FREE eBook

LEARNING coffeescript

Free unaffiliated eBook created from
Stack Overflow contributors.

#coffeescrip

t

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: coffeescript

It is an unofficial and free coffeescript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official coffeescript.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with coffeescript

Remarks

This section provides an overview of what coffeescript is, and why a developer might want to use it.

It should also mention any large subjects within coffeescript, and link out to the related topics. Since the Documentation for coffeescript is new, you may need to create initial versions of those related topics.

Examples

Hello Word (Linux and OS X)

CoffeeScript is a scripting language that compiles into JavaScript. Any code written in CoffeeScript can be translated into JavaScript with a one-to-one matching.

CoffeeScript can be easily installed with `npm`:

```
$ mkdir coffee && cd coffee
$ npm install -g coffee-script
```

The `-g` flag will install CoffeeScript globally, so it will always be available on your CLI. Don't use the `-g` flag if you want a local installation:

```
$ mkdir coffee && cd coffee
$ npm install coffee-script
```

When the package is installed, create a `helloword.coffee` file in the working directory and write some CoffeeScript code in it.

```
console.log 'Hello word!'
```

This code can be executed by calling the CoffeeScript binary. If you installed CoffeeScript globally, simply run:

```
$ coffee helloword.coffee
```

If you installed CoffeeScript locally, you will find the binary in the installation folder:

```
$ ./node_modules/coffee-script/bin/coffee helloword.coffee
```

In both cases, the result will be printed in the console: `Hello word!`

Read Getting started with coffeescript online: https://riptutorial.com/coffeescript/topic/4233/getting-started-with-coffeescript

Chapter 2: Arrays

Examples

Mapping values

You want to convert all elements in an array to some other form.

For example, you have

```
theUsers = [
  {id: 1, username: 'john'}
  {id: 2, username: 'lexy'}
  {id: 3, username: 'pete'}
]
```

and you want to have an array of usernames only, i.e.

```
['john', 'lexy', 'pete']
```

Method 1 - using `.map`

```
theUsernames = theUsers.map (user) -> user.username
```

Method 2 - using comprehension

```
theUsernames = (user.username for user in theUsers)
```

Filtering values

```
theUsers = [
  {id: 1, username: 'john'}
  {id: 2, username: 'lexy'}
  {id: 3, username: 'pete'}
]
```

To retain only users whose id is greather than 2, use the following:

```
[{id: 3, username: 'pete'}]
```

Method 1 - using `.filter`

```
filteredUsers = theUsers.filter (user) -> user.id >= 2
```

Method 2 - using comprehension

```
filteredUsers = (user for user in theUsers when user.id >= 2)
```

Slicing

If you want to extract a subset of an array (i.e. `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]`) you can easily do this with one of the following examples:

- `numbers[0..2]` will return `[1, 2, 3]`
- `numbers[3...-2]` will return `[3, 4, 5, 6]`
- `numbers[-2..]` will return `[8, 9]`
- `numbers[..]` will return `[1, 2, 3, 4, 5, 6, 7, 8, 9]`

With two dots (3..6), the range is inclusive `[3, 4, 5, 6]`
With three dots (3...6), the range excludes the end `[3, 4, 5]`
Adding a `-` to the range will start the count at the end of the array
An omitted first index defaults to zero
An omitted second index defaults to the size of the array

The same syntax can be used with assignment to replace a segment of an array with new values

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

The above row will replace the numbers array with the following : `[1, 2, -3, -4, -5, -6, 7, 8, 9]`

Concatenation

You want to combine arrays into one.

For example, you have

```
fruits = ['Broccoli', 'Carrots']
spices = ['Thyme', 'Cinnamon']
```

and you want to combine them into

```
ingredients = ['Broccoli', 'Carrots', 'Thyme', 'Cinnamon']
```

Method 1 - using `.concat`

```
ingredients = fruits.concat spices
```

Method 2 - using splats

```
ingredients = [fruits..., spices...]
```

Method 3 - using `.concat` with indeterminate number of arrays

If the number of arrays can vary, e.g. you have array of arrays:

```
arrayOfArrays = [[1], [2,3], [4]]
[].concat.apply([], arrayOfArrays) # [1, 2, 3, 4]
```

Comprehensions

You can do neat things via the results of Array "comprehensions"...

Like assign multiple variables... from the result of a looping `for` statement...

```
[express,_] = (require x for x in ['express','underscore'])
```

Or a syntactically sweet version of a "mapped" function call, etc...

```
console.log (x.nme for x in [{nme:'Chad',rnk:99}, {nme:'Raul', rnk:9}])
```

```
[ 'Chad', 'Raul' ]
```

Notice the `( )` surrounding these statements. These parenthesis are required to make the enclosed comprehension "work".

Read Arrays online: https://riptutorial.com/coffeescript/topic/4459/arrays

Chapter 3: Classes

Examples

Classes, Inheritance, and Super

CoffeeScript provides a basic class structure that allows you to name your class, set the superclass, assign prototypal properties, and define the constructor, in a single assignable expression.

Small example below:

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

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

This will show 4 popups:

1. Slithering...
2. Sammy the Python moved 5m.
3. Galloping...
4. Tommy the Palomino moved 45m.

Prototypes

If you feel the need to extend an object's prototype, `::` gives you quick access to an it so you can add methods to it and later use this method on all instances of that method.

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

The above example will give you the ability to use the dasherize method on all Strings. This will

replace all underscores to dashes.

Read Classes online: https://riptutorial.com/coffeescript/topic/5158/classes

Chapter 4: Conditionals

Examples

if, if / then, if / else, unless, ternary operator

The most basic instance of an `if` construct evaluates a condition and executes some code according to the condition outcome. If the condition returns `true`, the code within the conditional is executed.

```
counter = 10
if counter is 10
  console.log 'This will be executed!'
```

The `if` construct can be enriched with an `else` statement. The code within the `else` statement will be executed whenever the `if` condition is not met.

```
counter = 9
if counter is 10
  console.log 'This will not be executed...'
else
  console.log '... but this one will!'
```

`if` constructs can be chained using `else`, without any limitation on how many can be chained. The first conditional that returns `true` will run its code and stop the check: no conditional below that point will be evaluated thereafter, and no code block from withing those conditionals will be executed.

```
if counter is 10
  console.log 'I counted to 10'
else if counter is 9
  console.log 'I counted to 9'
else if counter < 7
  console.log 'Not to 7 yet'
else
  console.log 'I lost count'
```

The opposite form of `if` is `unless`. Unlike `if`, `unless` will only run if the conditional returns `false`.

```
counter = 10
unless counter is 10
  console.log 'This will not be executed!
```

The `if` statements can be placed in a single line, but in this case, the `then` keyword is required.

```
if counter is 10 then console.log 'Counter is 10'
```

An alternative syntax is the Ruby-like:

```
console.log 'Counter is 10' if counter is 10
```

The last two blocks of code are equivalent.

The ternary operator is a compression of an `if / then / else` construct, and can be used when assigning values to variables. The final value assigned to the variable will be the one defined after the `then` when the `if` condition is met. Otherwise, the value after the `else` will be assigned.

```
outcome = if counter is 10 then 'Done counting!' else 'Still counting'
```

Switch

TL; DR: CoffeeScript `switch` statements use `when` for each case and `else` for the default case. They use `then` for one-line cases and commas for multiple cases with a single outcome. They intentionally disallow fallthrough and so don't need an explicit `break` (since it's always there implicitly). A switch statement can be used as a returnable, assignable expression.

CoffeeScript `switch` statements are a sort of control statement that allows you to take different actions based on a value. They are like `if` statements, but where an `if` statement usually takes one of two actions based on whether something is `true` or `false`, `switch` statements take one of any number of actions depending on the value of any expression - a string, number, or anything at all.

CoffeeScript `switch` start with the keyword `switch` followed by the expression to switch on. Then, each case is represented by the keyword `when` followed by the value for that case.

```
switch name
  when "Alice"
    # Code here will run when name is Alice
    callAlice()
  when "Bob"
    # Code here will run when name is Bob
    giveBobSandwich()
```

There is also a shorthand syntax for when each case is one line, using the `then` keyword instead of a newline:

```
livesLeft = 2
switch livesLeft
  when 3 then fullHealth()
  when 2 then healthAt 2
  when 1 then healthAt 1
  when 0 then playerDie()
```

You can mix and match the two formats as necessary:

```
livesLeft = 2
switch livesLeft
  when 3 then fullHealth()
  when 2 then healthAt 2
  when 1
    healthAt 1
```

```
    alert "Warning! Health low!"
  when 0 then playerDie()
```

Although the most common things to switch on are a variable (as in the previous example) or the result of a functoin, you can switch on any expression you choose:

```
indexOfAnswer = 0
switch indexOfAnswer + 1
  when 1 then console.log "The answer is the 1st item"
  when 2 then console.log "The answer is the 2nd item"
  when 3 then console.log "The answer is the 3rd item"
```

You can also have multiple cases lead to the same action:

```
switch password
  when "password", "123456", "letmein" then console.log "Wrong!"
  when "openpoppyseed" then console.log "Close, but no cigar."
  when "opensesame" then console.log "You got it!"
```

A very useful feature is a default or catch-all case, that will only execute if none of the other criteria are met. CoffeeScript signifies this with the `else` keyword:

```
switch password
  when "password", "123456", "letmein" then console.log "Wrong!"
  when "openpoppyseed" then console.log "Close, but no cigar."
  when "opensesame" then console.log "You got it!"
  else console.log "Not even close..."
```

(Note that you don't need the `then` keyword for the `else` case because there is no condition.)

Now here's an example of all the features of `switch` in action!

```
switch day
  when "Mon" then go work
  when "Tue" then go relax
  when "Thu" then go iceFishing
  when "Fri", "Sat"
    if day is bingoDay
      go bingo
      go dancing
  when "Sun" then go church
  else go work
```

You can also have the condition of a case be an expression:

```
switch fullName
  when myFullName() then alert "Doppelgänger detected"
  when presidentFirstName + " " + presidentLastName
    alert "Get down Mr. president!"
    callSecretService()
  when "Joey Bonzo" then alert "Joey Bonzo everybody"
```

CoffeeScript `switch` statements also have a unique trait: they can return values like a function. If

you assign a variable to a `switch` statement, then it will be assigned whatever the statement returns.

```
address = switch company
  when "Apple" then "One Infinite Loop"
  when "Google" then "1600 Amphitheatre Parkway"
  when "ACME"
    if isReal
      "31918 Hayman St"
    else
      "Unknown desert location"
  else lookUpAddress company
```

(Remember that the last statement in a block is implicitly returned. You can also use the `return` keyword manually.)

Switch statements can also be used without a control expression, turning them in to a cleaner alternative to if/else chains.

```
score = 76
grade = switch
  when score < 60 then 'F'
  when score < 70 then 'D'
  when score < 80 then 'C'
  when score < 90 then 'B'
  else 'A'
```

(This is functionally equivalent to `grade = switch true` because the first case that evaluates to `true` will match. However, since each case implicitly `break`s at the end, only the first case to match will be executed.)

Read Conditionals online: https://riptutorial.com/coffeescript/topic/4317/conditionals

Chapter 5: Destructuring Assignment

Examples

Swap

When you assign an array or object literal to a value, CoffeeScript breaks up and matches both sides against each other, assigning the values on the right to the variables on the left.

```
# Swap
[x, y] = [y, x]
```

Extract Values from an Object

```
person =
  name: "Duder von Broheim"
  age: 27
  address: "123 Fake St"
  phoneNumber: "867-5309"

{name, age, address, phoneNumber} = person
```

Named Function Parameters

CoffeeScript allows to deconstruct objects and arrays when they are fed to functions as arguments.

A function that leverages deconstruction will specify in its signature all the fields that are expected within its body. When invoking such function, an object or array containing all the expected fields has to be passed as argument.

```
drawRect = ({x, y, width, height}) ->
  # here you can use the passed parameters
  # color will not be visible here!

myRectangle =
  x: 10
  y: 10
  width: 20
  height: 20
  color: 'blue'

drawRect myRectangle
```

```
printTopThree = ([first, second, third]) ->
  # here you can use the passed parameters
  # 'Scrooge McDuck' will not be visible here!

ranking = ['Huey', 'Dewey', 'Louie', 'Scrooge McDuck']
```

```
printTopThree ranking
```

First and Last Element

```
array = [1, 2, 3, 4]

[first] = array # 1

[..., last] = array # 4

[first, middle..., last] = array # first is 1, middle is [2, 3], last is 4
```

Read Destructuring Assignment online: https://riptutorial.com/coffeescript/topic/4461/destructuring-assignment

Chapter 6: Functions

Examples

Small Arrow functions

```
# creates a function with no arguments, which returns 3
get_three = () ->
    return 3

# same as above
get_three = -> 3

# creates a function with arguments
add_three = (num) -> num + 3

# multiple arguments, etc.
add = (a, b) -> a + b
```

Read Functions online: https://riptutorial.com/coffeescript/topic/5723/functions

Chapter 7: Loops

Examples

Looping a Function

The following codes will output the numbers 1 through 10 in the console, although `console.log` could be any function that accepts an input.

Method 1 - Standard

```
for x in [1..10]
    console.log x
```

Method 2 - Compact

```
console.log x for x in [1..10]
```

Read Loops online: https://riptutorial.com/coffeescript/topic/6006/loops

Chapter 8: Operators

Examples

Existential Operator

CoffeeScript's existential operator `?` check if the variable is **null** or **undefined**.

1. Check for `null` or `undefined`.

```
alert "Hello CoffeeScript!" if myVar?
```

javascript equivalent:

```
if (typeof myVar !== "undefined" && myVar !== null) {
  alert("Hello CoffeeScript!");
}
```

2. Safer conditional assignment

You can also use this operator safer conditional assignment

```
language = favoriteLanguage ? "coffeescript"
```

javascript equivalent:

```
language = typeof favoriteLanguage !== "undefined" && favoriteLanguage !== null ?
favoriteLanguage : "coffeescript";
```

3. Safe chaining of methods

Instead of chaining the methods with `.` chain them with `?.` to avoid raising the **TypeError**.

```
firstName = user?.profile?.firstname
```

javascript equivalent:

```
firstName = typeof user !== "undefined" && user !== null ? (ref = user.profile) != null ?
ref.firstname() : void 0 : void 0;
```

If all of the properties exist then you'll get the expected result if the chain is broken, **undefined** is returned

Full list of default operators

CoffeeScript	JavaScript
`is, ==`	`===`
`isnt, !=`	`!==`
`not`	`!`
`and`	`&&`
`or`	`||`
`true, yes, on`	`true`
`false, no, off`	`false`
`@, this`	`this`
`of`	`in`
`in`	*No equivalent*
`a ** b`	`Math.pow(a, b)`
`a // b`	`Math.floor(a / b)`
`a %% b`	`(a % b + b) % b`

Read Operators online: https://riptutorial.com/coffeescript/topic/4915/operators

Chapter 9: Pro's & Con's of using Coffeescript

Examples

Pros

Simplicity

Probably the best part of CoffeeScript is its simplicity. CoffeeScript allows for a more concise and simplistic syntax than plain JavaScript. One simple but surprisingly time-saving feature is that CoffeeScript has no need for `;` or `{}`, eliminating the need to spend hours finding out the place from which a `}` is missing.

Loops

Creating a loop that outputs the value of each item in an array unless the value is "monkey" in CoffeeScript is very easy.

```
animals = ["dog", "cat", "monkey", "squirrel"]
for item in animals when item isnt "monkey"
    console.log item
```

in CoffeeScript compiles to

```
var animals, i, item, len;

animals = ["dog", "cat", "monkey", "squirrel"];

for (i = 0, len = animals.length; i < len; i++) {
    item = animals[i];
    if (item !== "monkey") {
        console.log(item);
    }
}
```

in JavaScript, but they both output

```
dog
cat
squirrel
```

String Interpolation

CoffeeScript:

```
"Hello, #{user}, how are you today?"
```

JavaScript:

```
"Hello, " + user + ", how are you today?";
```

Read Pro's & Con's of using Coffeescript online: https://riptutorial.com/coffeescript/topic/6278/pro-s---con-s-of-using-coffeescript

Chapter 10: Strings

Examples

Placeholder replacements

Placeholders can be used in strings to automatically substitute the values in the final string.

```
container = "cup"
liquid = "coffee"
string = "Filling the #{container} with #{liquid}..."
```

The above String - when printed - will say: `Filling the cup with coffee...`

You can even use Coffee-script inside these placeholders

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

Block strings

Block strings can be used to hold formatted or indentation-sensitive text (or, if you just don't feel like escaping quotes and apostrophes). The indentation level that begins the block is maintained throughout, so you can keep it all aligned with the body of your code.

```
html = """
       <strong>
         cup of coffeescript
       </strong>
       """
```

Multiline strings

Multiline strings are allowed in CoffeeScript. Lines are joined by a single space unless they end with a backslash. Indentation is ignored.

```
mobyDick = "Call me Ishmael. Some years ago --
  never mind how long precisely -- having little
  or no money in my purse, and nothing particular
  to interest me on shore, I thought I would sail
  about a little and see the watery part of the
  world..."
```

Read Strings online: https://riptutorial.com/coffeescript/topic/5062/strings

Credits

S. No	Chapters	Contributors
1	Getting started with coffeescript	Artisan72, b3by, Badacadabra, Community, Kevin Chavez
2	Arrays	4444, Alex Gray, fracz, fzzle, Molske
3	Classes	Molske
4	Conditionals	b3by, c0d3rman, Molske, Vyren
5	Destructuring Assignment	b3by, Daniel X Moore
6	Functions	Kevin Chavez
7	Loops	Vyren
8	Operators	Deepak Mahakale, fzzle, Molske
9	Pro's & Con's of using Coffeescript	Vyren
10	Strings	Max Dudzinski, Molske