CoffeeScript

Your guide book on
App Development with CoffeeScript

By Nicholas Brown

CoffeeScript

Your guide book on App Development with CoffeeScript

By Nicholas Brown

Table of Contents

Disclaimer

Introduction

The CoffeeScript programming language is easy for anyone to learn, including beginners. It largely borrows from the Ruby and Python programming languages. These languages are known for the flexible and extensive features they offer to programmers for applications development. The good thing about CoffeeScript is that one can create a large application with a small amount of code. This is because you can eliminate even brackets and curly braces if they are not necessary, unlike in most other programming languages. This also makes it easy to learn CoffeeScript. Creation of apps also becomes easy with CoffeeScript. The language also treats everything that you write as an expression. This book is a guide on how to program in CoffeeScript. Enjoy reading!

Chapter 1- A Brief Overview of CoffeeScript

CoffeeScript is a programming language which is finally compiled into JavaScript. The programming language largely borrows from Ruby and Python, and most of its syntax is related to the one used in these two programming languages. The Ruby on rails version 3.1 included the support for Coffeescript.

In Coffeescript, everything is almost treated as an expression. Examples of expressions in Coffeescript include the *"if," "switch,"* and *"for"* expressions. Most of these expressions usually have a postfix version. An example of this is the "*if*" expression, which can be written after the test condition. In this programming language, any unnecessary brackets and parenthesis can be omitted if need be. An example of this is when dealing with blocks of code, as we can omit brackets and use indentation. It has been found to add some sugar to the syntax used in JavaScript. For you to program in CoffeeScript, you don't have to learn the features in JavaScript which are less known. It involves less coding, meaning that one can perform tasks faster. Reading and maintenance of code is also made faster and easier.

This programming language has numerous features, and this is what we are going to discuss.

Chapter 2- Installing CoffeeScript

You should know that Coffeescript is a Node.JS package. This means that it relies on Node.js and npm as the package manager.

Installation on OS X

Installation of Node.js on OS X can be done using Homebrew, which is an open source package manager developed for OS X. Begin by installing the Homebrew and then updating it by using the command "*brew update*" on the terminal.

For you to install the Node.js, you have to type the command "*brew install node*" on the terminal. Any additional instructions should be followed, especially if you are in need of changing the PATH variable.

Finally, you can execute the command "*npm install -g coffee-script,*" and the installation will be done globally.

Installation on Windows

Installation of this on the Windows OS is a bit easier. Just visit the official Node.js website, and then click on the *"Install"* button. The version of the Windows OS you are using will be automatically detected, and the installer will start automatically. You can then go through the installation steps.

After that, you should perform the global installation. Just open the terminal, execute the command "*npm install -g coffee-script,*" and this will be done.

It is good for you to verify whether the installation was done successfully or not. Just open the terminal, and then execute the following command:

coffee –v

Consider the Coffesscript command given below:

coffee -o javascripts/ -c coffeescripts/

With the above command, all the files with a *“.coffee”* extension will be compiled into a folder named *“coffeescripts”* to *“.js”* contained in a parallel tree structure of a folder named *“javascripts.”*

Note that the *“-c”* option means to compile, while the *“-o”* option means the output. However, you should note that the order should be the output and then the compile as shown above. In case the order is switched or changed, then the command will not work, so avoid this.

Watcher

In our previous case, one has to execute the command each time that they are in need of compiling the files. Consider the command given below, which shows us the alternative to the other way of doing it:

```
coffee -w -o javascripts/ -c coffeescripts/
```

In the above command, we have used the "*-w*" option. What the watcher will do is that it will listen to the changes made to the files contained in the folder *coffeescripts,"* and then compile them immediately. Also, addition of another Coffeescript file to this folder will mean that it is compiled immediately, but in case you add another folder to the folder "*coffeescripts"* and then add the file inside it, then compilation will not be done. However, this is expected in the later versions of this.

Joining Files

All the files with a .coffee extension can also be compiled into a single JavaScript file. With this, the number of HTTP requests which the browser is expected to make will be greatly reduced, and the performance will be greatly improved. This can be done using the "*-j*" option. This is shown in the code given below:

```
coffee -j javascripts/app.js -c coffeescripts/*.coffee
```

"*" is referred to as the wildcard operator.

Chapter 3- Functions

For us to define a function in Coffeescript, we use a list of arguments which is optional contained in parenthesis, an arrow, and the body of the function. An empty function is represented using an arrow as “->.”

The *“function”* statement has been removed and replaced with an arrow “->.” The functions can be indented or liners on multiple lines. An implicit return is made to the last statement of a function. Also, the return statement need not be used unless one is in need of returning earlier inside your function.

Consider the example given below:

func = -> “myFunction”

After compilation of the above function, we should get the statement *“myFunction.”* The arrow will be returned into a function statement.

Also, we can use multiple statements if we need, provided we indent the lines of code properly as shown below:

```
var func;
func = function() {
 return “myFunction”;
};
```

Function arguments

Coffeescript allows the specification of arguments for a particular function, This can be done inside parenthesis and before the arrow as shown below:

```
times = (a, b) -> a * b
```

Default arguments are also supported in Coffeescript, as shown below:

```
times = (a = 1, b = 2) -> a * b
```

Splats can also be used for the purpose of accepting default arguments:

```
sum = (numbers…) ->
 result = 0
 numbers.forEach (n) -> result += n
 result
```

In the example given above, we have used the variable *"nums"* to denote an array of arguments which have been passed to the function. This is not an "*argument"* object, but just a real array, and if you are in need of manipulating it, you need not care about "*Array.prototype.splice*" or "*jQuery.makeArray()."*

```
trigger = (events…) ->
 events.splice(1, 0, this)
 this.constructor.trigger.apply(events)
```

Consider the next example showing how arguments can be made palatable in CoffeeScript:

```
planets = (first, second, third) ->
 alert second

planets 'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter'
```

# alert will be ‘Venus’

In the above JavaScript code, the last few parameters will just get dropped. However, CoffeeScript supports a feature known as splatting, which can help solve this problem. This is best demonstrated in the code given below:

```
planets = (first, second, rest…) ->
 alert rest

planets ‘Mercury’, ‘Venus’, ‘Earth’, ‘Mars’, ‘Jupiter’
# the alert will give us ‘Earth’,‘Mars’,‘Jupiter’
```

It is also possible for us to combine both the destructuring assignment feature with splatting. This is a good feature for us to deal with arrays having a variable length. Consider the code given below, which best describes how this can be done:

```
weekDays = “Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday”

[first, others…, last] = weekDays.split ‘,’
```

Function Invocation

Invocation of functions in Coffeescript is done in the same way as in JavaScript, and the functions “parens (),” “apply(),” or “call()” are normally used. In Coffeescript, just like in Ruby, the functions will be automatically called in case they are invoked with only one argument. This is shown below:

```
x = “Hello!”
alert x
# Equivalent to:
alert(x)
alert inspect x
# Equivalent to:
alert(inspect(x))
```

Although it is not a must for you to use parenthesis, we recommend that you always use them. The argument should always be wrapped in parenthesis as shown below:

```
alert inspect(x)
```

In case you pass no argument during the invocation, then there is no way that CoffeeScript will work it out if you are in need of invoking the function, but you can treat it like a variable.

Function context

Contexts usually change, most probably with event callbacks. CoffeeScript provides us with a few helpers which can assist us in doing this. An example of such is the variation in the arrow (->) and the fat arrow (=>).

If you use the fat arrow rather than the thin function, this means that the context of the function should be bound to the local one. This is shown below:

```
this.clickHandler = -> alert “It has been clicked”
element.addEventListener “click”, (e) => this.clickHandler(e)
```

For us to define the class and static functions, we can use the “@” annotation. The example given below demonstrates how this can be done:

```
class Num
```

```
 @random: ->

    return 5  # selected using fair dice roll

    # and guaranteed to be a random number

alert Num.random()
```

Object literals & array definition

In CoffeeScript, we can specify the object literals in the same way as in JavaScript, using braces and key/value statements. However, these braces are optional compared to function invocation. Other than the comma separation, one can make use of indentation and new lines. This is shown below:

```
obj1 = {one: 1, two: 2}

# Without the braces
obj2 = one: 1, two: 2

# Using the new lines instead of the commas
obj3 =
 one: 1
 two: 2

User.create(name: “John Joel”)
```

In arrays, whitespaces can be used in place of the comma separators, but the square brackets will be a necessity as shown below:

```
arr1 = [1, 2, 3]

arr2 = [
 1
 2
 3
]

arr3 = [1,2,3,]
```

Chapter 4- Flow control

The *“if”* and *“else”* statements in CoffeeScript make use of the parentheses. Consider the example given below:

```
if (true === true) {
 “Very ok”;
}
if (true !== true) {
 “Surprised”;
}
if (1 > 0) {
 “Ok”;
} else {
 “None of the above!”;
}
```

In case the “*if*” statement is contained in a single line, you will have to make use of the “*then”* keyword, and CoffeeScript will be in a position to know when the block has begun. There is no support for conditional operators in CoffeeScript, but you can just use a single line “if/else” statement.

CoffeeScript also features a Ruby Idiom as shown below:

```
alert “It’s cold!” if heat < 5
```

The keyword *“not”* can also be used for the purpose of indentation rather than the exclamation mark, and this will make your code more readable. This is shown below:

```
if (!true) {
 “Surprised”;
}
```

In the example given above, we could also have used the *“unless”* statement, which is the opposite of the “*if*” statement. This is shown in the code given below:

```
unless true
 “Surprised”
```

CoffeeScript, also introduced the statement “*is,*” and this can be used as shown below:

```
if true is 1
 “Type conversion has failed!”
```

“isn’t” can be used as an alternative to “is not.” This is shown in the example given below:

```
if true isnt true
 alert “Opposite day!”
```

Conditionals

Let us look at the variation of the "*if/else*" statement. Consider the code given below:

```
if lazy and bored
 sleep()
else
 walk()
// The Raw JS below

if (lazy && bored) {
    sleep();
} else {
    walk();
}
```

The ternary operator can be handled as shown below:

```
activity = if friday then isRelaxing else isWorking

    // The Raw JS below

 activity = friday ? isRelaxing : isWorking;
```

An additional semantic can be added by use of the *“unless”* keyword. This function will work in the opposite way as the “*if*” statement. This is shown below:

```
speedRunning() unless tired

 speedWorking unless focus is extremelyLow
```

And the compiled JavaScript will be as follows:

```
if (!lazy) {

    speedRunning();

}

if (focus !== extremelyLow) {

    speedWorking;
}
```

What about Libraries

When you use CoffeeScript, nothing happens to the Libraries, but they remain to be the same. This is because CoffeeScript is able to work with any third party library, whether it is big or small, because this is finally compiled to get JavaScript. You are just needed to reformat or refactor the code a bit slightly, but you should be determine with incompatibles.

Consider the code given below:

```
$(document).ready(function() {
 elementCollection = $('.collection');
    for (j=0; j<=elementCollection.length;j++)
    {
    item = elementCollection[j];
    // checking for a random property here. The check can also be skipped //here as we
have done
    color = item.hasProperty()? yesColor : noColor;
    // I am very aware that there are better ways on to do this
    $(item).css ('background-color', color);

    }

});
```

Instead of having to write the code as shown above, you can do it as shown below:

```
$(document).ready ->

    elementCollection = $('.collection')

    for item in elementCollection

    color = if item.hasProperty() then yesColor else noColor

    $(item).css 'background-color', color
```

String interpolation

This feature is used in CoffeeScript. Consider the example given below:

```
fav_color = “Green. No, blue…”
question = “Bridgekeeper: Which color do you like?.. choose from the availables
   Galahad: #{fav_color}
   Bridgekeeper: Wrong!
   “
```

In the above example, multiline strings have been used, meaning that they are allowed in CoffeeScript, and there is no need for us to use “+.”

Chapter 5- Loops and Comprehensions

The syntax used for array iterations in JavaScript is somehow complex, and it resembles the one used in old programming languages such as C. CoffeeScript comes with a new syntax so as to save on this problem as shown in the code given below:

```
for name in [“John”, “Meshack”, “Brian”]
 alert “Release #{name}”
```

In case you are in need of the current iteration syntax, just pass in an extra argument as shown below:

```
for name, j in [“John the Shopkeeper”, “Brian the footballer”]
 alert “#{j} - Release #{name}”
```

One can also use the postfix form so as to iterate on a single line. This is shown below:

release person for person in ["", "John", "Brian"]

Looping Through Collections

It is easy for one to loop through arrays. One has to use the "for..in" so as to loop through the loop. Consider the code given below, which shows how this can be used:

```
sites = ['MyWebsite','ThemeForest','ActiveWeb']
for site in sites
 alert site
```

If you need to put all your statements in one line, do it as shown below:

```
sites = ['MyWebsite','ThemeForest','ActiveWeb']
alert site for site in sites
```

In CoffeeScripts, all the above are compiled into "*for*" loops. Know that with the best practices, the length of the array has to be cached beforehand. The code given below best describes this:

```
var site, sites, _j, _len;

sites = [‘MyWebsite’, ‘ThemeForest’, ‘ActiveWeb’];

for (_j = 0, _len = sites.length; _j < _len; _j++) {

    site = sites[_j];

    alert(site);

}
```

Iterating over the associate arrays (or dictionaries, hashes, or key-value pairs) is very easy when using the “*of*” keyword. Consider the code given below which shows how this can be done:

```
managers = ‘MyWebsite’: ‘Jeffrey Boss’, ‘ThemeForest’: ‘Mark Donald’, ‘ActiveWeb’: ‘Lance Soldier’

 for site, manager of managers

    alert manager + ” manages ” + site
```

The above code will just be compiled into a basic for loop. This is shown in the code given below:

```
var manager, managers, site;

managers = {

    ‘MyWebsite’: ‘Jeffrey Boss’,

‘ThemeForest’: ‘Mark Donald’,

‘ActiveWeb’: ‘Lance Soldier’};

for (site in managers) {

    manager = managers[site];

    alert(manager + ” manages ” + site);

}
```

That is what we can say about it.

With comprehensions in CoffeeScript, the code can be made to be more readable. Consider the example given below:

```
congratulate = (first, second, third, others…) ->

 addCongratsToUser first, 10

 addCongratsToUser second, 6
addCongratsToUser third, 2
    addCongratsToUser user, 1 for user in others
```

The good thing with looping in CoffeeScript is that we finally get the array of the results once the loop has completed looping. Suppose that I am in need of counting from 0 to 100, and I need to do it in multiples of 10. I just have to prepend the loop with a variable assignment as shown below:

```
mults = for number in [0..10]
    number * 10
```

If you need to bring all of the above into one line, you can just make use of parenthesis. This is shown in the code given below:

```
mults = (number * 10 for number in [0..10])
```

Most loops written in CoffeeScript are just comprehensions over objects, arrays, and ranges. *“for”* loops are always replaced with comprehensions. Optional guard clauses are used together with the value of the current array index. Array comprehensions are just expressions, unlike what happens in for loops, and we can return and assign them. Consider the example given below:

```
# Take dinner.

eat food for food in [‘chapati’, ‘githeri’, ‘rice’]

# Fine five course of dining.

groups = [‘greens’, ‘caviar’, ‘truffles’, ‘roast’, ‘cake’]

menu j + 1, dish for dish, j in groups


# Health conscious meal.

meals = [‘broccoli’, ‘spinach’, ‘sukuma wiki’]

eat food for food in meals when food isnt ‘spinach’
```

For you to specify the end of your comprehension, you can choose to use a range. However, this should be done if you are aware of where your loop starts and ends as shown below:

```
count = (num for num in [20..10])
```

In the above case, the result of each iteration will be returned into an array, and the value of the comprehension has been assigned to a variable.

We can use comprehension so as to iterate through the keys and values used in an object. The "*of*" should be used for signaling comprehension over properties of an object rather than the values contained in an array. This is shown below:

```
yearsOfAge = joel: 20, boss: 15, anthony: 11
ages = for child, age of yearsOfAge
 ”#{child} is #{age}”
```

The "*while*" loop is the only low-level loop which CoffeeScript provides to its users. This makes a difference when compared to what happens in JavaScript, as it can be used as an expression, and an array having the result of iteration through our loop. This is shown below:

```
# BCOM 101
if this.studyingBCOM
 buy()  while supply > demand
 sell() until supply > demand


number = 6
lyrics = while number -= 1
 ”#{number} little cats, jumping on bed.
    One of them fell out and then bumped his head.”
```

To enhance readability, be aware that the *“until”* keyword is the same as *“while not,”* and the keyword *“loop”* is similar to *“while true.”*

In JavaScript, when working with loops for generation of functions, we are expected to insert a closure wrapper so as to make sure that the loop variables have been closed over, and the generated functions will not have to share only the final values. The *“do”* keyword is provided in CoffeeScript, and this serves to invoke the function which is passed immediately, and the arguments will be forwarded. This is shown in the code given below:

```
for file in list
```

```
do (file) ->

  fs.readFile file, (err, contents) ->

  compile file, contents.toString()
```

Chapter 6- Objects and Arrays

The objects and arrays used in CoffeeScript are closely related to those used in JavaScript. You have the choice of using the commas or not if each property has been listed on its own line. For you to create objects, you can use indentation rather than the braces. Consider the example given below:

```
song = [“do”, “re”, “mi”, “fa”, “so”]

dancers = {Joel: “Rock”, Nicholas: “Roll”}

blist = [

 1, 0, 1

 0, 0, 1

 1, 1, 0

]


kids =

 brother:

    name: “Max”

    age:  14
 sister:
    name: “Ida”
    age:  10
```

Note that in JavaScript, it is impossible for one to use reserved keywords such as “*class*” as the properties of the object, without having to quote them as strings. CoffeeScript will take note of the reserved words which have been used as keys in your object and then quote them for you, meaning that you don’t have to be worried about it. Consider the code given below:

```
$(‘.account’).attr class: ‘active’
log object.class
```

Variable Safety and Lexical Scoping

It is the work of the CoffeeScript compiler to ensure that all of your variables are properly declared within the lexical scope. This is why you are not expected to write the *“var”* on your own. Consider the example given below, showing how this is done:

```
outer = 1
chNumbers = ->
 inner = -1
 outer = 10
inner = chNumbers()
```

As you might have noticed, all of the variable declarations have been pushed to the top of your closest scope, and this should be the first time when they appear.

Variables

CoffeeScript supports both class and instance variables. Let us show how these can be created.

Class Variables

Consider the code given below, which shows how these can be created in CoffeeScript:

```
class MyClass
 @MAX_ANIMALS: 40
 MAX_ZOOKEEPERS: 4

 helpfulInfo: =>
    “Zoos may have animals up to a maximum of #{@constructor.MAX_ANIMALS}
and #{@MAX_ZOOKEEPERS} zoo keepers.”
```

```
MyClass.MAX_ANIMALS

# => 40



MyClass.MAX_ZOOKEEPERS

# => undefined (This is a member of prototype)



MyClass::MAX_ZOOKEEPERS

# => 4



myclass = new MyClass

myclass.MAX_ZOOKEEPERS

# => 4

myclass.helpfulInfo()

# => “They may have animals up to a maximum of 50 and 3 zoo keepers.”



myclass.MAX_ZOOKEEPERS = “smelly”

myclass.MAX_ANIMALS = “seventeen”

myclass.helpfulInfo()

# => “They may have animals up to a maximum of 50 and smelly zoo keepers.”
```

Instance variables

Instance variables have to be defined inside the method of a class, and then initialize the defaults in the constructor. Consider the code given below, which shows how this can be done:

```
class MyClass
 constructor: ->
    @anims = [] # the instance variable should be defined here
 addAnimal: (name) ->
    @anims.push name

myclass = new MyClass()
myclass.addAnimal ‘heyna’

otherClass = new MyClass()
otherClass.addAnimal ‘lion’

myclass.anims
```

```
# => [‘heyna’]

otherClass.anims
# => [‘lion’]
```

However, avoid adding the variable accidentally to the prototype, and this is why it should not be defined outside a constructor. The code given below best demonstrates this:

```
class BadClass
 anims: []              # will translate to BadClass.prototype.anims = []; and is shared between the instances
 addAnimal: (name) ->
   @anims.push name  # this will work due to the concept of prototypes in #JavaScript

myclass = new BadClass()
myclass.addAnimal ‘heyna’

otherClass = new BadClass()
otherClass.addAnimal ‘lion’
```

myclass.animals

# => [‘heyna’,‘lion’] # Oops…

otherClass.animals

# => [‘heyna’,‘lion’] # Oops…

BadClass::animals

# => [‘heyna’,‘lion’] # The value will be stored in the prototype

That is it. As you have noticed, CoffeeScript will store the values of the class variables in the class itself rather than in the prototype which it defines.

Chapter 7- Decision Making Statements

The “*if/else*” statement in CoffeeScript can be used without the need for parenthesis and curly braces. In the case of multi-conditional statements, we use indentation so as to align and group them.

The ternary operator in CoffeeScript can be used for the purpose of compiling an “*if*” statement into a JavaScript expression. Closure wrapping can also be used if it is possible for us to use it. In CoffeeScript, there exist no explicit ternary statement as you just have to use a single “*if*” statement in a single line. This is shown below:

```
mood = improved if dancing

if happy and knowsIt

 clapsHands()

 paPaPa()

else

 showIt()

date = if thursday then sue else jill
```

Splats

The splats (…) are used in CoffeeScript for the purpose of function definition and its invocation, and the variable numbers of the arguments to be a bit palatable. In JavaScript, the arguments object is used for the purpose of working with functions which accept the variable numbers of the arguments. This is shown below:

```
gold = silver = remaining = “unknown”
medals = (first, second, rest…) ->
 gold   = first
 silver = second
 remaining   = rest
contenders = [
 “Michael Joseph”
 “David Rudisha”
 “Ken Boss”
 “Joel Felix”
 “Anthony Johnson”
 “Roman Seba”
 “Thomas Jiung”
 “Tyson Gay”
 “Asman Powell”

 “Usain Bolt”

]


medals contenders…

alert “Gold: ” + gold

alert “Silver: ” + silver

alert “The Field: ” + remaining
```

Chapter 8- Array Slicing

For us to extract slices of arrays, we can use ranges. When two dots are used in the range, example, 5..8, then it is inclusive, but when three dots are used, example, 5…8, then the range is exclusive at the end. In the former case, we have the values 5,6,7,8, while in the latter case, we have the values 5,6,7. The last value has not been included in our latter case.

In slice indices, there are useful defaults. If the first index is omitted, then it will default to zero, while if the second index is omitted, it just defaults to the size of your array. Consider the example code given below:

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]

start   = nums[0..2]

middle  = nums[3…-2]

end     = nums[-2..]

copy    = nums[..]
```

The syntax used above can also be applied to assignments so as to replace a particular segment of an array with some new values while splicing it. This is shown below:

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

nums[3..6] = [-3, -4, -5, -6]
```

You have to note that in JavaScript, strings are immutable, meaning that we cannot splice them.

Although we have not been using the *"return"* statement in our CoffeeScript programs, they have really returned a value. What happens is that the CoffeeScript compiler always tries to ensure that every statement written in CoffeeScript is used as an expression. The return always gets pushed down into any possible branch of execution as shown in the function given below:

```
grade = (student) ->
 if student.excellent
    "A"
 else if student.good
    if student.workedHard then "B" else "B-"
 else
    "C"
```

```
eldest = if 24 > 21 then "Mercy" else "Eliza"
```

Although our functions will always return the final value, it is good for us to return quickly from the function body which writes the explicit return once you are sure that you are done.

Since declarations of variables is done at the top of the scope, we can use assignments within expressions, even for the variables we have not seen before. This is shown in the code given below:

```
seven = (one =1) + (two = 2) + (four = 4 )
```

Expressions which could have been used in JavaScript when used in CoffeeScript are converted into an expression just by wrapping them in a closure. With this, one can do most things, such as assignment of a comprehension to a defined variable. Consider the example given below:

```
# Our ten global properties which comes first.

globs = (name for name of window)[0…10]
```

This also involves a simple task such as passing a “*try/catch*” statement into a function call as shown in the code given below:

```
alert(
 try
    nonexistent / undefined
 catch error
    “The occurrence error is … #{error}”
)
```

In JavaScript, there exists a handful of expressions which cannot be converted into expressions meaningfully, and these include the break, continue, and return statements. If these are used within a block of code, the CoffeeScript will not try to perform the conversion.

Chapter 9- Operators and Aliases

The use of the operator “==” usually leads to an undesired coercion, CoffeeScript translates it into “===” and the “!=” to “!==.” Also, the operator “is” is compiled into “===” and “isn’t” to “!==.”

If you need to make your work simple, use the operator “//” for division and “**” to represent exponential. The modulus operator “%” works the same as in JavaScript. Consider the example code given below:

```
-7 % 5 == -2 # gives remainder of 7 / 5

-7 %% 5 == 3 # n %% 5 always lies between 0 and 4


tabs.selectTabAtIndex((tabs.currentIndex - count) %% tabs.length)
```

Consider the next example given below:

```
launch() if ignition is on
```

```
vol = 10 if band isnt SpinalTap

letTheWildRumpusBegin() unless answer is no

if vehicle.speed < limit then accelerate()

winner = yes if pick in [40, 90, 15]

print inspect “The name is #{@name}”
```

The Existential Operator

Checking for a variable existence in JavaScript is of great importance. The existential operator in CoffeeScript "?" is used for this purpose, and always returns a "*true*" value unless the variable is null or has not been defined.

This operator can also be used for the purpose of a safer conditional assignment that is provided by "||=," and this applies when dealing with strings and numbers. Consider the example given below:

myVar = true if mind? and not world?

speed = 0

speed ?= 20

prints = mine ? "bear"

The accessor variant of our for our existential operator "?." may also be used for soaking up null references in a chain of the properties. This should be used in place of the dot operator (.) in places where you may find the base value being null or undefined. If you find all of your properties existing, then you will obtain your expected result, in any case the chain is broken, and instead of getting the TypeError, you will get undefined. This is shown in the code given below:

```
zip = lottery.drawWinner?().address?.zipcode
```

Nifty, Semantic Aliases

Aliases are good for making code to be much more readable. CoffeeScript provides us with aliases for keywords and operators, and this really makes it easy for us to make the code intuitive and more readable.

Consider the sample code given below:

```
if shirt is burningOnFire
 lookForWater()

if game isnt good
 boringGame();
```

The above code should be compiled to the following:

```
if (shirt === burningOnFire) {

    lookForWater();

}

if (game !== good) {

    boringGame();

}
```

We want to use this example so as to demonstrate how logical operators are mapped in CoffeeScript. The code can be built to the following:

```
if shirt is burningOnFire and not aDream

 lookForWater()


if game isnt good or haughtyDevs

 boringGame();
```

The above code should be compiled to the following:

```
if (shirt === burningOnFire && !aDream) {

    lookForWater();

}

if (game !== good || haughtyDevs) {

    gameBoring();

}
```

Chapter 10- Classes and Inheritance

In JavaScript, we are provided with a well-defined syntax on how we can do inheritance. There are several libraries in JavaScript which provide us with syntactic sugar, and we can enjoy the feature of inheritance except for a number of exceptions.

In CoffeeScriipt, we do not need to repeatedly attach functions to our prototype, since we are provided with a basic structure for our classes which helps us to name classes, set up super classes, assign the prototypal properties, and perform a definition of our constructor, and this is done in a single assignable expression.

The constructor functions are always named so as to support the helpful stack traces. Consider the example given below, which shows how this is done:

```
class Animal
 constructor: (@animalName) ->


 move: (meters) ->
```

```
    alert @animalName + ” moved #{meters}m.”


class dog extends Animal

 move: ->

    alert “run…”

    super 10


class Horse extends Animal

 move: ->

    alert “Galloping…”

    super 40


poodle = new Dog “Poodle the dog breed”

tom = new Horse “Tommy is my horse”


poodle.move()

tom.move()
```

If you are not interested in structuring your prototypes classically, CoffeeScript provides you with the conveniences of a low level. With the operator *“extends,”* one is provided with a good setup for the prototype, and this can be used for the purpose of creating an inheritance chain between the constrictor functions. Consider the code given below:

```
String::dasherize = ->
 this.replace /_/g, “-”
```

You have to note that the class definitions are just blocks of executable code, and they are good for enhancing metaprogramming.

Destructuring Assignment

There is a destructuring syntax in CoffeeScript, just like in JavaScript. Once an array or an object literal has been assigned to a value, the CoffeeScript will break up and then match both sides against each other, and the values in the right will be assigned to the variables located on the left. The simplest use of this is in parallel assignment. The code given below shows how this can be done:

```
bait   = 1000
theSwitch = 0

[bait, theSwitch] = [theSwitch, bait]
```

This is also helpful when it comes to the use of functions in which we expect to get multiple values. Consider the example given below, which shows how this can be done:

```
wReport = (location) ->
 # Making an Ajax request for fetching the weather…
 [location, 72, “It is Sunny”]
```

```
[city, temp, forecast] = wReport “Berkeley, CA”
```

The feature destructing assignment can be used with any depth of an array or object nesting, and this will help us to pull out nested properties which are deep. Consider the code given below:

```
futurists =
 sculptor: “John Joel”
 painter:  “Mercy Anthony”
 poet:
    name:   “Nicholas Bott”
    address: [
    “Via Roma 42R”
    “Bellagio, Italy 22021”
    ]

{poet: {name, address: [street, city]}} = futurists
```

It is also possible for us to combine destructuring assignments with splats. This can be done as shown below:

```
tag = “<impossible>”

[open, contents…, close] = tag.split(””)
```

We can make use of the extraction feature so as to retrieve elements from the array’s end, and we will not have to assign the rest of the values. This can also work effectively in the parameter listing. Consider the example given below, which best describes this:

```
text = ” We can make use of the extraction feature so as to retrieve elements from the
array’s end and we will not have to assign the rest of the values”

[first, …, last] = text.split ” “
```

The destructuring assignment feature is very useful once it has been combined with a class constructor for the purpose of assigning properties to the instance from options object which has been passed to the constructor. This is shown in the code given below:

```
class Man
 constructor: (options) ->
    {@name, @age, @height = ‘average’} = options
tim = new Man name: ‘John’, age: 5
```

With the above example, it is also very clear that if there are no properties in the destructured object or array, one can provide defaults, as it is done in JavaScript. The difference brought in CoffeeScript is that both null and undefined values are treated the same.

Chapter 11- Generator and Bound Functions

In JavaScript, we use the keyword “*this*” to refer to the object to which the function has been attached. If a function is passed as a callback, or maybe it is attached to a different object, then we will lose the value of the object “*this.*”

The fat arrow “=>” can be used for defining a function, and in binding it to the current value of the “*this*” object. Consider the example given below, which best describes this:

```
Account = (buyer, cart) ->
 @buyer = buyer
 @cart = cart

 $(‘.shopping_cart’).on ‘click’, (event) =>
    @buyer.purchase @cart
```

In case the thin arrow had been used in the above function callback, the property @buyer would have been referring to an undefined buyer property of our DOM element, and when we try to call the “*purchase()*” function, an exception would have been raised.

The keyword *“yield”* is also good for supporting the ES6 generator functions in CoffeeScript functions. To simplify everything, a generator function is just any function which yields something. Consider the example given below which shows how this can be used:

```
pSquares = ->
 number = 0
 loop
    number += 1
    yield number * number
 return

window.ps or= pSquares()
```

Embedded JavaScript

Sometimes, you may need to embed some JavaScript codes inside your CoffeeScript code. Consider the example given below, which shows how this can be done:

```
Hello = `function() {
 return [document.title, “Hi JavaScript”].join(“: “);
```

}`

Chapter 12- Switch/When/Else

The use of the *“switch”* statement in CoffeeScript seems to be a bit awkward. One is used to implement a *“break”* at the end of each *“case”* statement so as to avoid executing the default statement. The *“switch”* can also be converted so as to get an assignable, returnable expression.

Consider the example given below:

```
switch day
 when “Mon” then avoid being lazy
 when “Tue” then remember your friends
 when “Thu” then go swimming
 when “Fri”, “Sat”
    if day is bingoDay
    go bingo
    go fotball
 when “Sun” then go church
 else go work
```

What happens in CoffeeScript is that a switch statement can take multiple values for each "*when*" clause which is being used.

It is also possible for us to use the switch statement without control expression, and it turns them into a cleaner alternative to chains of "*if/else.*" This is shown in the code given below:

```
score = 85
grade = switch
 when score < 60 then 'F'
 when score < 70 then 'D'
 when score < 80 then 'C'
 when score < 90 then 'B'
 else 'A'
# grade == 'B'
```

Try/Catch/Finally

The “*try*” expression in CoffeeScript takes the same syntax as in JavaScript. However, CoffeeScript allows you to omit the parts for catch and finally. In the catch part, the error parameter can also be omitted if you don’t need. This is shown in the code given below:

```
try
 breaksLoose()
 catsAndDogsLiveTogether()
catch error
 print error
finally
 cleanUp()
```

Chained Comparisons

In CoffeeScript, chained comparisons are borrowed from Python, and this makes it easy for us to test in case the value falls within a certain range. This is shown in the code given below:

```
cholesterol = 120

healthy = 190 > cholesterol > 50
```

String Interpolation

In CoffeeScript, the feature string interpolation is used in the same way as in the Ruby programming language. Interpolated values are allowed in double-quoted strings by use of "#{ … }," while the single-quoted strings are literals. Interpolation may also be used in object keys. Consider the code given in the example below:

author = "JohnPitson"

quote = "An author should be creative. — #{ author }"

sentence = "#{ 22 / 7 } is a very decent approximation of the π"

Multiline strings are also supported in CoffeeScript. A single space is used for joining lines, unless the line ends with a backslash. Indentation is also ignored. Consider the example given below:

mobyDick = "My name is Nicholas. I am a programmer —

I have worked as a software developer for long – I am a Computer Science

```
    graduate, and I have numerous certificates
related to IT –my interest is to work as a programmer forever
```

Block strings can be used for the purpose of holding text which has been formatted and is indentation sensitive. The indentation level begun by the block has been maintained throughout the code, meaning that it has to be kept aligned with the code.

```
html = ”””
   <strong>
   This is coffeescript
   </strong>
   ”””
```

Block strings which have been double quoted allow for interpolation, just like the other double-quoted strings.

Chapter 13- Block Regular Expressions

Block regexes are supported in CoffeeScript just like the block comments and strings. Block regexes are the extended regular expressions which do ignore whitespaces, and they contain interpolation and comments. These are very useful as they always work towards making complex regular expressions to be simple. If you are in need of quoting from a CoffeeScript source, do it as shown below:

```
OPERATOR = /// ^ (
 ?: [-=]>             # function
    | [-+*/%<>&|^!?=]=  # compound assign / compare
    | >>>=?             # zero-fill right shift
    | ([-+:])\1         # doubles
    | ([&|<>])\2=?      # logic / shift
    | \?.               # soak access
    | .{2,3}            # range or splat
) ///
```

Cake, and Cakefiles

CoffeeScript support cake which is used in tasks which are in need of building and testing the CoffeeScript language. For you to define the tasks, you have to do it in a file with the name "*cakefile,*" and for you to invoke it, you have to execute the command "*cake [task]*" from the directory. If you are in need of displaying the available tasks and options, just type the command "*cake.*"

The definition of the tasks is done in the CoffeeScript, meaning that you are free to add some arbitrary code to the file. A task should be defined with a name, a long description, and the function which is to be invoked once the task has been run. In case your task is taking a command-line option, the option can be defined with long and short flags, and this is always made available in the object "*options.*" Consider the task given below which makes use of the Node.js API for rebuilding the CoffeeScript server:

```
fs = require ‘fs’

option ‘-o’, ‘—output [DIR]’, ‘directory for the compiled code’

task ‘build:parser’, ‘rebuild the Jison parser’, (options) ->

 require ‘jison’

 code = require(‘./lib/grammar’).parser.generate()

 dir  = options.output or ‘lib’
```

```
fs.writeFile “#{dir}/parser.js”, code
```

If you are in need of invoking one task before the other, such as performing the build before the test, you can make use of the “*invoke*” function “invoke ‘build.’” The cake tasks are a minimal way that one can expose the CoffeeScript functions to our command line. If you are in need of dependencies, or the async callbacks, it will be good if you put them in the code itself but not in the cake task.

Chapter 14- Converting jQuery to CoffeeScript

jQuery and CoffeeScript can be used hand in hand.

Safe jQuery Closure

One can make use of the $ sign as shown below:

```
(($) ->

) jQuery
```

When compiled, we get the following:

```
(function($) {

})(jQuery);
```

DOM Ready

The code for this can be written as shown below:

```
$ ->
 console.log(“The DOM is ready”)
```

When compiled, the jQuery should be as follows:

```
$(function() {
 return console.log(“The DOM is ready”);
});
```

Calling a Method with no Params

The CoffeeScript code can be written as shown below:

```
$(“.submit”).click ->
 console.log(“Has been submitted!”)
```

The compiled code for the above will be as shown below:

```
$(“.submit”).click(function() {
 return console.log(“Has been submitted!”);
});
```

Calling a Method with one Param

This can be done as follows in CoffeeScript:

```
$(“.button”).on “click”, ->
 console.log(“The button has been clicked!”)
```

The compiled version of the above code will be as shown below:

```
$(“.button”).on(“click”, function() {
 return console.log(“The button has been clicked!”);
});
```

Calling a Method with Multiple Params

This can be done as shown below:

```
$(document).on “click”, “.button2”, ->
 console.log(“The delegated button has been clicked!”)
```

When compiled, the code will appear as shown below:

```
$(document).on(“click”, “.button2”, function() {
    return console.log(“The delegated button has been click!”);
});
```

When the params are in an anonymous function, we can implement them as follows in CoffeeScript:

```
$(“.button”).on “click”, (event) ->

 console.log(“The button has been clicked!”)

 event.preventDefault()
```

The compiled version should be as follows:

```
$(“.button”).on(“click”, function(event) {

 console.log(“The button has been clicked!”);

 return event.preventDefault();

});
```

To return a “*false*” in CoffeeScript, do it as follows:

```
$(“.button”).on “click”, ->

 False
```

The compiled version for the same will be as follows:

```
$(“.button”).on(“click”, function() {

 return false;

});
```

A Simple Plugin

This can be created as shown below:

```
$.fn.extend
 favColor: (options) ->
    settings =
    option1: “blue”
    settings = $.extend settings, options
    return @each () ->
    $(this).css
    color: settings.color
```

When compiled, the above will be as follows:

```
$.fn.extend({
 favColor: function(options) {
    var settings;

    settings = {

    option1: “blue”

    };

    settings = $.extend(settings, options);

    return this.each(function() {

    return $(this).css({

    color: settings.color

    });

    });

 }

});
```

The plugin can then be used as shown below:

```
$(“a”).favColor

    color: “green”
```

Ajax

The CoffeeScript should be as follows:

```
$.ajax

    url: “file.html”

    dataType: “html”

    error: (jqXHR, textStatus, errorThrown) ->

    $(‘body’).append “AJAX Error: #{textStatus}”

    success: (data, textStatus, jqXHR) ->

    $(‘body’).append “A Successful AJAX call: #{data}”
```

We will then have made an Ajax call. Its compiled version will be as follows:

```
$.ajax({

 url: “file.html”,

 dataType: “html”,

 error: function(jqXHR, textStatus, errorThrown) {

    return $(‘body’).append(“AJAX Error: ” + textStatus);

 },

 success: function(data, textStatus, jqXHR) {
    return $(‘body’).append(“A Successful AJAX call: ” + data);

 }

});
```

Animation

There are two ways that this can be implemented in JavaScript. Consider the first method shown below:

```
div.animate {width: 190}, 2000
```

```
div.animate
 width: 190
 height: 190
 2000
```

The compiled version of this code will be as follows:

```
div.animate({
 width: 190
}, 2000);
```

```
div.animate({
 width: 190,
 height: 190
}, 2000);
```

The following code shows how animation with a callback can be done in JavaScript:

```
div.animate
 color: green
 2000
 ->
    doAction()
```

The compiled version of this should be as shown below:

```
div.animate({
 color: green
}, 2000, function() {
 return doAction();
});
```

Promises

The code given below shows how we can implement promises in CoffeeScript:

```
$.when(
 $.get("/feature/", (html) ->
    myStore.html = html;
 ),
 $.get("/style.css", (css) ->
    myStore.css = css;
 )
).then ->
 $("<style />").html(myStore.css).appendTo("head")
 $("body").append(myStore.html)
```

The compiled version for the same should be as shown below:

```
$.when($.get("/feature/", function(html) {

 return myStore.html = html;

}), $.get("/style.css", function(css) {

 return myStore.css = css;

})).then(function() {

 $("<style />").html(myStore.css).appendTo("head");

 return $("body").append(myStore.html);

});
```

When the fat arrow is used, we can come up with the code given below:

```
$(".button").click ->

 setTimeout ( =>

    $(@).slideUp()

 ), 400
```

Its compiled version will be as follows:

```
$(“.button”).click(function() {

 return setTimeout(((function(_this) {

    return function() {

    return $(_this).slideUp();

    };

 })(this)), 400);

});
```

Conclusion

We have come to the conclusion of this guide. As a programming language, CoffeeScript offers numerous features to the users which they can really enjoy. The language borrows largely from Python and Ruby programming languages and as you might have noticed, most of its syntax is largely borrowed from these two programming languages. If you are an expert in either Python or Ruby, then programming in CoffeeScript is easy for you. It greatly relies on Node.js, so for you to program in CoffeeScript, you must have installed the Node.js and its dependency module, which is "*npm.*" Installation of Node.js is easy, as you just have to go to its official website and then click on the "*Install*" button and a button will appear prompting you to start the installation process. You have to remember that CoffeeScript code is always compiled into JavaScript code.

The language supports the use of flow control and decision making statements, so learn how to implement these in your program to enhance its functionalities. Functions are well supported, and there are variations on how arguments can be used. Note that for each CoffeeScript code that you write, you can get an equivalent JavaScript code. This is because the CoffeeScript is always compiled to get a JavaScript. It is easy for one to learn CoffeeScript.