expressions

An expression is something that can be *evaluated*. An expression has a value. Almost everything in CoffeeScript is an expression. CoffeeScript even goes to some effort to make some things expressions that aren't expressions in the underlying JavaScript. This emphasis on expressions means that they're a good place to start exploring the syntax of CoffeeScript—starting with small expressions and moving on to larger ones.

All of the examples in this section can be run on the CoffeeScript Read-Eval-Print Loop (REPL). Start your REPL:

```
> coffee
coffee>
```

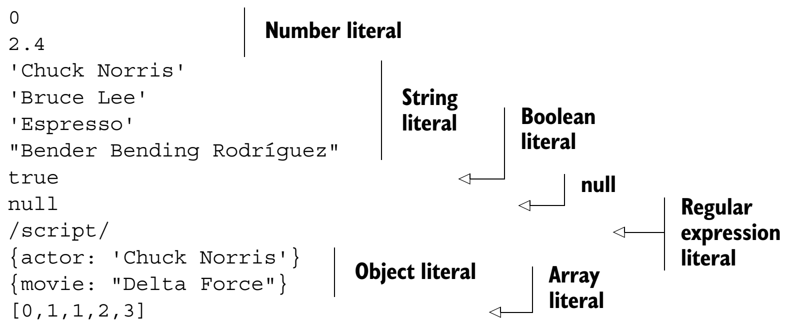
Ready?

2.2.1 Literal values

The smallest expressions in CoffeeScript are ones that evaluate to themselves. When you type them into the REPL and press Enter, you see the same thing shown on the

[2] Significant indentation will already be familiar to Python and F# programmers and anybody who has used HAML or SASS.

next line. These expressions are called *literal values*, and the notation used to write them is called *literal notation*:



```
0
2.4
'Chuck Norris'
'Bruce Lee'
'Espresso'
"Bender Bending Rodríguez"
true
null
/script/
{actor: 'Chuck Norris'}
{movie: "Delta Force"}
[0,1,1,2,3]
```

The CoffeeScript literal values shown here are exactly the same in JavaScript. Not all literal values are the same though, so everybody (even seasoned JavaScript programmers) needs to learn something about literal values in CoffeeScript.

Function

One very important expression that's different from JavaScript is the *function literal*:

```
(x) -> x
```

In JavaScript, the function literal requires a bit more typing:

```
function (x) { return x; }
```

Functions are used all the time in both JavaScript and CoffeeScript. Removing the `function` keyword, curly braces, and `return` statements for CoffeeScript reduces the amount of boilerplate and gives your code greater prominence over language syntax.

Object

The curly braces on an object are optional in CoffeeScript:

```
movie: "Delta Force"
# { movie: "Delta Force"}
```

Boolean aliases

CoffeeScript has aliases for the literal values `true` and `false`:

```
on
# true
yes
# true
off
# false
no
# false
```

Regular expressions

Like JavaScript, a regular expression literal in CoffeeScript begins and ends with a forward slash /:

```
/abc/
# /abc/
```

Unlike JavaScript, though, a regular expression must not start with a literal space:

```
/ abc/
# error: unexpected /
```

Once you have expressions, you need a way to name things. You need variables.

2.2.2 Variables

Names that refer to local values are called *variables*. A name should contain only letters, numbers, and underscores. Other characters are permitted, such as `π` (pi) and `$` (dollar sign), but you don't need them. A variable name must not be one of the reserved names. A list of reserved words that you can't use for names is in appendix A.

UNDEFINED

When you use a name that hasn't had any value assigned to it, you get an error telling you that the name is not defined:

```
pegasus
# ReferenceError: pegasus is not defined
```

Names that aren't defined have a special type called *undefined*. To define a variable, you assign a value to a name. This causes the variable to *reference* that value:

```
answer = 42
neighborOfTheBeast = 668
blameItOnTheBoogie = yes
texasRanger = {actor: 'Chuck Norris'}
```

When you evaluate a variable, you get the value referenced by it:

```
answer
# 42
```

If you assign a new value to the variable, the name will then evaluate to that new value:

```
texasRanger = true
texasRanger
# true
```

In CoffeeScript you can assign anything you want to a variable regardless of what was previously assigned to it. Any language that allows this is called *dynamically typed*.

To create a variable, you have to assign a value to it. When you do that, you use an operator.

2.3 Operators

Simple expressions are important, but in order to do things with those expressions (like assign a value to a variable) you need some operators to go with them. CoffeeScript has many operators that work exactly the same as they do in JavaScript:

```
+  -  !  /  %  >  <  >=  <=  .  &&  ||  *
```

Other operators from JavaScript, such as the ternary operator, are either different or unavailable in CoffeeScript, so for the time being you should avoid using them.

It's now time to look at the essential operators that provide basic syntax to CoffeeScript, some of the new operators that CoffeeScript introduces, and how operators are used to combine expressions. All of the examples in this section can be run on the CoffeeScript REPL. Type them all into the REPL, see the results, and experiment with them.

2.3.1 Essentials

Some operators come from JavaScript with only minor changes and, in some cases, aliases. The operator precedence rules are unchanged from JavaScript.

Assignment

This operator is provided by the `=` symbol. Use it when you need to assign a value to a name:

```
wuss = 'A weak or ineffectual person'
chuckNorris = 'Chuck Norris'
```

CoffeeScript doesn't let you accidentally declare global variable names—you don't want global variables.

Negation

This operator is provided by `!` or the alias `not`. Use it to get `true` or `false` depending on whether the value is truthy or falsy:

```
!true
# false
```

Falsy values

The values `null`, `false`, `''`, `undefined`, and `0` are called *falsy* values in CoffeeScript because they have the value `false` when coerced to a boolean, such as when used in an `if` clause. All other values in CoffeeScript will have the value `true`, making them *truthy*. You can observe this on the REPL by evaluating each of them prefixed with `!!`, such as `!!''`, which will give you `false`.

Equality and inequality

These operators are provided by `==`, `!=`, or the aliases `is` and `isnt`, respectively. Use them to determine whether two values are equal or not:

```
chuckNorris = 'Chuck Norris'
weak = 'weak'
chuckNorris is weak
# false
chuckNorris isnt weak
# true
```

Be careful; `isnt` and `is not` are *not* the same thing in CoffeeScript:

```
5 isnt 6                          ◁ Means 5 and 6 are
# true                              not the same value
5 is not 6        ◁ Means 5 is the same
# false             value as not 6
```

The lesson is to avoid using `is not` in CoffeeScript and instead use `isnt` to test for inequality.

TYPE COERCION

Unlike in JavaScript, the equality and inequality operators in CoffeeScript aren't *type coercive*. They're equivalent to `===` and `!==` in JavaScript:

```
'' == false
# false
```

What does it mean that these operators aren't type coercive? Well, if the equality operator in CoffeeScript were type coercive, then the expression `1 == '1'` would evaluate to `true` because the operator would try coercing the values when it compares them:

```
1 == '1'
# false
```

In CoffeeScript the equality operator requires that both sides have the same value, not that they can be coerced into the same value. If you want a value to be coerced, then you should do it yourself. Read on to find out how.

ADD AND SUBTRACT

These operators are provided by `+` and `-`. Use `+` only with numbers or strings:

```
3 + 3
# 6
'string' + ' concatenation'
# 'string concatenation'
```

When you add a string to a number, the `+` operator will coerce the number to a string:

```
4 + '3'
# '43'
```

Use `-` only with numbers,

```
3 - 3
# 0
```

not with strings, which will evaluate to the primitive value that means *not a number* (`NaN`):

```
'apples' - 'oranges'
# NaN
```

If you get `NaN`, it usually means something has gone wrong.

MULTIPLY AND DIVIDE

These operators are provided by `*` and `/`. Use them with numbers. They work exactly the same as in JavaScript:

```
3*3
# 9
```

When you multiply a string by a number, the `*` operator will attempt to coerce the string into a number:

```
'3'*3
# 9
```

If the string can't be coerced into a number, then you get `NaN`:

```
'bork'*3
# NaN
```

You tried to multiply a string by a number and something went wrong.

Modulo

Modulo is the division remainder. Use it to see if a number is evenly divisible by another number:

```
3%2
# 1

4%2
# 0

not (3%2)
# false

not (4%2)
# true
```

Comparison

These operators are provided by `<`, `>`, `<=`, and `>=`. Use them when you want to compare number values or string values:

```
42 > 0
# true

42 >= 42
# true
```

Numbers are compared exactly how you'd expect:

```
time = 13
time > 12
# true
```

Strings are too. They're compared alphabetically:

```
'Aardvark' < 'Zebra'
# true
```

When you try to compare things that can't reasonably be compared, you get `false`:

```
2 > 'giraffe'
# false
```

The guard (logical and)

This operator is provided by `&&` or the alias `and`. You use it when you want to evaluate an expression only if another expression is `true`:

```
chuckNorris is weak and pickFight
```

You evaluate the guard by first looking to the left of the `and` operator:

```
chuckNorris is weak
# false
```

If that evaluates to `false`, the expression to the right of the `and` operator isn't evaluated. The value of the expression then will be the value of the left-hand side:

```
chuckNorris is weak and pickFight
# false
```

THE DEFAULT (LOGICAL OR)

The counterpart of the guard, default, is provided by `||` or the alias `or`. You use it to evaluate an expression only if another expression is `false`:

```
runAway = 'Running away!'
chuckNorris is weak or runAway
```

The coffee machine program uses the default operator to provide a default style of coffee:

```
makeCoffee = (requestedStyle) ->
 requestedStyle || 'Espresso'

makeCoffee()
# 'Espresso'
```

Evaluate the default in your head by first looking to the left of the `or`:

```
chuckNorris is weak
# false
```

When that evaluates to `false`, the expression to the right of the `or` *is* evaluated:

```
runaway
# 'Running away!'
```

That's the opposite of how the guard operator works.

FUNCTION INVOCATION

A function is invoked by placing a value after it. Consider the `makeCoffee` function:

```
makeCoffee = (style) ->
  style || 'Espresso'
```

Invoke it with the value `'Cappuccino'`:

```
makeCoffee 'Cappuccino'
# 'Cappuccino'
```

The last value evaluated in a function is the value you get when that function is invoked. In this example, `'Cappuccino'` is the last value evaluated. Function values are covered in depth in chapter 3.

NEW

You use the `new` operator to get an instance of a class of object. Following are the date and time when this sentence was first written:

```
new Date()
# Sun, 21 Aug 2011 00:14:34 GMT
```

PROPERTY ACCESS

This operator is provided by `.` or by `[]`. You use them to access a property on an object:

```
texasRanger = actor: 'Chuck Norris'       ◁ An object defining a
texasRanger.actor                           single property, actor
# 'Chuck Norris'
```

Square brackets are useful when you have the property name in a value:

```
movie = title: 'Way of the Dragon', star: 'Bruce Lee'    ◁ An object defining
myPropertyName = 'title'                                    two properties:
movie[myPropertyName]                                       title and star
# 'Way of the Dragon'
```

A date object has a `getHours` property that can be used to get the hours part of a date. You can invoke that function on the date object created using the `new` operator:

```
now = new Date()
# Sun, 21 Aug 2011 00:14:34 GMT
now.getHours()
# 0
```

Objects and properties are covered in depth in chapter 4. You can now turn your eyes to types of things in CoffeeScript.

2.3.2 *Types, existential, and combining operators*

CoffeeScript has some operators that are not in JavaScript. These operators provide cleaner syntax for some common JavaScript idioms. One of these, called the existential operator, is useful for expressions such as "is there a house roast?"

```
houseRoast?
```

To understand the existential operator, you need to understand `undefined`, `null`, and types.

UNDEFINED

A variable that hasn't been assigned a value doesn't reference anything and so has the value `undefined`. When you evaluate an undefined value, you get a reference error (as you saw earlier with Pegasus):

```
pegasus
# ReferenceError: pegasus is not defined
```

NULL

A variable that is defined can have the `null` value. The `null` value is equal to itself:

```
reference = null
reference == null
# true
```

TYPES

CoffeeScript is dynamically and weakly typed, which means that the `typeof` operator was never going to be particularly useful. Also, the type of `null` is object:

```
typeof null
# 'object'
```

This problem in JavaScript makes `typeof` barely worth the pixels it appears on.

Programmers who normally use a language that's statically typed need to suspend disbelief as they learn how to solve the same problems without types. Techniques for doing so are covered in chapter 7. For now, remember that CoffeeScript is dynamically and weakly typed:

```
dynamicAndWeak = '3'
weakAndDynamic = 5
dynamicAndWeak + weakAndDynamic
# '35'
dynamicAndWeak = 3
dynamicAndWeak + weakAndDynamic
# 8
```

5 is coerced by + operator to '5'

Used to reference a string, now references a number

There are no type declarations, and the types of variables can change (remember, this is dynamic typing). Also, the addition operator works differently in the two examples because of type coercion (this is known as weak typing). *Don't rely on types in CoffeeScript.*

Existential operator

The existential operator is provided by `?`. You use it to evaluate whether something is defined *and* has a value other than `null` assigned to it:

```
pegasus?
# false
roundSquare?
# false
pegasus = 'Horse with wings'
pegasus?
# true
```

When something hasn't been defined, it has the *undefined* type:

```
typeof roundSquare
# 'undefined'
```

But *undefined* has an uncomfortable place in JavaScript, and type checks should generally be avoided. Here's a common phrase from JavaScript:

```
typeof pegasus !== "undefined" and pegasus !== null
#false
```

With the existential operator, you have a simpler way to express the same thing:

```
pegasus?
# false
```

CoffeeScript has other more advanced but less commonly used operators that are covered in chapter 7. For now, you can get by just fine with the basic operators.

Combining expressions

Operators can be used as parts of expressions, and one expression can be made up of multiple expressions. Operators are used to connect expressions, and connecting expressions using operators results in another expression, as shown in figure 2.1.

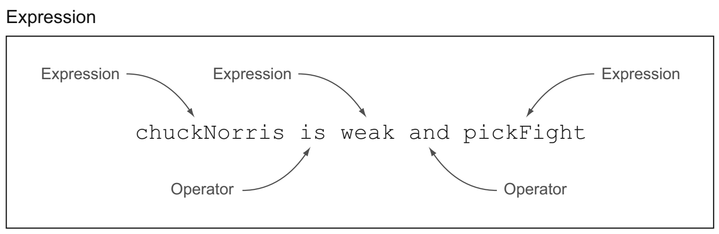


Figure 2.1 Expression anatomy

2.3.3 Exercises

To learn CoffeeScript, you need to write CoffeeScript. At the end of some sections in this book is a set of exercises for you to attempt. The answers to the exercises for all sections are in appendix B.

Suppose you just obtained two items, a torch and an umbrella. One of the items you purchased, and the other was a gift, but you're not sure which is which. Both of these are objects:

```
torch = {}
umbrella = {}
```

Either the torch or the umbrella has a `price` property, but the other does not. Write an expression for the combined cost of the torch and umbrella. *Hint: Use the default operator.*

2.4 Statements

Expressions and operators are important, but you'll also need to use statements to work effectively in CoffeeScript. Statements are executed but don't produce a value. When expressions are executed, they *do* produce a value:

```
balance = 1000
while balance > 0
  balance = balance - 100
```

Here, the `while` keyword is a statement, so it doesn't have a value but is an instruction only. In comparison, `balance - 100` is an expression and it has a value. That value is assigned to the variable `balance`.

In CoffeeScript you should always prefer expressions because expressions will lead to simpler programs—you'll learn more about that in later chapters. This section takes some common statements from JavaScript and demonstrates how they're used in CoffeeScript as part of an expression. Before getting to the individual examples, you'll look at the basic syntactic parts, or anatomy, of an expression.

2.4.1 Anatomy

Things that are only statements in JavaScript can be used as expressions in CoffeeScript. An example of this is the `if` statement, as shown by figure 2.2.

In JavaScript an `if-else` block like this can't be used as an expression. In CoffeeScript it can be.

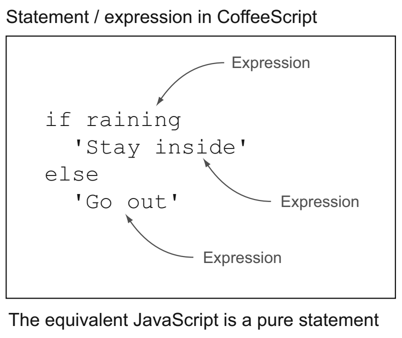

Statement / expression in CoffeeScript
Expression
if raining
'Stay inside'
else
'Go out'
Expression
Expression
The equivalent JavaScript is a pure statement

Figure 2.2 Expression and statement anatomy

2.4.2 Statements as expressions

Things that are only statements in other languages, including JavaScript, can be used in CoffeeScript as expressions for the values they produce. Ruby programmers and Lisp programmers will be familiar with the idea that *everything is an expression*, but if you come from a language that doesn't do this, then it's time for some reeducation.

IF STATEMENTS

These are provided by `if` and optional `else` keywords. Use them when you want different things to be evaluated, depending on whether a particular value is `true` or `false`:

```
if raining
  'Stay inside'
else
  'Go out'
```

You can imagine the equivalent JavaScript. It has parentheses and curly braces. More importantly, though, an `if` block is also an expression in CoffeeScript; it has a value and can be assigned to a variable:

```
raining = true

activity = if raining
    'Stay inside'
  else
    'Go out'

activity
# 'Stay inside'
```

Don't use ternary expressions

CoffeeScript compiles an `if` statement used in an expression to use JavaScript's ternary expression. The ternary operator looks like `raining ? 'Go out':'Stay inside'`. If you're a JavaScript developer, don't use the ternary operator directly in CoffeeScript—it won't work.

Switch statements

These are provided by the `switch` and `when` keywords, with the default option using the `else` keyword. Use them when you want different things to be evaluated depending on the value of an expression. The switch is often a good replacement for multiple `if`, `else` blocks:

```
connectJackNumber = (number) ->
  "Connecting jack #{number}"

receiver = 'Betty'

switch receiver
  when 'Betty'
    connectJackNumber 4
  when 'Sandra'
    connectJackNumber 22
  when 'Toby'
    connectJackNumber 9
  else
    'I am sorry, your call cannot be connected'

# 'Connecting jack 4'
```

You can use a `switch` block in an expression in the same way you can use an `if` block in an expression:

```
month = 3
monthName = switch month
  when 1
    'January'
  when 2
    'February'
  when 3
    'March'
  when 4
    'April'
  else
    'Some other month'

monthName
# 'March'
```

Use a switch to determine if a style of coffee has milk in it:

```
style = 'latte'
milk = switch style
  when "latte", "cappuccino"
    yes
  else
    no

milk
# true
```

Only one block of the switch is evaluated (there is no fall-through). A switch at the end of a function returns the evaluation of one block:

```
hasMilk = (style) ->
  switch style
    when "latte", "cappuccino"
      yes
    else
      no

hasMilk 'espresso'
# false
```

Match multiple options.

The `when` keyword takes multiple options, each separated by a comma. If there is no `else` clause and none of the `when` clauses are matched, then evaluating the switch expression results in the value `undefined`:

```
pseudonym = 'Thomas Veil'

identity = switch pseudonym
  when 'Richard Bachman'
    'Stephen King'
  when 'Ringo Starr'
    'Richard Starkey'

identity
#
```

identity is declared but has the value undefined. Evaluating it on the REPL results in an empty line.

Loops

Loops are provided by the `while`, `until`, or `loop` keywords. Use them to do something, such as clean, repeatedly:

```
clean = (what) ->
  if what is 'House'
    'Now cleaning house'
  else
    'Now cleaning everything'

clean 'House'
# 'Now cleaning house'
```

You can continue cleaning the house while a variable `messy` is truthy:

```
messy = true
while messy
  clean 'House'
  messy = false
```

This particular while loop exits the first time around. Without assigning true to the messy variable, the loop won't exit.

Or you can use an `until` statement. Suppose you have a variable `spotless` that is truthy when things are clean. You can use that instead of `while`:

```
spotless = false
until spotless
  clean 'Everything'
  spotless = true
```

This until loop exits the first time around. Without assigning true to the spotless variable, the loop won't exit.

Some things never end, though, and that's what the `loop` keyword is for:

```
loop clean 'Everything'
```

If you run this on the REPL, it will eventually exit with FATAL ERROR: JS Allocation failed - process out of memory.

Loops are expressions, so they have a value, and if you have a loop that terminates, you can get the value:

```
x = 0
evenNumbers = while x < 6
  x = x + 1
  x * 2

evenNumbers
# [2, 4, 6, 8, 10, 12]
```

Most likely you won't use these looping constructs in CoffeeScript very often. They are there if you need them, though.

Finally, if you really need to, you can get out of a `while`, `until`, or `for` by using the `break` keyword. If you aren't familiar with the `break` keyword, then happily move on.

Exception blocks

These are provided by the `try`, `catch`, and `finally` keywords. Use them to deal with *exceptional* circumstances inside a block of code. You should use the `finally` block to clean up after yourself. Exceptions are created with the `throw` statement. A `try` can also have a `catch`, a `finally`, or both:

```
flyAway = (animal) ->
  if animal is 'pig'
    throw 'Pigs cannot fly'
  else
    'Fly away!'

peter = 'pig'
try
  flyAway peter
catch error
  error
finally
  'Clean up!'
```

Of course, pigs don't fly, so any attempt to make one fly is an exceptional circumstance. In this example, if the `animal` is `'pig'`, then the `catch` block is evaluated, resulting in `'Pigs cannot fly'`.

A `try...catch` also works as an expression. If no exception is thrown, then the value of the `try` is the value of the `try` expression:

```
charlotte = 'spider'
whatHappened = try
  flyAway charlotte
catch error
  error

whatHappened
# Fly away!
```

On the other hand, if an exception is thrown, then the value of the entire expression is the value of the `catch`:

```
whatHappened = try
  flyAway peter
catch error
  error

whatHappened
# Pigs cannot fly
```

Any variables assigned in a `try`, `catch`, or `finally` will be defined:

```
try definedOutsideTheTry = true
definedOutsideTheTry
# true
```

Finally, while it is possible, as shown here, to write a `try` without a `catch`, it's generally not a good idea to ignore exceptions. Deal with them.

INLINE BLOCKS

These are provided by the `then` keyword immediately after an `if`, `case`, or `catch` keyword to supply an expression as a block without a newline or indentation:

```
year = 1983
if year is 1983 then hair = 'perm'

hair
# 'perm'
```

Or for a `while`:

```
while messy then clean 'Everything'
```

Or inside a `switch`:

```
lastDigit = 4
daySuffix = switch lastDigit
  when 1 then 'st'
  when 2 then 'nd'
  when 3 then 'rd'
  else 'th'
```

An inline `then` is useful when the expression is small:

```
time = 15
allowed = if time < 12 then 'Yes' else 'No!'
allowed
# 'No!'
```

SUFFIX IF

An `if` statement can also go after an expression:

```
hair = 'permed' if year is 1983
```

Putting the `if` statement after the expression is more readable in some circumstances, so you should decide which version to use based on context.

2.4.3 Pure statements

If a statement isn't used as part of an expression but is used only to tell the computer to do something, then that's a *pure statement*. Some statements in JavaScript are only for control flow—there's no way for CoffeeScript to provide an expression version of those statements.

BREAK, CONTINUE, RETURN

The `break`, `continue`, and `return` statements can't be used as expressions. You might occasionally need `return`, but you should avoid `break` and `continue`. You might see this common idiom in other programming languages for reading line by line:

```
loop
  line = reader.readLine()
  if not line then break
```

Loop over the lines of the file until there are no lines left.

Although it's possible to do that, it's not how you do things in CoffeeScript. In general, avoid using `break`, `continue`, or `return`. Use expressions instead.

2.4.4 Exercise

Suppose you have a variable `animal` that contains a string with the singular name for one of the animals: antelope, baboon, badger, cobra, or crocodile. Write some code to get the collective name for the animal. The collective names for the possible animals in the same order are herd, rumpus, cete, quiver, and bask:

```
animal = 'crocodile'
# <rest of answer goes here>
collective
# bask
```

2.5 Strings

It's possible to write CoffeeScript programs using only expressions containing the literal values already shown, but other language features and libraries provide convenient ways to do common tasks. Every program deals with text at some point, so it's useful to have some more string tools in your string toolbox. This section demonstrates some of the built-in string methods from JavaScript that are useful in CoffeeScript. This section also introduces string interpolation, which provides an elegant way to use variables inside strings.

2.5.1 Methods

CoffeeScript strings have all of the same built-in methods as JavaScript strings. Here are some of the most useful string methods.

SEARCHING

Use the `search` method on a string to find another string within:

```
'haystack'.search 'needle'
# -1

'haystack'.search 'hay'
# 0

'haystack'.search 'stack'
# 3
```

The number returned by `search` is the index in the string at which the match starts. If it returns `-1`, then no match was found. If it returns `0`, the match starts at the beginning of the string.

Suppose you have all of the coffee drinks you serve containing milk in a string:

```
'latte,mocha,cappuccino,flat white,eiskaffee'
```

How do you write a new `hasMilk` function to use instead of a `switch`?

```
milkDrinks = 'latté,mocha,cappuccino,flat white,eiskaffee'

hasMilk = (style) ->
  milkDrinks.search(style) isnt -1

hasMilk 'mocha'
# true

hasMilk 'espresso romano'
# false
```

REPLACING

You use the `replace` method on a string when you want to replace one substring with another:

```
'haystack'.replace 'hay', 'needle'
# 'needlestack'
```

Suppose you want to fix the spelling of a coffee drink:

```
milkDrinks.replace 'latté', 'latte'
```

UPPERCASE AND LOWERCASE

There's a convenient way to convert a string to either all lowercase or all uppercase:

```
'Cappuccino'.toLowerCase()
# 'cappuccino'

'I am shouting!'.toUpperCase()
# 'I AM SHOUTING!'
```

SPLITTING

Use `split` when you want to split a string into an array of strings. You can split a string on the comma character using the `/,/` regular expression literal:

```
'Banana,Banana'.split /,/
# [ 'Banana', 'Banana' ]

'latte,mocha,cappuccino,flat white,eiskaffee'.split /,/
# [ 'latte', 'mocha', 'cappuccino', 'flat white', 'eiskaffee' ]
```

That's enough string methods for now. On to something that JavaScript doesn't have.

2.5.2 *Interpolation*

Suppose you're displaying a web page to a user, and you want to include their name in the web page. You have the name in a variable:

```
userName = 'Scruffy'
```

Use interpolation. Provided by `#{}` inside double-quoted string literals, interpolation injects values into a string:

```
"Affirmative, Dave. I read you."
```

Use string interpolation to replace *Dave* with the actual username:

```
"Affirmative, #{userName}. I read you."
```

You might do the same with a style of coffee:

```
coffee = 'Ristresso'
"Enjoy your #{coffee}!"
# 'Enjoy your Ristresso!'
```

Without interpolation you'd have to add strings, which is tedious:

```
"Affirmative," + userName + ". I read you."
```

Imagine that you want to write a program that displays the string "Hi, my name is Scruffy. Today is Tuesday," where Tuesday is replaced with the current day of the week:

```
userName = 'Scruffy'

dayOfWeek = new Date().getDay()

dayName = switch dayOfWeek
  when 0 then 'Sunday'
  when 1 then 'Monday'
  when 2 then 'Tuesday'
  when 3 then 'Wednesday'
  when 4 then 'Thursday'
  when 5 then 'Friday'
  when 6 then 'Saturday'

"Hi, my name is #{userName}. Today is #{dayName}."
```

Use the getDay method of a date to get the day as a number.

Switch on the number to get the name of the day.

Use string interpolation to display the message.

2.5.3 Exercise

Get the collective animal name to be output in a string like the following:

```
"The collective of cobra is quiver"
```

2.6 Arrays

An *array* is an ordered set of values where a particular value is retrieved using the index of the value in the array. So far you've seen array literals in the form `[1,2,3]`. There are some features of arrays in CoffeeScript that you need to know, in particular, ranges and comprehensions. Just as there are features that make working with strings easier, there are features that make working with arrays easier.

Items in an array are accessed in order by using square brackets, with the first item being `0`:

```
macgyverTools = ['Swiss Army knife', 'duct tape']
macgyverTools[0]
# 'Swiss Army knife'
```

```
macgyverTools[1]
# 'duct tape'
```

This section covers the basic use of arrays, how to transform them, how to extract values from them, and how to comprehend their contents.

2.6.1 *length, join, slice, and concat*

It's now time to explore some built-in properties and methods from JavaScript for arrays that you'll commonly need. None of them modify the original array.

LENGTH

All arrays have a `length` property that returns one greater than the index of the last item in the array:

```
fence = ['fence pail', 'fence pail']
fence.length
# 2
```

An item at any position in an array will affect the length of that array:

```
fence[999] = 'fence pail'
fence.length
# 1000
```

JOIN

Use `join` to convert an array into a string. It takes a string to use as the joining text between each item:

```
['double', 'barreled'].join '-'
# 'double-barreled'
```

SLICE

Use `slice` to extract part of an array:

```
['good', 'bad', 'ugly'].slice 0, 2
# ['good', 'bad']
```

When you use `slice`, the first number is the start index and the second number is the finish index. The item at the finish index isn't included in the result:

```
[0,1,2,3,4,5].slice 0,1
# [0]

[0,1,2,3,4,5].slice 3,5
# [3,4]
```

CONCAT

Use `concat` to join two arrays together:

```
['mythril', 'energon'].concat ['nitron', 'durasteel', 'unobtanium']
# [ 'mythril', 'energon', 'nitron', 'durasteel', 'unobtanium' ]
```

The array methods described don't modify the existing array:

```
potatoes = ['coliban', 'desiree', 'kipfler']

saladPotatoes = potatoes.slice 2,3
saladPotatoes
# ['kipfler']
```

```
potatoes
# ['coliban', 'desiree', 'kipfler']
potatoes.join 'mayonnnaise'
potatoes
# ['coliban', 'desiree', 'kipfler']
potatoes.concat ['pumpkin']
potatoes
# ['coliban', 'desiree', 'kipfler']
```

Enough potatoes. Time to look at the `in` operator.

2.6.2 *in*

In CoffeeScript the `in` operator has particular meaning for arrays. In JavaScript the `in` operator is used for objects, but in CoffeeScript it's used for arrays (the `of` operator is used for objects). Be mindful of that difference.

Contains

This is provided by `in` for an array. Use it to determine if an array contains a particular value:

```
'to be' in ['to be', 'not to be']
# true
living = 'the present'
living in ['the past', 'the present']
# true
```

Suppose you split a string of beverages containing milk into an array:

```
milkBeverages = 'latte,mocha,cappuccino'.split /,/
```

The `in` operator shows if a particular beverage is present:

```
'mocha' in milkBeverages
```

2.6.3 *Ranges*

Ranges are provided by two or three dots between two numbers. Use a range when you need a short way of expressing an array containing a sequence of numerical values. Use two dots to include the upper bound:

```
[1..10]
# [ 1,2,3,4,5,6,7,8,9,10 ]
[5..1]
# [ 5,4,3,2,1 ]
```

Use three dots to exclude the upper bound:

```
[1...10]
# [ 1,2,3,4,5,6,7,8,9 ]
```

Range extraction also provides an alternative to the `slice` method for getting part of an array:

```
['good', 'bad', 'ugly'][0..1]
# ['good', 'bad']
```

2.6.4 Comprehensions

Comprehensions provide a way to look at the array of things (such as ingredients in a recipe) and to manipulate the values without having to use loops. CoffeeScript provides a rich set of comprehensions that can apply to either arrays or objects.

FOR...IN... COMPREHENSION

Array comprehensions allow you to evaluate an expression for each item in an array. Here's a one-line comprehension that's easy to experiment with on the REPL:

```
number for number in [9,0,2,1,0]
# [9,0,2,1,0]
```

Using the name `number` in the comprehension declares it as a variable. You can use any variable name you like:

```
x for x in [9,0,2,1,0]
# [9,0,2,1,0]
```

However, it is best to use a different variable name just for the constructor (you'll learn more about why later on). Now use a comprehension to add 1 to every item in the array:

```
number + 1 for number in [9,0,2,1,0]
# [10,1,3,2,1]
```

Use a comprehension to convert every item to a 0:

```
0 for number in [9,0,2,1,0]
# [0,0,0,0,0]
```

The name after the `for` keyword in a comprehension creates a variable with that name. It's possible to access the variable outside of the comprehension:

```
letter for letter in ['x','y','z']
# [x,y,z]

letter
# 'z'
```

But it's a very bad idea to do so. Leave comprehension variables in the comprehensions where they belong.

USING COMPREHENSIONS

Imagine you're making a chocolate cake. You have the ingredients supplied as an array of strings:

```
ingredients = [
  'block of dark chocolate'
  'stick butter'
  'cup of water'
  'cup of brown sugar'
  'packet of flour'
  'egg'
]
```

Suppose you want to make a cake that's twice as big. Make a new ingredients list that puts *2x* in front of all of the ingredients:

```
doubleIngredients = ("2x #{ingredient}" for ingredient in ingredients)
```

```
doubleIngredients
# [
#   '2x block of dark chocolate'
#   '2x stick butter'
#   '2x cup of water'
#   '2x cup of brown sugar'
#   '2x packet of flour'
#   '2x egg'
# ]
```

How do you mix all these ingredients? Suppose you have a `mix` function:

```
mix = (ingredient) ->
  "Put #{ingredient} in the bowl"
```

Invoke it for each item in the array:

```
instructions = (mix ingredient for ingredient in doubleIngredients)
```

Here, the function `mix` is invoked with the value of each ingredient and the result of all that is assigned to the instructions variable that now references an array:

```
[
  'Put 2x block of dark chocolate in the bowl'
  'Put 2x stick butter in the bowl'
  'Put 2x cup of water in the bowl'
  'Put 2x cup of brown sugar in the bowl'
  'Put 2x packet of flour in the bowl'
  'Put 2x egg in the bowl'
]
```

Notice the absence of loops. Comprehensions can simplify your code. Remember the `switch` statement from listing 2.1?

```
hasMilk = (style) ->
  switch style
    when 'latte', 'cappuccino', 'mocha'
      yes
    else
      no
```

Suppose you have some coffee styles in an array:

```
styles = ['cappuccino', 'mocha', 'latte', 'espresso']
```

Create a new comprehension with the result of invoking `hasMilk` for each item in the array:

```
hasMilk style for style in styles
# [true, true, true, false]
```

You can see Agtron use a comprehension when replying to Scruffy's array of beverages in figure 2.3.

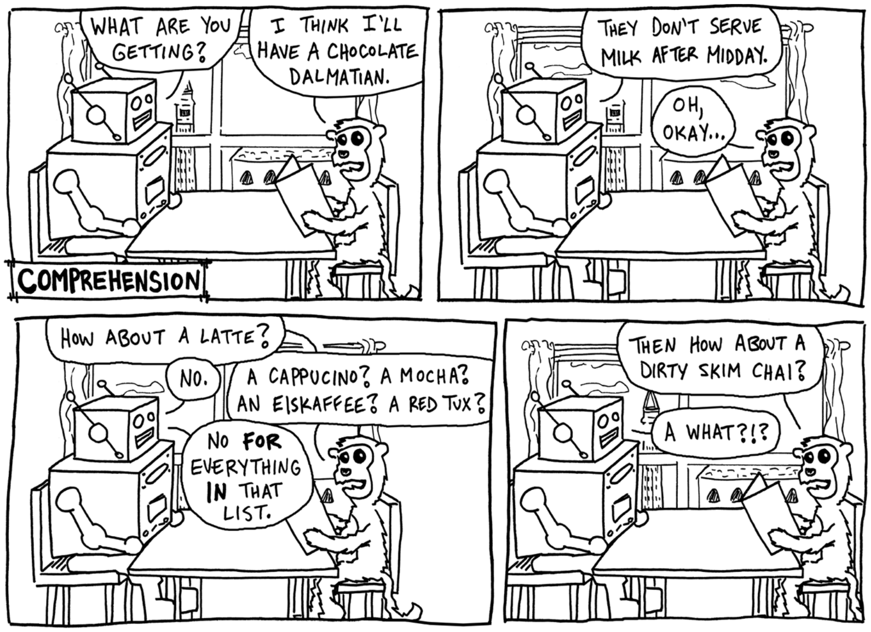

COMPREHENSION
WHAT ARE YOU GETTING?
I THINK I'LL HAVE A CHOCOLATE DALMATIAN.
THEY DON'T SERVE MILK AFTER MIDDAY.
OH, OKAY...
HOW ABOUT A LATTE?
NO.
A CAPPUCINO? A MOCHA? AN EISKAFFEE? A RED TUX?
NO FOR EVERYTHING IN THAT LIST.
THEN HOW ABOUT A DIRTY SKIM CHAI?
A WHAT?!?

Figure 2.3 Don't repeat yourself. Use a comprehension.

THE WHEN COMPREHENSION GUARD

A `when` at the end of a comprehension works like a guard; to make a flourless chocolate cake, you remove the flour from the ingredients:

```
mix = (ingredient) -> "Mixing #{ingredient}"
for ingredient in ingredients when ingredient.search('flour') < 0
  mix ingredient
```

Similarly, to get only the even numbers from a range of numbers, use a `for..in` comprehension with a `when` guard against odd numbers:

```
num for num in [1..10] when not (num%2)
# [ 2, 4, 6, 8, 10]
```

THE BY COMPREHENSION GUARD

Use `by` to perform an array comprehension in jumps. For example, people experimenting with something called polyphasic sleep might sleep every six hours:

```
day = [0..23]
sleep = (hour) -> "Sleeping at #{hour}"
sleep hour for hour in day by 6
# [ 'Sleeping at 0','Sleeping at 6','Sleeping at 12','Sleeping at 18' ]
```

Suppose you want to select every second person in an array; you can use the `by` keyword to do so:

```
person for person in ['Kingpin', 'Galactus', 'Thanos', 'Doomsday'] by 2
# ['Kingpin', 'Thanos']
```

Multiply an array

Suppose you have an array of your lucky numbers:

```
luckyNumbers = [3,4,8,2,1,8]
```

How do you multiply every item in the array by 2? Here's the *wrong* answer:

```
i = 0
twiceAsLucky = []

while i != luckyNumbers.length
  twiceAsLucky[i] = luckyNumbers[i]*2
  i = i + 1
# [1,2,3,4,5,6]

twiceAsLucky
# [6,8,16,4,2,16]
```

Depending on your REPL version, this while loop might even generate REPL output that you don't need!

You can write a more concise solution using a comprehension:

```
number * 2 for number in luckyNumbers
```

Comprehensions help you to write simpler code that matches your intentions without having to worry about intermediate variables and loop counters.

2.6.5 Exercise

Suppose you have a string containing animal names:

```
animals = 'baboons badgers antelopes cobras crocodiles'
```

Write a program to output the following:

```
['A rumpus of baboons',
 'A cete of badgers',
 'A herd of antelopes',
 'A quiver of cobras',
 'A bask of crocodiles']
```

2.7 Heres for comments, docs, and regexes

CoffeeScript provides variants of strings, comments, and regular expression literals that can contain whitespace, such as newlines. All of these are indicated with syntax similar to their nonwhitespace counterparts but have a triple of the character for opening and closing the literal. Because they can contain literal whitespace, heredocs, herecomments, and heregexes are useful where formatting needs to be preserved and also for retaining clarity in code that would be difficult to read if the whitespace wasn't preserved.

2.7.1 Comments

Standard comments use a single # and continue to the end of the line:

```
# This is a comment
```

These standard CoffeeScript comments aren't included in the compiled JavaScript.

HERECOMMENTS

The CoffeeScript block comment called the *herecomment* is included as a block comment in the compiled JavaScript. Start and finish a block comment with three consecutive hashes (###):

```
###
This is a herecomment
It will be a block comment in the generated JavaScript
###
```

This herecomment will appear in the compiled JavaScript as a block comment:

```
/*
This is a herecomment
It will be a block comment in the generated JavaScript
*/
```

2.7.2 *Heredocs*

These are written as literal strings that contain literal whitespace. Use a heredoc when your text maintains whitespace for formatting:

```
'''
This
String
Contains
Whitespace
'''
```

Aside from maintaining whitespace, heredocs work like any other string literal. They can be assigned to a variable:

```
stanza = '''
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?
'''
```

When used with double-quoted strings, heredocs support string interpolation:

```
title = 'Tiny HTML5 document'
doc = """
<!doctype html>
<title>#{title}</title>
<body>
"""

doc
# '<!doctype html>\n<title>Tiny HTML5 document</title>\n<body>'
```

The literal newlines in the heredoc appear as `\n` newline characters in a string when the heredoc is evaluated.

2.7.3 Heregexes

CoffeeScript has the same regular expression support as the underlying JavaScript runtime with regular expression literals contained within single forward slashes:

```
/[0-9]/
```

CoffeeScript also supports a notation for regular expressions containing whitespace such as newlines. These *heregexes* are written between triple forward slashes; they're useful when writing more complicated regular expressions that have a reputation for being impenetrable to understanding:

```
leadingWhitespace = ///
  ^\s\s*  # start and pre-check optimizations for performance
///g
```

Syntax and language features are important, but they don't write programs for you. In the next section, you'll write a toy program and run it in the two environments that will be used the most in this book: web browsers and Node.js.

2.8 Putting it together

To learn a programming language, you need to write programs with it. By looking at a program here, you'll also get context and examples for housekeeping, such as how to run the program once it's written.

Some of the code listings in this section might use techniques that are unfamiliar to you. Those techniques will be clear to you after chapter 3.

2.8.1 Running in a browser

To run CoffeeScript programs in a web browser, you should compile them to JavaScript and then include the JavaScript file in your HTML document. Suppose you have the barista program in a file called barista.coffee. First, go to the command line and use `coffee` to compile the script:

```
> coffee -c barista.coffee
```

This generates a barista.js file that you then include in an HTML document as a script:

```
<!doctype html>
<title>Barista</title>
<body>
<form id='order'>
<input id='request' />
<input type='submit' value ='order' />
</form>
The barista.
<div id='response'></div>
</body>
<script src='barista.js'></script>
</html>
```

If you load that file in your web browser, then the barista.js script is executed. In the following listing you see a browser-based implementation of the barista program. The browser version has the house roast specified at the top of the file.

Listing 2.3 A browser barista (barista.coffee)

```
houseRoast = 'Yirgacheffe'

hasMilk = (style) ->
  switch style.toLowerCase()
    when 'latte', 'cappuccino', 'mocha'
      yes
    else
      no

makeCoffee = (requestedStyle) ->
  style = requestedStyle || 'Espresso'
  console.log houseRoast
  if houseRoast?
    "#{houseRoast} #{style}"
  else
    style

barista = (style) ->
  time = (new Date()).getHours()
  if hasMilk(style) and time > 12 then "No!"
  else
    coffee = makeCoffee style
    "Enjoy your #{coffee}!"

order = document.querySelector '#order'
request = document.querySelector '#request'
response = document.querySelector '#response'

order.onsubmit = ->
  response.innerHTML = barista(request.value)
  false
```

When the order element (a form) is submitted, then evaluate the following function.

Find the parts of the HTML document that you need to interact with. Assign references to them to variables.

Use innerHTML to set the content of the response element to be the order response.

Return false from the function so that the order form doesn't submit.

This program accepts the coffee order from an input field and displays the response in the web page.

2.8.2 Running on the command line

If you run CoffeeScript from a standard install on the command line and provide a CoffeeScript file (such as the one from listing 2.4), then the program in the file will be executed:

```
> coffee 2.4.coffee
You need to specify an order.
```

The output of the program indicates that you need to specify an order. You do that with arguments.

Program arguments

Any Node.js program run on the command line has access to the command-line arguments passed to it via the `process`. The first command-line argument is available at `process.argv[2]`. Suppose the program is invoked as follows:

```
> coffee 2.4.coffee 'Cappuccino'
```

Here, the program `process.argv[2]` is `'Cappuccino'`. What happened to `argv[0]` and `argv[1]`? They're reserved for other properties. The `process.argv[0]` is the runtime (in this case, `coffee`) and the `process.argv[1]` is the filesystem path for the program executed (in this case, the full path to the 2.4.coffee file).

The filesystem module

The other essential task in a Node.js program is to read files. For example, suppose your command-line barista program needs to read the house roast from a file before serving a coffee. To do this in Node, you'll require the filesystem module:

```
fs = require 'fs'
```

At this point you don't need to know much about how the module system works. It's covered in depth in chapter 12. Back to the program, though; in the next listing you see a full implementation of the command-line barista program.

Listing 2.4 A command-line barista

```
fs = require 'fs'

houseRoast = null

hasMilk = (style) ->
  switch style.toLowerCase()
    when "latte", "cappuccino"
      yes
    else
      no

makeCoffee = (requestedStyle) ->
  style = requestedStyle || 'Espresso'
  if houseRoast?
    "#{houseRoast} #{style}"
  else
    style

barista = (style) ->
  time = (new Date()).getHours()
  if hasMilk(style) and time > 12 then "No!"
  else
    coffee = makeCoffee style
    "Enjoy your #{coffee}!"
```

```
main = ->
  requestedCoffee = process.argv[2]
  if !requestedCoffee?
    console.log 'You need to specify an order'
  else
    fs.readFile 'house_roast.txt', 'utf-8', (err, data) ->
      if data then houseRoast = data.replace /\n/, ''
      console.log barista(requestedCoffee)

main()
```

Read the command-line input.

Read the file containing the house roast and set it.

Call the barista program with the command-line argument.

The program in listing 2.4 expects to find a file called house_roast.txt that contains the name of the house roast. Suppose that file contains Yirgacheffe and that it's currently before midday. Here's some sample output:

```
> coffee 2.4.coffee
You need to specify an order.

> coffee 2.4.coffee 'Ristretto'
Enjoy your Yirgacheffe Ristretto!
```

The output you'll get when you invoke the program depends on the order and the time of day; experiment with it and explore how it works. The programs in listings 2.3 and 2.4 use some concepts in CoffeeScript and related to CoffeeScript (such as asynchronous programs and web browsers) that you might not yet fully grasp. That's fine; the following chapters will lead you to a better understanding of these concepts.

2.9 Summary

You've learned a lot of syntax in this second chapter. It was important to immerse you in the syntax so that you could begin to get used to it. You've learned that CoffeeScript makes programs easier to understand by emphasizing expressions, cleaning syntax by removing unnecessary characters, and providing succinct alternatives to some common JavaScript idioms (such as dealing with `null` and `undefined` values). In the next chapter you'll start to really do things with CoffeeScript. The next chapter is about functions.

WEB DEVELOPMENT/JAVASCRIPT

CoffeeScript IN ACTION

Patrick Lee

JavaScript runs (almost) everywhere but it can be quirky and awkward. Its cousin CoffeeScript is easier to comprehend and compose. An expressive language, not unlike Ruby or Python, it compiles into standard JavaScript without modification and is a great choice for complex web applications. It runs in any JavaScript-enabled environment and is easy to use with Node.js and Rails.

CoffeeScript in Action teaches you how, where, and why to use CoffeeScript. It immerses you in CoffeeScript's comfortable syntax before diving into the concepts and techniques you need in order to write elegant CoffeeScript programs. Throughout, you'll explore programming challenges that illustrate CoffeeScript's unique advantages. For language junkies, the book explains how CoffeeScript brings idioms from other languages into JavaScript.

What's Inside

- CoffeeScript's syntax and structure
- Web application patterns and best practices
- Prototype-based OOP
- Functional programming
- Asynchronous programming techniques
- Builds and testing

Readers need a basic grasp of web development and how JavaScript works. No prior exposure to CoffeeScript is required.

Patrick Lee is a developer, designer, and software consultant, working with design startup Canva in Sydney, Australia.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/CoffeeScriptinAction

Free eBook

SEE INSERT

"This book will help you become a CoffeeScript Ninja!"

—Phily Austria, Paystr LLC

"Truly entertaining ... dives deep into CoffeeScript."

—Andrew Broman
University of Wisconsin, Madison

"By far the best resource for learning CoffeeScript or for improving your existing skills."

—John Shea, Endicott College

"Makes learning CoffeeScript fun!"

—Kenrick Chien
Blue Star Software

ISBN 13: 978-1-617290-62-6
ISBN 10: 1-617290-62-9
54499
9 781617 290626

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]