CoffeeScript IN ACTION

Patrick Lee

MANNING

CoffeeScript in Action

by Patrick Lee

Chapter 7

brief contents

7

Style and semantics

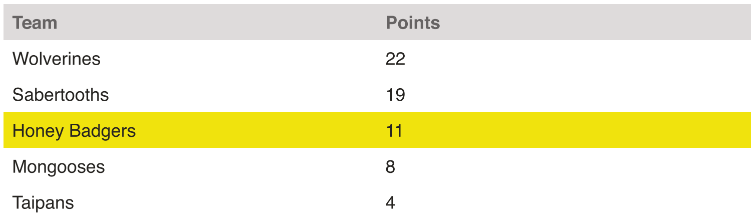
This chapter covers

- Rest and spread parameters
- Destructuring assignment
- Dealing with types
- Effective comprehensions
- Fluent interfaces
- Nuances of significant whitespace

In chapter 2 you took a highway drive through the CoffeeScript syntactic landscape. CoffeeScript is a small language, so the drive didn't take you very long. That said, chapter 2 was deliberately fast-paced, and at the speed you were going, some details were necessarily postponed for later. Well, later has arrived.

To understand the full breadth of CoffeeScript programs you'll find in the wild, you need to appreciate some subtler syntactic and semantic aspects of the language that require a bit more care and attention to grasp. These aspects—covered in this chapter—are spread and rest parameters, destructuring assignment, semantic issues around types and dealing with null values, effective use of comprehensions, fluent interfaces, and finally the nuances of dealing with significant indentation. First up, rest and spread parameters.

Click here to open the source code file for this chapter.



Team	Points
Wolverines	22
Sabertooths	19
Honey Badgers	11
Mongooses	8
Taipans	4

Figure 7.1 Table of teams ranked by points

7.1 Rest and spread parameters

Imagine you're displaying a table of teams in a competition ordered by the number of points they have (figure 7.1). When a team moves up in the rankings, you highlight the row containing that team.

Suppose you already have a `highlight` function that takes a name, finds it, and highlights it by invoking `color` and `find` functions that have been implemented elsewhere:

```
highlight = (name) ->
  color find(name), 'yellow'
```

Suppose now that your colleague Bobby Tables wants to use the `highlight` function with multiple names. He changes it to work with an array:

```
highlight = (names) ->
  for name in names
    color find(name), 'yellow'
```

Unfortunately, by changing the function Bobby has broken all the other places in your program where `highlight` is invoked with a single name. Sure, this might be little more than a minor inconvenience, but it's an inconvenience that you can avoid entirely by using some syntactic sugar called *rest parameters*.

The arguments object

Experienced JavaScript developers will be familiar with the `arguments` object and the problems it poses by not inheriting from the array prototype. The good news is that rest parameters in CoffeeScript mean that you *never* have to use the `arguments` object again.

7.1.1 Rest

The rest parameter puts multiple function parameters into an array.[1] A rest parameter is a name followed by an ellipsis (three dots), and it can be used to create a better `highlight` function:

```
highlight = (names...) ->
  for name in names
    color find(name), 'yellow'
```

[1] If you're familiar with Ruby or Perl, you might be familiar with the splat, which serves the same purpose.

When the `highlight` function is invoked, then `names` is an array containing as many elements as there are arguments:

```
highlight()
highlight 'taipans'
highlight 'taipans', 'wolverines', 'sabertooths'
```

names will be [].

names will be ['taipans'].

names will be ['taipans', 'wolverines', 'sabertooths'].

Suppose now that you want the `highlight` function to color the first team (identified by the first parameter) gold and color all the other teams (identified by the other parameters) blue. How do you do that? By putting the ellipsis on the second parameter:

```
highlight = (first, rest...) ->
  color find first 'gold'
  for name in rest
    color find(name), 'blue'
highlight 'taipans', 'sabertooths', 'wolverines'
```

Taipans will be colored gold; sabertooths and wolverines will be colored blue.

Using rest parameters, you can convert multiple function parameters into a single array. Using *spread* parameters, you can do the reverse—convert a single array into multiple parameters.

7.1.2 Spread

Now that the `highlight` function takes individual arguments, what do you do when you have an array of names? Suppose you have the teams ranked in an array, top team first:

```
teams = ['wolverines', 'sabertooths', 'mongooses']
```

You want to pass the first item (called the *head* of the array) as one parameter and the remaining items (called the *tail*) as the remaining parameters. Doing that manually is clumsy:

```
highlight teams[0], teams[1], teams[2]
```

Your fingers would be sore from typing after a hundred team names or so. The alternative is to *spread* the teams in the array across the function parameters:

```
highlight teams...
```

Compare two different invocations of the `highlight` function:

```
highlight teams
highlight teams...
```

highlight is invoked with the teams array as the first argument.

highlight is invoked with the teams array spread across as many arguments as there are items in the array.

Suppose now that you have an `insert` function that takes a spread, and you want to invoke it from within another function that also takes a spread. Passing the name of the rest parameters directly to `insert` doesn't work:

```
insert = (teams...) ->
  teams
```

For demonstration, this insert function simply returns the names parameter.

```
initialize = (teams...) ->
  insert teams
```

Invoke insert with the name of the rest parameter.

```
initialize 'wolverines', 'sabertooths', 'mongooses'
# [ [ 'wolverines','sabertooths', 'mongooses' ] ]
```

The rest parameter for insert ends up as an array wrapped in an array—definitely not what you wanted. But using the spread you can get the result you need:

```
initialize = (teams...) ->
  insert teams...

initialize 'wolverines', 'sabertooths', 'mongooses'
# [ 'wolverines','sabertooths', 'mongooses' ]
```

Using rest and spread in combination, you can make a flexible and elegant program to highlight the team names in the table. This program appears in the following listing. Note that this listing is intended to run in a web browser; you can use the server from listing 7.5 to achieve that.

Listing 7.1 Competition

```
find = (name) ->
  document.querySelector ".#{name}"
```

The find function uses the native document.querySelector to find elements matching the class name.

```
color = (element, color) ->
  element.style.background = color
```

Set the background color of the element via a style property.

```
insert = (teams...) ->
  root = document.querySelector '.teams'
  for team in teams
    element = document.createElement 'li'
    element.innerHTML = team
    element.className = team
    root.appendChild element
```

Add elements to the page for each team.

```
highlight = (first, rest...) ->
  color find(first), 'gold'
  for name in rest
    color find(name), 'blue'
```

Highlight the first element gold and then the rest blue via a comprehension.

```
initialize = (ranked) ->
  insert ranked...
  first = ranked.slice(0, 1)
  rest = ranked.slice 1
  highlight first, rest...
```

Initialize by inserting the elements and then highlighting.

```
window.onload = ->
  initialize [
    'wolverines'
    'wildcats'
    'mongooses'
  ]
```

Invoke initialize with the array of teams when the browser window loads.

Although rest and spread are only small bits of syntactic sugar (remember, syntactic sugar makes things sweeter), they can make a big difference for code readability, which is a key motivation for many of the syntactic changes that CoffeeScript introduces to JavaScript. Another one of those changes that you haven't yet explored in detail is called *destructuring*.

7.2 Destructuring

Like spread and rest parameters, destructuring assignment allows you to write with a single, terse expression something that would normally require multiple expressions. Think of rest and spread parameters on steroids—destructuring allows you to unpack *any* array or object into variables or arguments.

7.2.1 Arrays

Suppose your program for the competition table needs to keep track of a particular team (such as your favorite team) by making it the active team. Here's a function that Bobby Tables has already written to switch this active team:

```
makeToggler = (active, inactive) ->
  ->
    temporary = active
    active = inactive
    inactive = temporary
    [active, inactive]

toggler = makeToggler 'komodos', 'raptors'

toggler()
# ['raptors', 'komodos']

toggler()
# ['komodos', 'raptors']
```

Notice how three variables are needed to perform the switch.

That doesn't seem very concise, does it? What's the alternative? With destructuring you can do away with so many expressions by assigning values to multiple variables in a single expression:

```
active = 'komodos'
inactive = 'raptors'

[active, inactive] = [inactive, active]

active
# raptors

inactive
# komodos
```

Destructuring assignment swaps the variables.

Rewrite the `toggle` function to use array destructuring:

```
toggler = (a, b) ->
  -> [a,b] = [b,a]

toggle = toggler 'on', 'off'

toggle()
# ['off', 'on']

toggle()
# ['on', 'off']
```

Suppose that in another part of the team competition program you need to relegate the team that finishes last. By using array destructuring you can avoid multiple and confusing array shuffling and variable assignments:

```
relegate = (team) -> "#{team.name} got relegated"

rank = (array..., using) ->
  array.sort (first, second) ->
    first[using] < second[using]

competitors = [
    name: 'wildcats'
    points: 5
  ,
    name: 'bobcats'
    points: 3
]

[first, field..., last] = rank competitors..., 'points'

relegate last
# 'bobcats got relegated'
```

For demonstration, just display the name of the team that got relegated.

Rank an array on a specific property.

You can omit the curly braces in an array of objects only if you comma-separate them and indent with care.

Rank the competitors array on the points property.

This gets you the same result but without shuffling values around multiple variables. That's a good result because every place you shuffle another variable or create a new bit of state is a place where you might introduce an error. This doesn't just apply to arrays—objects can also be destructured.

7.2.2 Objects

Imagine that you have some data for all of the teams in the competition. All of the data you need is inside an object, but it's not exactly in the structure you want. Here's the current structure:

```
data =
  team2311:
    name: 'Honey Badgers'
    stats:
      scored: 22
      conceded: 22
      points: 11
  team4326:
    name: 'Mongooses'
    stats:
      scored: 14
      conceded: 19
      points: 8
```

Instead of the current structure, how do you get at the specific parts of the data structure that you need? You might pull the object apart manually:

```
for id, team of data
  name = team.name
  points = team.stats.points
  {
    name: name
    points: points
  }
# [ { name: 'Honey Badgers', points: 11 },
#   { name: 'Mongooses', points: 8 } ]
```

That's no fun. With destructuring you can do it succinctly:

```
for id, team of data
  {name: team.name,  points: team.stats.points}
# [ { name: 'Honey Badgers', points: 11 },
#   { name: 'Mongooses', points: 8 } ]
```

One place where you see this object destructuring used frequently is at the start of a file in a Node.js program:

```
{pad, trim, dirify} = require 'util'
```

Why? Because the `require` function returns an object, and manually unpacking the object is tedious:

```
util = require 'util'
pad = util.pad
trim = util.trim
dirify = util.dirify
```

Exactly *how* the module system works is covered in detail in chapter 12, but what you can see already is how destructuring is used to remove noise from a program and so helps you to make it simpler.

You're not yet finished with making things simpler, though; CoffeeScript has another bit of syntactic sugar for objects: the object shorthand.

7.2.3 Object shorthand

Consider a `competition` module that exposes a `highlight` and an `initialize` method:

```
makeCompetition = ->
  find = ->
    # function body omitted
  color = ->
    # function body omitted
  highlight = ->
    # function body omitted
  initialize = ->
    # function body omitted

  highlight: highlight,
  initialize: initialize
```

Expose only the highlight and initialize functions as public by assigning them to properties of the returned object. Remember, you can leave the squigglies (curly braces) off when writing object literals in CoffeeScript.

```
competition = makeCompetition()
# { highlight: [Function], initialize: [Function] }
```

Here the variable names are the same as the property names on the object, so it's repetitive to write both:

```
highlight = ->
initialize = ->

object =
  highlight: highlight
  initialize: initialize
```

Repetitive

Instead of being repetitive, the object shorthand means you can pack variable values back into an object based on their names:

```
makeCompetition = ->
  find = ->
  color = ->
  highlight = ->
  initialize = -> 'initialized'

  {highlight, initialize}

competition = makeCompetition()
competition.initialize()
# 'initialized'
```

Use object shorthand to export the value of the variable with the name.

The object shorthand also works when specifying parameters. Suppose `makeCompetition` accepts an `options` argument containing the `maxCompetitors` and the `sortOrder`:

```
makeCompetition = (options) ->
  options.max
  options.sort
```

Although an `options` argument saves you from having an unwieldy number of arguments to the function, the end result is often repetitive regardless. You can avoid that by using object destructuring directly in the parameter list. An object in the parameter definition will result in the object properties being destructured to individual named parameters:

```
makeCompetition = ({max, sort}) ->
  {max, sort}
```

This is very handy; you now essentially have named parameters that can be supplied in any order without needing an intermediate `options` parameter:

```
makeCompetition max: 5, sort: ->
# {max: 5, sort: [Function]}

makeCompetition sort: (->), max: 5
# {max: 5, sort: [Function]}
```

The named arguments can go in any order and the effect will be the same.

Although this is useful, you should think carefully before using destructuring and make sure you aren't making your program hard to read. These examples and the issue of readability are revisited later in this chapter.

7.2.4 Array destructuring expansion

Since CoffeeScript version 1.7, array destructuring also works with expansion. What does that mean? Suppose you have an array of competitors:

```
competitors = [
  { name: 'wildcats', points: 3 }
  { name: 'tigers', points: 1 }
  { name: 'taipans', points: 5 }
]
```

How do you get just the first team and the last team? Prior to CoffeeScript 1.7 you needed to name the `first`, `last`, and `middle`:

```
[first, middle..., last] = competitors
first
# { name: 'wildcats', points: 3 }

last
# { name: 'taipans', points: 5 }
```

Since 1.7, though, you can elide the `middle` and the result will be the same:

```
[first, ..., last] = competitors
first
# { name: 'wildcats', points: 3 }

last
# { name: 'taipans', points: 5 }
```

This is another small way in which CoffeeScript syntax can help you focus on just the things that matter to you.

7.2.5 Exercises

Got the knack for destructuring? Try these exercises to find out:

- Given an array of numbers such as `[1,2,3,4,5,6]`, write a function that uses destructuring and a comprehension to reverse each subsequent pair of numbers in the array so that, for example, `[1,2,3,4,5,6]` becomes `[2,1,4,3,6,5]` and `[1,2,1,2,1,2]` becomes `[2,1,2,1,2,1]`.
- Suppose you've received some JSON containing a phone directory in a format like this:

  ```
  {"A":[{"name":"Andy", "phone":"5551111"},...],"B":[...],...}
  ```

  Write a function that produces the last phone number for a given letter found in the phone directory.

7.2.6 Putting it together

Now that you've completed some exercises, it's time to see this new syntax in a program. Rest and spread parameters and destructuring techniques for arrays and objects are demonstrated in the competition program of the following listing. This listing is intended to run in a web browser; you can use the server from listing 7.5 to achieve that.

Listing 7.2 Competition with module pattern

```
makeCompetition = ({max, sort}) ->
  find = (name) ->
    document.querySelector ".#{name}"

  color = (element, color) ->
    element.style.background = color

  insert = (teams...) ->
    root = document.querySelector '.teams'
    for team in teams
      element = document.createElement 'li'
      element.innerHTML = "#{team.name} (#{team.points})"
      element.className = team.name
      root.appendChild element

  highlight = (first, rest...) ->
    color find(first.name), 'gold'
    for team in rest
      color find(team.name), 'blue'

  rank = (unranked) ->
    unranked.sort(sort).slice(0, max)

  initialize = (unranked) ->
    ranked = rank unranked
    insert ranked...
    first = ranked.slice(0, 1)[0]
    rest = ranked.slice 1
    highlight first, rest...

  { initialize }

sortOnPoints = (a, b) ->
  a.points > b.points

window.onload = ->
  competition = makeCompetition(max: 5, sort: sortOnPoints)
  competition.initialize [
    { name: 'wolverines', points: 22 }
    { name: 'wildcats', points: 11 }
    { name: 'mongooses', points: 33 }
    { name: 'raccoons', points: 12 }
    { name: 'badgers', points: 19 }
    { name: 'baboons', points: 16 }
  ]
```

A maker function that returns a competition object. The arguments are named by using object syntax in the function declaration.

Rank the competitors using the sort method on the array prototype.

Return an object with an initialize property and the value of the initialize variable as the value. This is the shorthand syntax for {initialize: initialize}.

The sorting strategy for the teams that will be given to the competition; it sorts on points.

Pass in object as named parameters to the maker function.

CoffeeScript mixes syntax and semantics from JavaScript (both current and future versions) as well as from other programming languages. Some things (such as destructuring) are very useful, whereas other things are (or can be) less useful. Take `null`, for example.

7.3 No nulls

When a variable or property is defined but doesn't have a value, then it has the special value `null`. The value `null` is the only value with `typeof null` in CoffeeScript. You might ask what `null` is good for—that's a good question. You see, the `null` value isn't a value you want. Instead, it usually means something has gone wrong. Remember the existential operator used to determine if a variable is either `undefined` or `null`?

```
roundSquare?
# false
```

Well, this existential operator can also be combined with the dot operator and the assignment operator. In this section you'll learn about these additional uses of the existential operator and how they help you live with `null`.

7.3.1 Null soak

Imagine you're writing an application that takes information about users and displays it on a web page. The information for a given user is in an object:

```
user =
  name:
    title: 'Mr'
    first: 'Data'
    last: 'Object'
  contact:
    phone:
      home: '555 2234'
      mobile: '555 7766'
    email:
      primary: 'mrdataobject@coffeescriptinaction.com'
```

How do you access the home phone number for this user?

```
user.contact.phone.home
# '555 2234'
```

Suppose that some of the user data is missing or incomplete. For example, consider a user who doesn't have any contact information:

```
user =
  name:
    first: 'Haveno'
    middle: 'Contact'
    last: 'Details'
```

Your program will throw an exception:

```
user.contact.phone.home
# TypeError: cannot read property 'phone' of undefined
```

To avoid this, you *might* wrap the access to the data in a big condition:

```
if user.contact and user.contact.phone and user.contact.phone.home
  user.contact.phone.home
```

Or perhaps you might wrap it in a `try` block:

```
try
  user.contact.phone.home
catch e
  'no contact number'
```

Don't do either. Instead, when rendering information about a user, either display the information or not—don't display an error, `null`, or `undefined`. Use the null soak operator (sometimes called the *safe navigation operator*) to soak `null` values and `undefined` properties so you don't have to test for them explicitly:

```
user?.contact?.phone?.home
```

This will suppress any errors when accessing the user data. An acceptable time to suppress these errors is when rendering data to a user. For example, suppose you have a `render` function that uses a heredoc with interpolation to display the phone number somewhere on a website:

```
render = (user) ->
  """
  <html>
  Home phone for #{user.name.first}: #{user.contact.phone.home}
  """
```

Imagine this `render` function is normally invoked when information about a user is retrieved. Without using the null soak operator, the `render` function will cause an error when a user property is missing:

```
user =
  name:
    first: 'Donot'
    last: 'Callme'

render user
# TypeError: cannot read property 'phone' of undefined
```

You don't want a `render` function to show an error to a user when data is missing. Instead, you want to *suppress* internal errors when rendering. Think about it: users don't want to read all your internal errors. Avoid showing errors by using the null soak:

```
render = (user) ->
  """
  <html>
  Home phone for #{user?.name?.first}: #{user?.contact?.phone?.home}
  """

user = null
render user: null
# <html>
# Home phone for undefined: undefined
```

To display something other than `undefined`, you can use the default operator `||` to specify a default value:

```
user = null
contact = user?.contact?.phone?.home || 'Not provided'
contact
# Not provided
```

This allows you to present the information you need without an explicit conditional.

Null soak is not for assignment

Because null soak is so convenient, it's tempting to use it with assignment:

```
user = {}
user?.contact?.phone?.home = '555 5555'
```

That's a *bad* idea. What's the result of the assignment? Is the value of `user.contact.phone.home` now `5`? No. The `null` was soaked immediately and `user` doesn't even have a `contact` property:

```
user.contact?
# false
```

Don't use null soak for property assignment. Only use it for *safe access*. Conditional assignment, on the other hand, can be useful and safe for local variables.

7.3.2 Conditional assignment

Suppose you want to assign a value to a variable only if it does *not* already contain a value. A simple way to do that is by combining the existential operator with assignment:

```
phone = undefined
phone ?= '555 5555'
phone
# '555 5555'

phone = null
phone ?= '555 1111'
phone
# '555 1111'

phone ?= 'something else'
phone
# '555 1111'
```

Be careful, though; existential assignment isn't for variables that haven't been defined:

```
variableYouNeverDeclared ?= 'something'
# error: the variable "variableYouNeverDeclared" can't be assigned with ?=
     because it has not been declared before
```

In the next listing you once again see the competition rankings program. This time the null soak is used to handle `null` values in the supplied data and when rendering an HTML view. This listing is intended to run in a web browser; you can use the server from listing 7.5 to achieve that.

Listing 7.3 Using null soak in a view

```
makeCompetition = ({max, sort}) ->
  render = (team) ->
    """
    <tr class='#{team?.name||''}'>
    <td>#{team?.name||''}</td>
    <td>#{team?.points||''}</td>
    <td>#{team?.goals?.scored||''}</td>
```

The view is just a lo-fi function called render. Notice how the nulls are soaked in expressions such as team?.goals?.scored and that the default operator is used to output an empty string as the default value.

```
      <td>#{team?.goals?.conceded||''}</td>
      </tr>
      """

    find = (name) ->
      document.querySelector ".#{name}"

    color = (element, color) ->
      element.style.background = color

    insert = (teams...) ->
      root = document.querySelector '.teams'
      for team in teams
        root.innerHTML += render team

    highlight = (first, rest...) ->
      color find(first.name), 'gold'
      for team in rest
        color find(team.name), 'blue'

    rank = (unranked) ->
      unranked.sort(sort).slice(0, max).reverse()

    initialize: (unranked) ->
      ranked = rank unranked
      insert ranked...
      first = ranked.slice(0, 1)[0]
      rest = ranked.slice 1
      highlight first, rest...

  sortOnPoints = (a, b) ->
    a.points > b.points

  window.onload = ->
    competition = makeCompetition max:5, sort: sortOnPoints
    competition.initialize [
        name: 'wolverines'
        points: 56
        goals:
          scored: 26
          conceded: 8
      ,
        name: 'wildcats'
        points: 53
        goals:
          scored: 32
          conceded: 19
      ,
        name: 'mongooses'
        points: 34
        goals:
          scored: 9
          conceded: 9
      ,
        name: 'raccoons'
        points: 0
    ]
```

The competition is initialized as before except that now the data has information about the goals scored by individual teams.

The raccoons were disqualified so they have zero points and don't have a goals property.

Remember what the type of `null` is?

```
typeof null
# null
```

There's only one thing with a type of null and it's the `null` value. As mentioned previously, there are languages with rich and powerful type systems. CoffeeScript isn't one of them, and types, including the null type, can be problematic.

7.4 No types—the duck test

CoffeeScript is dynamically and weakly typed. The most noticeable thing about the `typeof` operator so far in your CoffeeScript travels should be its absence. When writing programs in CoffeeScript, there's rarely any benefit to be gained from examining types by using the `typeof` operator. Instead, use a technique called duck typing:

```
class Duck
  walk: ->
  quack: (distance) ->

daffy = new Duck
```

What can you do with a duck? You might put it on a leash and take it for a walk:

```
daffy.walk()
```

If it meets another duck, they might talk to each other:

```
donald = new Duck

donald.quack()
daffy.quack()
```

That's great. Suppose you want to organize a duck race:

```
class DuckRace
  constructor: (@ducks) ->
  go: ->
    duck.walk() for duck in @ducks
```

This `DuckRace` doesn't know if only ducks are competing. For example, a faster animal such as a hare could enter the race:

```
class Hare
  run: ->
  walk: -> run()

hare = new Hare

race = new DuckRace [hare]
```

That's unfair! How will you prevent non-ducks from entering the race? If you have experience in a strongly typed language, then you might think that the `typeof` operator will do the trick.

7.4.1 Don't rely on typeof, instanceof, or constructor

You want to be strict with entrants to the DuckRace and ensure that they're all real, certified ducks, and your first thought is to use the typeof operator. Unfortunately, the typeof operator in CoffeeScript isn't very useful:

```
daffy = new Duck
typeof daffy
# 'object'
```

Almost everything in CoffeeScript is an object, so there's really no point using typeof here. So, you think that perhaps you can test to see if the object was created from the Duck class by using the instanceof operator:

```
daffy instanceof Duck
# true
```

Great, that's a duck. Is instanceof the solution? No. What happens when you change the DuckRace constructor to admit any objects that are instanceof Duck?

```
class DuckRace
  constructor (applicants) ->
    @ducks = d for d in applicants when d instanceof Duck
  go: ->
    duck.walk() for duck in @ducks
```

The race is run. Unfortunately, one of the ducks isn't really a duck. It was a duck, but it was turned into a snake by an evil warlock:

```
duck = new Duck
ultraDuckMarathon = new DuckRace [duck]

turnIntoSnake = ->
  duck.walk = null
  duck.slither = ->

turnIntoSnake duck

ultraDuckMarathon.go()
# TypeError: Property walk of object #<AsianDuck> is not a function
```

The walk property of this duck is assigned the value null. It no longer knows how to walk.

You tried to use a type to solve your problem (via the instanceof operator) and the result was a type error. Irony.

The instanceof operator doesn't promise anything. It doesn't guarantee that an object has a particular interface or works a particular way. It doesn't tell you if something is a duck. All it tells you is which class or constructor the object was created with. In a dynamic language like CoffeeScript, the class or constructor of an object doesn't guarantee anything about what the object actually does right now. The instanceof operator is even more brittle because the result will change if you reassign a prototype:

```
class Duck
daffy = new Duck
Duck:: = class Snake

daffy instanceof Duck
# false
```

As a last resort you decide to use the `constructor` property:

```
class Duck
daffy = new Duck
daffy.constructor.name
# Duck
```

That tells you the constructor for the `daffy` object. Again, though, this is a flawed approach in a dynamic language like CoffeeScript. Suppose you create a duck without using a class:

```
duck =
  walk: ->
  quack: ->

daffy = Object.create duck
```

Is `daffy` a duck? It was created from a prototypical duck. If the `constructor.name` is the criteria for being a duck, then `daffy` isn't a duck:

```
daffy.constructor.name
# 'Object'
```

So, with all of the different techniques for constructing and modifying objects in CoffeeScript, it quickly becomes apparent that none of the approaches you try to determine whether something is the correct type are going to work. What's the alternative then? It's called duck typing.

7.4.2 How to use duck typing

The principle behind duck typing is that if there's no reliable way in the language to be sure what interface an object implements, then you should rely on the interface itself. Put simply, *if it walks like a duck and quacks like a duck, then it's a duck*:

```
class DuckRace
  duck: (contestant) ->
    contestant.quack?.call and contestant.walk?.call
  constructor: (applicants...) ->
    @ducks = (applicant for applicant in applicants when @duck applicant)

duck =
  name: 'Daffy'
  quack: ->
  walk: ->

cow =
  name: 'Daisy'
  moo: ->
  walk: ->

race = new DuckRace duck, cow
race.ducks
# [ { name: 'Daffy', quack: [Function], walk: [Function] } ]
```

If it has a call property, then it can be invoked. See section 6.4.4.

Without types, how do you have confidence that your program works correctly? You get confidence by writing tests. Good tests for the `DuckRace` class will show how it should be used and demonstrate that it works correctly when used as intended. There's more about testing in chapter 10.

Postel's law

Be conservative in what you do; be liberal in what you accept from others.

Duck typing means you're liberal in what you accept. What you accept only has to adhere to an interface.

In listing 7.4 you see a new version of the competition program. This time teams can be added dynamically. The competition organizes a tournament between teams where the winner of each game is determined randomly. Without relying on a type system, any object that can't act like a team for the purposes of the competition is excluded. This listing is intended to run in a web browser; you can use the server from listing 7.5 to achieve that.

Listing 7.4 Competition

```
makeCompetition = ({max, sort}) ->

  POINTS_FOR_WIN = 3
  POINTS_FOR_DRAW = 1
  GOALS_FOR_FORFEIT = 3
```

Uppercase constants as convention, but not really constants (see chapter 13 for more on constants and the const keyword in ECMAScript 6).

```
  render = (team) ->
  find = (name) ->
  color = (element, color) ->
  insert = (teams...) ->
  highlight = (first, rest...) ->
  rank = (unranked) ->
```

These functions appear in listing 7.3 and are omitted here for brevity.

```
  competitive = (team) ->
    team?.players is 5 and team?.compete()?
```

Duck type. Determine if a team has enough players and competes.

```
  blankTally = (name) ->
    name: name
    points: 0
    goals:
      scored: 0
      conceded: 0
```

Provide a blank object to tally scores on.

The roundRobin function loops through the teams and plays them all against each other once.

```
  roundRobin = (teams) ->
    results = {}
    for teamName, team of teams
      results[teamName] ?= blankTally teamName
      for opponentName, opponent of teams when opponent isnt team
        console.log "#{teamName} #{opponentName}"
        results[opponentName] ?= blankTally opponentName
        if competitive(team) and competitive(opponent)
          # omitted
        else if competitive team
          # omitted
        else if competitive opponent
          # omitted
```

Code for adding the scores and goals is included in the source. It's omitted here because it's boring and not helpful for understanding the concepts.

```
    results

  run = (teams) ->
    scored = (results for team, results of roundRobin(teams))
    ranked = rank scored
    console.log ranked
    insert ranked...
    first = ranked.slice(0, 1)[0]
    rest = ranked.slice 1
    highlight first, rest...

  { run }
```

Convert the object into an array of its values.

```
sortOnPoints = (a, b) ->
  a.points > b.points

class Team
  constructor: (@name) ->
  players: 5
  compete: ->
    Math.floor Math.random()*3
```

The Team class.

```
window.onload = ->
  competition = makeCompetition(max:5, sort: sortOnPoints)

  disqualified = new Team "Canaries"
  disqualified.compete = null
```

A team created with the Team class that doesn't compete. Doesn't meet the interface.

```
  bizarros = ->
  bizarros.players = 5
  bizarros.compete = -> 9
```

A function with the properties added. Does meet the interface.

```
  competition.run {
    wolverines : new Team "Wolverines"
    penguins: { players: 5, compete: -> Math.floor Math.random()*3 }
    injured: injured
    sparrows: new Team "Sparrows"
    bizarros: bizarros
  }
```

A team created with the Team class that does meet the interface.

An object literal that meets the interface.

No typeof is required. Instead, the teams are tested to see whether they have the properties required for something to be considered a team *for the purposes of the competition.* Another function or module elsewhere in the program could have different expectations of what a team is. Types are not built in. Instead, consider the local requirements for objects and use them to determine whether all objects are suitable.

Surely typeof is used sometimes in CoffeeScript?

Yes. The `typeof` operator is sometimes useful for distinguishing between built-in types. If you're writing library code, you may have good reason to determine if something is `typeof function`, `object`, `string`, or `number`. For anything else, steer clear of `typeof`.

Duck typing is a way of thinking. In a dynamic language like CoffeeScript that doesn't enforce types, you don't look at the type but instead look at the actual object you're dealing with. With duck typing you can express programs naturally in CoffeeScript.

Finally, the following listing provides a server that you can use to experiment with the browser code in listings 7.1 through 7.4.

Listing 7.5 The server

```
http = require 'http'
fs = require 'fs'
coffee = require 'coffee-script'

render = (res, head, body) ->
  res.writeHead 200, 'Content-Type': 'text/html'
  res.end """
    <!doctype html>
    <html lang=en>
      <head>
        <meta charset=utf-8>
        <title>Chapter 7</title>
        <style type='text/css'>
        * { font-family: helvetica, arial, sans-serif; }
        body { font-size: 120%; }
        .teams td { padding: 5px; }
        </style>
        #{head}
      </head>
      <body>
        #{body}
      </body>
    </html>
  """

listing = (id) ->
  markup =
    1: """
       <ul class='teams'>
       </ul>"""
    2: """
       <ul class='teams'>
       </ul>"""
    3: """
       <table class='teams'>
         <thead>
           <tr>
             <th>Team</th><th>Points</th><th>Scored</th><th>Conceded</th>
           <tr>
         </thead>
       </table>"""
    4: """
       <table class='teams'>
         <thead>
           <tr>
             <th>Team</th><th>Points</th><th>Scored</th><th>Conceded</th>
```

```
          <tr>
        </thead>
      </table>"""
  script =
    1: "<script src='1.js'></script>"
    2: "<script src='2.js'></script>"
    3: "<script src='3.js'></script>"
    4: "<script src='4.js'></script>"

  head: script[id], body: markup[id]

routes = {}

for n in [1..6]
  do ->
    listingNumber = n
    routes["/#{listingNumber}"] = (res) ->
      render res, listing(listingNumber).head, listing(listingNumber).body
    routes["/#{listingNumber}.js"] = (res) ->
      script res, listingNumber

server = http.createServer (req, res) ->
  handler = routes[req.url] or (res) ->
    render res, '', '''
      <ul>
        <li><a href="/1">Listing 7.1</a></li>
        <li><a href="/2">Listing 7.2</a></li>
        <li><a href="/3">Listing 7.3</a></li>
        <li><a href="/4">Listing 7.4</a></li>
      </ul>
    '''
  handler res

script = (res, listing) ->
  res.writeHead 200, 'Content-Type': 'application/javascript'
  fs.readFile "7.#{listing}.coffee", 'utf-8', (e, source) ->
    if e then res.end "/* #{e} */"
    else res.end coffee.compile source

server.listen 8080, '127.0.0.1'
```

So, what else did you zoom through in earlier chapters? Comprehensions! It's now time to revisit comprehensions and explore their use.

7.5 When to use comprehensions (and when not to)

Comprehensions provide a natural and powerful syntax for dealing with elements in arrays and properties of objects. You'll recognize this simple comprehension for even numbers:

```
evens = (num for num in [1..10] when num%2 == 0)
```

A comprehension is much easier to read than a JavaScript `for` loop:

```
numbers = [1,2,3,4,5,6,7,8,9,10]
evens = []
```

```
for (var i = 0; i !== numbers.length; i++) {
  if(numbers[i]%2 === 0) {
  evens.push(numbers[i]);
}
```

That's not really a fair comparison with JavaScript, though. In recent years it has become more common in JavaScript to use array methods such as `map`, `reduce`, and `filter` instead of `for` loops:

```
evens = numbers.filter(function(item) {
  return item%2 == 0;
});
```

This works in CoffeeScript too:

```
evens = numbers.filter (item) -> item%2 == 0
```

So, if the array methods work in CoffeeScript, should you use them or should you use comprehensions? How do comprehensions compare to these array methods that JavaScript programmers have become more comfortable with? You'll explore that question in this section, learning where comprehensions are appropriate and where they're not.

Careful, the array methods are recent additions

New array methods such as `map` and `filter` are specified in the fifth edition of the ECMAScript specification. Some older web browsers don't support all features of the fifth edition. A compatibility table for ECMAScript 5 features, such as the new array methods, appears in table 13.2.

7.5.1 *map*

The `map` method is used to take an array and map it to a different array. Suppose you purchase a book for $10, a toaster for $50, and a printer for $200. The tax rate is 10%. If you have these prices in an array, how do you calculate the tax paid on each item? With a comprehension, you do this:

```
paid = [10, 50, 200]
taxes = (price*0.1 for price in paid)
# [1, 5, 20]
```

In JavaScript this is done with the array `map` method (`Array::map`). This technique can be expressed directly in CoffeeScript:

```
taxes = paid.map (item) -> item*0.1
# [1, 5, 20]
```

Which one you use is largely a matter of preference.

7.5.2 filter

Suppose you have an array of your friends' addresses, and you want to know which ones live in `'CoffeeVille'`:

```
friends = [
  { name: 'bob', location: 'CoffeeVille' },
  { name: 'tom', location: 'JavaLand' },
  { name: 'sam', location: 'PythonTown' },
  { name: 'jenny', location: 'RubyCity' }
]
```

You can find out with a filter:

```
friends.filter (friend) -> friend.location is 'CoffeeVille'
```

This looks similar when expressed using a comprehension:

```
friend for friend in friends when friend.location is 'CoffeeVille'
```

There's no clear-cut winner. Does this means that comprehensions are overrated? Suppose you have an array of your friends in a variable named `mine`:

```
mine = ['Greg Machine', 'Bronwyn Peters', 'Sylvia Rogers']
```

Consider this to be your set of friends. Now, suppose you spark up a conversation at a party, and you want to know if you have any friends in common with the person you're talking to. Consider the set of their friends to be named `yours`; the solution with a comprehension is elegant:

```
common = (friend for friend in mine when friend in yours)
```

It's no coincidence that this set relationship is elegantly expressed using a comprehension because the syntax for comprehensions is based on a mathematical notation for describing sets. So if you're dealing with sets, comprehensions are a natural fit.

How about the last of the three favorites of JavaScript programmers: `reduce`?

7.5.3 reduce

Imagine you're a loan shark—people owe you money. How do you calculate the total amount you're owed? An array of two people who owe you money is easy to add up just by looking at it or by explicitly adding the two values:

```
friends = [
  { name: 'bob', owes: 10 }
  { name: 'sam', owes: 15 }
]

total = friends[0].owes + friends[1].owes
# 25
```

What if you loaned money to a thousand people? You'd use a comprehension:

```
owed = 0
for friend in friends
  owed += friend.owes
```

The built-in `Array::reduce` also works:

```
owing = (initial, friend) ->
  if initial.owers then initial.owes + friend.owes
owed = friends.reduce owing
```

It isn't clear if the comprehension is better. Comprehensions are useful, but they're not the only technique to consider. Comprehensions also have some gotchas you need to be aware of. The first involves functions.

7.5.4 Defining functions inside comprehensions

When using comprehensions, you need to be careful about scoping because there's a mistake you can easily make even when you know how comprehensions work.[2] In the snippet that follows, what will be the output of the last line? You might be surprised at the answer; try it on your REPL:

```
people = [ 'bill', 'ted' ]
greetings = {}

for person in people
  greetings[person] = ->
    "My name is #{person}"

greetings.bill()
```

Why does the final expression here evaluate to `'My name is ted'`? You see, functions have access to variables via lexical scope regardless of when they're invoked. Here, there's only one `person` variable in scope. Once the comprehension has run, it will contain the last value assigned to it and not the value it had when the function was declared.

If you *really* need to define a function inside a comprehension and have it access some value from inside the comprehension, then you'll need to create a new lexical scope with another function:

```
people = [ 'bill', 'ted' ]
greetings = {}

for person in people
  do ->
    name = person
    greetings[name] = ->
      "My name is #{name}"

greetings.bill()
# My name is bill
```

Now the function(s) assigned to greetings[name] will close over the name variables created for each one. Revisit chapter 3 later if you're still not entirely comfortable with closures and how they close over variables.

In contrast, if you use a `forEach` for this example, then you avoid the problem because you're creating a function scope by default:

```
people.forEach (name) -> greetings[name] = "My name is #{name}"
```

[2] I made one of these mistakes while preparing the listings for this very chapter.

A comprehension that needs a scope probably shouldn't be a comprehension.

A note on generators

Future versions of JavaScript (and by extension CoffeeScript) will include something called *generators* (discussed in chapter 13) that will make comprehensions a more powerful general programming tool. Until that day, though, limit comprehensions to expressing set relationships and use more general-purpose programming constructs, such as functions, elsewhere.

The next piece of syntax used often in JavaScript that needs a discussion in relation to CoffeeScript is the fluent interface. Used by many libraries, including the popular jQuery, the fluent interface is a staple of any JavaScript diet.

7.6 Fluent interfaces

What's a fluent interface? It's a chain of method calls on a single object:

```
scruffy.eat().sleep().wake()
```

It's a bit like a function composition (chapter 6) except that all the function calls act on a particular object. In this section you'll see why fluent interfaces are useful, how to create them, the issues with indentation and side effects, and finally how to create a fluent wrapper for an object that wasn't designed to have one.

7.6.1 Why create them?

Imagine you're helping Scruffy create some animations for an in-browser game called *turt.ly*. He's using an API that Agtron created that allows him to animate the turtle, but he laments that the API is a smidge silly because the class it provides has only four methods: `forward`, `rotate`, `move`, and `swap`:

```
class Turtle
  forward: (distance) ->
    # moves the turtle distance in the direction it is facing
    this
  rotate: (degrees) ->
    # rotates the turtle 90 degrees clockwise
    this
  move: ({direction, distance}) ->
    # moves the turtle in a given direction
    this
  stop: ->
    # stops the turtle
```

The reason for making the methods evaluate to this is explained later in this section.

The `forward` method is invoked with an integer that specifies how far forward the turtle should move. The `rotate` method takes an integer that is the number of degrees that the turtle should rotate *clockwise*—by turning right. To make the turtle walk around a 10 x 10 square, Scruffy has to give it seven commands:

```
turtle = new Turtle
turtle.forward 10
turtle.rotate()
turtle.forward 10
turtle.rotate()
turtle.forward 10
turtle.rotate()
turtle.forward 10
```

When asked about this, Agtron suggests that Scruffy can extend the API by adding a square method to the prototype (see chapter 5):

```
Turtle::square = (size) ->
  @forward size
  @rotate()
  @forward size
  @rotate()
  @forward size
  @rotate()
  @forward size
```

Notice that when adding a square method, you use the @ symbol to invoke a method on the turtle object that the method was invoked on.

Or with brevity:

```
Turtle::square = (size) ->
  for side in [1..4]
    @forward size
    @rotate 90
```

The square method has made making squares easier. But Scruffy still complains that when he wants to draw two squares he has to keep “saying” *turtle*:

```
turtle = new Turtle
turtle.square 4
turtle.forward 8
turtle.square 4
```

He says this is how his math teacher used to talk to him:

> Scruffy, pay attention.
> Scruffy, stop monkeying around.
> Scruffy, go stand outside.
> Scruffy, go to the principal’s office.

How can you avoid sounding like Scruffy’s math teacher? In JavaScript you might be tempted to use the with statement:

```
turtle = new Turtle();

with(turtle) {
  left();
  forward();
}
```

This example is JavaScript.

But the `with` statement hides variable scope and makes your program ambiguous:

This example is JavaScript.

```
address = '123 Turtle Beach Road';
with (turtle) {
  rotate(90);
  address = '55 Dolphin Place';
}
```

Is that the address variable or the address property of the turtle object?

The `with` statement can make programs confusing. It's been deprecated in JavaScript, and CoffeeScript doesn't have a `with` statement at all. Instead, fluent interfaces provide a solution without the ambiguity. When used with CoffeeScript's significant indentation, though, fluent interfaces do have potential for ambiguity. So before moving on, it's important to understand why that happens and how to avoid it.

7.6.2 The indentation problem

In CoffeeScript you need to be careful about indentation when using a fluent interface. If you're not careful, you might get some unexpected results. Imagine for a minute that *you* wrote the CoffeeScript compiler—what would you consider to be the meaning of chained syntax when parentheses are omitted?

```
turtle = new Turtle
turtle
.forward 2
.rotate 90
.forward 4
```

What do you expect this means?

Should there be any difference between that and a chained syntax with different indentation?

```
turtle
  .forward 2
  .rotate 90
  .forward 4
```

What do you expect this means?

They compile to the same thing. Should they? Much more importantly, versions *before* CoffeeScript 1.7 will compile both of these examples in a way you might not expect. Here's how CoffeeScript 1.6.3 compiles it:

```
turtle.forward(2..rotate(90..forward(4)));
```

Compiled JavaScript for the previous examples.

That will result in the error `Object 90 has no method forward` in JavaScript. With the potential for problems, what style *should* you use with fluent interfaces?

Fluent interfaces with parentheses

It's safest to use fluent interfaces *with parentheses and all flush left*:

```
turtle = new Turtle
turtle
.forward(2)
.rotate(90)
.forward(4)
```

Note that compilation will put it on one line and add a semicolon at the end.

That said, indenting method calls in a fluent method call chain is very common in other languages, so you'll frequently see this written so:

```
turtle = new Turtle
turtle
  .forward(2)
  .rotate(90)
  .forward(4)
```

Although the former (flush left) is cleaner (semantically), you should get used to seeing the latter (indented) style. If you're passing an object to a chained method, it's possible to skip the parentheses by indenting arguments:

```
NORTH = 0
turtle = new Turtle
turtle
.move
  direction: NORTH
  distance: 10
.stop()
```

Invoking the move method with an object that has direction and distance properties

This is helpful, and it helps to make the earlier competition examples easier to read:

```
makeCompetition
  max: 5
  sort: ->

makeCompetition
  sort: ->
  max: 5
```

This is useful for object parameters, but in most other cases you should avoid arguments on newlines.

FLUENT INTERFACES WITHOUT PARENTHESES

The minimalist inside you thinks that a fluent interface with parentheses seems wasteful and looks exactly like the equivalent JavaScript:

```
turtle
.forward(3)
.rotate(90)
.forward(1)
```

The good news is that if you're using CoffeeScript 1.7 or later, the compiler will recognize fluent interfaces with parentheses omitted so that the same expression without the parentheses has the same result:

```
turtle
.forward 3
.rotate 90
.forward 1
```

Since CoffeeScript 1.7 the preceding code will compile to the following JavaScript:

```
turtle.forward(2).rotate(90).forward(4);
```

Compiled JavaScript

As long as you keep the indentation consistent, you can pick whichever indentation level you prefer, and the compiler will close the parentheses on fluent call chains for you, so the following also works:

```
turtle
  .forward 3
  .rotate 90
  .forward 1
```

You can think of the dot on the newline as closing all and only implicit calls. It does not close other function calls:

```
wait = (duration, callback) ->
  setTimeout callback, duration

wait 5, ->
  turtle
    .forward 10
    .forward 3
```

A dot on a newline does not close the callback function passed to wait.

In CoffeeScript 1.7 the function call and fluent chain compile as follows:

```
wait(5, function() {
  return turtle.forward(1).rotate(90);
});
```

Compiled JavaScript.

Now, back to Scruffy's API work and how a fluent interface makes life easier for him.

7.6.3 Creating fluent interfaces

If you're creating an API from scratch, then a fluent interface requires a specific usage of `this` (a.k.a. `@`). Remember, inside a method call, `@` refers to the current object (the receiver of the method call). Consider what it means to use `@` as the final value in a method:

```
class Turtle
  rotate: (degrees) ->
    # rotate the turtle
    @
```

It doesn't matter how the turtle is rotated.

Return the turtle.

Because `@` on a line by itself at the end of a function looks a bit lonely, you'll often see the `this` keyword used instead:

```
class Turtle
  rotate: (degrees) ->
    # rotate the turtle
    this
```

Using `this` as the final value in a method returns the object that received the method call. So, if you return `this`, then method calls can be chained:

```
turtle = new Turtle
turtle.rotate(90).rotate(90)
```

If you're creating your own API, you can use this technique to make it fluent. You don't always write the API, though; oftentimes you only use it. For example, Scruffy didn't write the API for the turtle, but he has to use it. How can he use it with a fluent interface?

7.6.4 Chain

Not everything that can benefit from a fluent interface actually has one. Many of the APIs provided by web browsers are like this. Take the `canvas` API, for example. The `canvas` is covered in more detail in chapter 11, but for now you only need to know that it provides some drawing capabilities to web browsers, a little bit like the turtle:

```
canvas = document.getElementById 'example'
context = canvas.getContext '2d'
context.fillRect 25, 25, 100, 100
context.strokeRect 50, 50, 50, 50
```

Get an element to use as the canvas.

Create a context to draw on.

Fill in a rectangle.

Stroke the outside of the rectangle.

See how you have to repeat the `context` every time, just like Scruffy had to repeat `turtle`? Instead of living with that, you can make your own fluent interface out of this nonfluent interface. Remember that `with` statement from JavaScript? You can declare something roughly similar in CoffeeScript using the `apply` method on a function:

```
using = (object, fn) -> fn.apply object

using turtle, ->
  @forward 2
  @rotate 90
  @forward 4
```

Close, but not quite the same. You (and Scruffy) really want a fluent interface that chains method calls:

```
chain(turtle)
.forward(2)
.rotate(90)
.forward(4)
```

Before CoffeeScript 1.7

```
chain turtle
.forward 2
.rotate 90
.forward 4
```

CoffeeScript 1.7 and later

To do this you need to create a `chain` function that takes an object and returns a fluent interface of the object's methods. This is one of those cases where you need Agtron's help. The implementation of `chain` that he helps you create and an example of using it are shown in the following listing. If you don't understand exactly how it works, then it's okay to move on and come back to it later.

Listing 7.6 The `chain` function

```
chain = (receiver) ->
  wrapper = Object.create receiver
  for key, value of wrapper
    if value?.call
      do ->
        proxied = value
        wrapper[key] = (args...) ->
          proxied.call receiver, args...
          wrapper

  wrapper

turtle =
  forward: (distance) ->
    console.log "moving forward by #{distance}"
  rotate: (degrees) ->
    console.log "rotating #{degrees} degrees"

chain(turtle)
.forward(5)
.rotate(90)
```

Inherit from the receiving object that provides the API.

Comprehend the properties of the wrapper object, all of which come from the prototype.

Bind the local variable to the function for use later.

Create a closure to bind the local variable.

Duck typing: if the value of the property has a call method, then treat it as a function.

Define a property on the wrapper that's a function.

This function calls the proxied function with the argument and with the receiver bound.

Fluent interface: return the wrapper so that calls can be chained.

Fluent interface: return the wrapper from the chain function so that the chain can be started.

The chained method calls.

By providing some syntactic sugar, CoffeeScript can make more-advanced JavaScript techniques a bit more manageable. This sugar doesn't come for free, though; sometimes it can make things ambiguous.

7.7 Ambiguity

Removing parts can make things simpler, but it can also sometimes make them ambiguous. Removing words and symbols from a programming language is no different. To effectively create simple programs, you need to understand where there's potential for ambiguity so that you can avoid it. In CoffeeScript the most common areas where people accidentally create ambiguity are with significant indentation and implicit variable declarations.

7.7.1 Whitespace and indentation

All whitespace looks the same. Lack of visual variety means that you need to take care to make sure programs written in a language with significant indentation aren't ambiguous. You got a hint of this earlier with the turtle:

```
NORTH = 0

turtle
.move
  direction: NORTH
  distance: 10
.stop()
```

When reading this program, it's important to notice that the `move` method is being invoked with an object that has `direction` and `distance` properties. Now consider a `makeTurtle` function that makes and returns an object:

```
makeTurtle = ->
  move: ->
    # move the turtle
    this
  stop: ->
```

Again, when reading the code you need to pay attention. It's not just an object containing a `stop` property that's returned; it's an object with a `stop` property *and* a `move` property. How do you avoid this ambiguity?

Add characters to avoid ambiguity

Suppose you have a function that returns an array of objects containing information about your friends. It's tempting to leave out all the squiggly braces and end up with something like this:

```
friends = ->
    name: 'Bob'
    address: '12 Bob Street Bobville'
  ,
    name: 'Ralph'
    address: '11 Ralph Parade Ralphtown'
```

Sometimes, though, including the square and squiggly brackets makes it easier to read:

```
friends = ->
  [{
    name: 'Bob'
    address: '12 Bob Street Bobville'
  },
  {
    name: 'Ralph'
    address: '11 Ralph Parade Ralphtown'
  }]
```

This might not seem to be in the spirit of CoffeeScript, but the general rule should be that if you need to look at the compiled JavaScript to figure out if your Coffee-Script syntax is correct, then there's a good chance it's ambiguous. How about those parentheses?

Only function declaration parentheses have spaces

When invoking a function, there should be no space before the first parenthesis. No.Space.Ever! Although many syntax questions are a matter of taste, this one is not:

```
clarity = (important) ->
clarity()
```

When declaring, spaces on outside of the parentheses

When invoking, no space before the left parenthesis

When invoking a function, having no parentheses or putting them around the outside works fine:

```
(clarity 10)
clarity 10
```

Parentheses outside

No parentheses

But you should never put a space before the parentheses when invoking. It's confusing, broken, and looks too similar to a function being invoked with a callback argument:

```
clarity (10)
clarity (x) -> x
```

Don't do this

The clarity function being invoked with another (likely a callback) function

Finally, a note on subsequent function calls.

ADD PARENTHESES FOR SUBSEQUENT FUNCTION CALLS

Suppose you have three functions, `x`, `y` and `z`, and you invoke them as follows:

```
x y z 4
```

By glancing at it, in what order do you think they're being invoked? Adding parentheses shows you:

```
x(y(z(2)))
```

Even if you got it right this time, it's almost guaranteed that unparenthesized function calls will catch you out in CoffeeScript at least once sometime in the future. To avoid the problem, it's best to add some parentheses any time the expression might be ambiguous. Remember, you shouldn't have to refer to the generated JavaScript to understand what a CoffeeScript program is supposed to do. If it looks ambiguous, then you need to rewrite it. As figure 7.2 shows, you should stick to common language

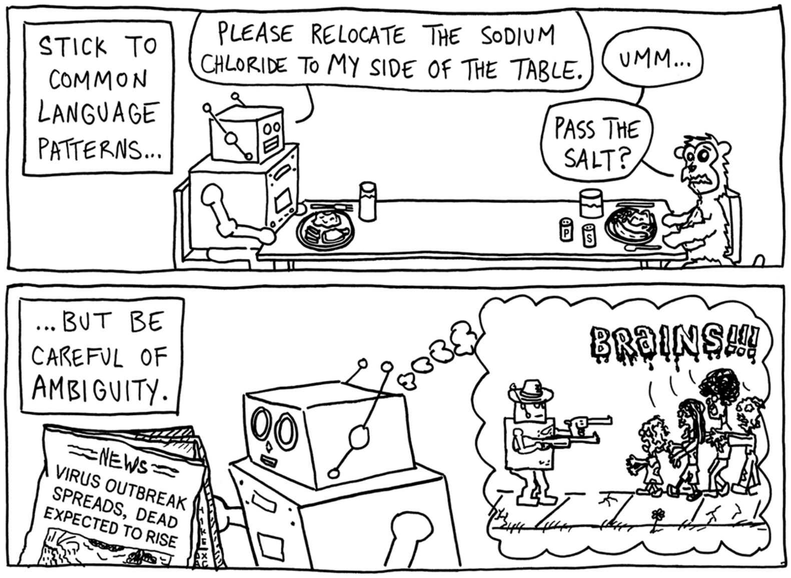

STICK TO COMMON LANGUAGE PATTERNS...
PLEASE RELOCATE THE SODIUM CHLORIDE TO MY SIDE OF THE TABLE.
UMM...
PASS THE SALT?
...BUT BE CAREFUL OF AMBIGUITY.
NEWS
VIRUS OUTBREAK SPREADS, DEAD EXPECTED TO RISE
BRAINS!!!

Figure 7.2 Stick to common language patterns, but be careful of ambiguity.

idioms, but be mindful of anything that might be ambiguous to you later, to the compiler, or to other people who have to work on your program.

One final area for potential confusion is in the way CoffeeScript implicitly declares variables for you.

7.7.2 Implicit variables

Remember that variables are declared for you, implicitly, the first time a variable name is used. If a variable name is already defined *anywhere* in the current lexical scope (including outer functions), then that variable is used. This is unlike the `var` keyword in JavaScript, which will create a variable in the current function scope regardless of whether a variable with the same name exists somewhere in the current lexical scope (such as an outer function). If your programs consist of small modules (as discussed in chapter 12) and you don't have deeply nested lexical scopes, then this implicit variable declaration is unlikely to cause you any pain.

If implicit variables become a problem for you, or if you simply don't like implicit variable declarations, then you can easily get around the absence of shadowing by taking advantage of the fact that function parameters *always* shadow and create local variables with a `do` expression:

```
x = 5

do (x) ->
  x = 3
  console.log x

console.log x

# 3
# 5
```

Using a do expression, you can have an explicit, local variable that can't clobber any outer variable when assigned.

This approach gives you a new way of writing a function that has variable names that shadow outer scopes:

```
shadowing = do (x) -> (y) ->
  x = 3
  x + y

shadowing 5
# 8
```

If you use a `do` expression in this way, then either there must be an outer variable that you want to shadow by assignment inside the `do` expression or you must assign a value in the parameters for the `do` expression:

```
do (notPreviouslyDefined) -> notPreviouslyDefined = 9
# ReferenceError: notPreviouslyDefined is not defined

do (notPreviouslyDefined='') -> notPreviouslyDefined = 'It is now'
# 'It is now'
```

Outside the `do` expression, the variable is still not defined:

```
notPreviouslyDefined
# ReferenceError: notPreviouslyDefined is not defined
```

Unfortunately, the syntax to achieve variables that shadow is a little clunky by CoffeeScript standards. Still, given that function parameters shadow the `do` expression, you can have local variables and can even approximate a future feature of JavaScript called `let` that you'll learn about in chapter 13. Until then, that's all there is on syntax.

7.8 Summary

CoffeeScript not only simplifies JavaScript's core syntax but also provides some syntactic sugar that can make programs easier to understand. Spread and rest parameters and destructuring provide the means to write concise expressions where long and confusing expressions (and statements) would otherwise be required. The syntax changes that CoffeeScript makes to JavaScript make programming more expressive and succinct.

Although CoffeeScript changes JavaScript's syntax dramatically, it makes only very small changes to JavaScript's semantics. You saw how to make the most natural use of CoffeeScript's dynamic types by learning to use duck typing.

Finally, succinctness of expression is a trade-off, and there is some potential for ambiguity in CoffeeScript. You learned how comprehensions and significant whitespace can be clarified and looked closely at how the common technique of fluent interfaces can be applied in CoffeeScript programs.

Moving on, as expressive and succinct as CoffeeScript is, the syntax and semantics of CoffeeScript have been decided for you. To have full control over the expressive power of the language you use, you need to be able to manipulate the language itself—you need metaprogramming, and that's exactly what the next chapter is about.

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.

"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

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

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]