CoffeeScript

"a little language that compiles into JavaScript"

Mark Volkmann
Object Computing, Inc.

Main Benefits ...

- Only exposes the "good parts" of JavaScript
  - no `==` or `!=` which perform type coercions
  - no accidental creation of global variables
  - each source file has its own scope
    - compiled output is wrapped in a function
- Less verbose than JavaScript
  - requires about one-third less code
  - eliminates many noise characters - `{ } ( ) ; function`
    - and eliminates JavaScript issue with missing semicolons
  - function bodies and blocks are indicated with indentation instead of braces
- Can use all JavaScript libraries
  - such as jQuery and Node.js
- Generates code that passes JavaScript Lint
  - http://www.javascriptlint.com doesn't check indentation

... Main Benefits

- Mostly one-to-one mapping to JavaScript code
  - uses standard JavaScript objects
    - `Array`, `Date`, `Function`, `Math`, `Number`, `Object`, `RegExp`, `String`
  - exception handling is same
    - `try`, `catch`, `throw`
  - generated code is very readable
  - no loss in performance
- Can mix CoffeeScript and JavaScript
  - though this isn’t typically needed
  - surround JavaScript code with back-ticks
    - can span multiple lines
- All expressions have a value
  - even `if` statements and loops
- Easier to model classes and inheritance

Brief History

- Created by Jeremy Ashkenas
    - first released on 12/25/2009
- Compiler
    - original written in Ruby
    - rewritten in CoffeeScript and released in March 2010

Endorsed By

- Brendan Eich - Mozilla
  - creator of JavaScript
  - "CoffeeScript is well done and more convenient to use than JS, provided you buy into the Python-esque significant space and the costs of generating JS from another source language. But semantically it's still JS."
  - "CoffeeScript is smart and fun new clothing for JavaScript. Clothes are important, and I'd be the first to proclaim that JavaScript needs wardrobe help."
  - "I believe CoffeeScript and other front ends for JS have a bright future"
  - working on adding what he considers the good parts of CoffeeScript into Harmony, a future version of JavaScript
    - http://brendaneich.com/2011/05/my-jsconf-us-presentation/

in a sense, the compile step just takes the place of running a lint tool on JavaScript code which is recommended

- David Heinemeier Hanson - 37signals
  - creator of Ruby on Rails web framework
  - "Enter CoffeeScript: a pre-compiler that removes all the unnecessary verbosity of JavaScript and simply makes it a pleasure to write and read"
  - "Yes, it's true, Rails 3.1 is going to ship with CoffeeScript and SCSS in the box ... It's bad ass."

Installing

- Install Node.js

```
git clone https://github.com/joyent/node.git
cd node
./configure
make
sudo make install
```

because the `coffee` tool runs on top of Node.js

- Install Node Package Manager (npm)

```
curl http://npmjs.org/install.sh | sudo sh
```

because CoffeeScript is bundled as a Node package

- Install CoffeeScript

```
sudo npm install -g coffee-script
```

  - verify install by running `coffee -v`
  - to update later, `npm update -g coffee-script`

Editor Support

- Provides
  - syntax highlighting
  - smart indentation
  - compilation shortcuts
- See list at https://github.com/jashkenas/coffee-script/wiki/Text-editor-plugins
- Includes
  - Cloud9IDE
  - Emacs
  - Intellij IDEA
  - NetBeans
  - TextMate
  - Vim - adds **`:CoffeeCompile`** and **`:CoffeeRun`** commands

Running

- Start REPL with **`coffee`**
  - ctrl-d to exit

to enter multi-line statements, terminate all but last with a backslash

- Run a script with **`coffee`** ***`file-path`***
  - file extension defaults to **`.coffee`**
- Get help on command-line options with **`coffee -h`**
  - **`-c`** for compile
  - **`-w`** for watch
  - **`-o`** to specify output directory
  - and many more
- Can compile all **`.coffee`** files under a given directory to **`.js`** files in another, maintaining directory structure
  - **`coffee -cwo js cs`**
    - continuously compiles modified **`.coffee`** files under **`cs`** directory and writes generated **`.js`** files under **`js`** directory
  - **`coffee -cwo . .`**
    - for **`.coffee`** and .js files under current directory

Comments

- Single-line start with #
- Multi-line are preceded by a line containing ### and followed by the same
  - convenient for alternating between contiguous sections of code

New Operators

Relational operators

- **`is`** instead of **`===`**
- **`isnt`** instead of **`!==`**

Other operators

- **`?`** - existential operator (described later)
- **`in`** - tests whether a given value is in an array
- **`of`** - tests whether a given property is in an object

Logical operators

- **`not`** instead of **`!`**
- **`and`** instead of **`&&`**
- **`or`** instead of **`||`**
- **`and=`** instead of **`&&=`**
- **`or=`** instead of **`||=`**

Ternary operator replaced

- instead of
  `condition ? true-value : false-value`
  write
  `if` ***`condition`*** **`then`** ***`true-value`*** **`else`** ***`false-value`***

`x or= y` assigns **`y`** to **`x`** only if **`x`** has no value (also see "soaks" later)

`x and= y` assigns **`y`** to **`x`** only if **`x`** has a value

Strings

- Double-quoted strings can use interpolation

```
name = 'Mark'
console.log "Hello, #{name}"
```

  - can have any expression inside `${ }`
- Single-quoted strings cannot use interpolation
- Multiline strings can be created using three quote characters on each end
  - can use interpolation if double-quote characters are used
  - great for generating HTML

```
wife = 'Tami'
letter = """
  Dear #{wife},
    Do I need to pick up milk on the way home?

  Your loving husband #{name}
"""
```

removes white space from beginning of lines equal to number of whitespace characters at beginning of first line; doesn't include a newline after last line

Statement Modifiers

- Can add these modifiers to the end of statements
  - `if` *`condition`*
  - `unless` *`condition`*
  - `while` *`condition`*
  - `until` *`condition`*

```
n = 3
console.log n if n > 0
console.log n unless n > 5
console.log(n--) until n is 0
console.log(n++) while n < 3
```

Output:

```
3
3
3
2
1
0
1
2
```

Defining and Calling Functions

- Syntax to define is ***`name = (parameters) -> code`***
  - for example, a function to cube a number
  - **`cube = (n) -> Math.pow n, 3`**
  - great syntax for implementing callbacks!
  - note the use of the JavaScript **`Math`** object and the lack of parens and curly braces
- Syntax to call is ***`name arguments`*** or ***`name(arguments)`***
  - arguments are separated by commas
  - for example, **`cube n`**
  - need parens if no arguments; otherwise it is interpreted as a reference to the function object, not a call to it
- Implicitly returns value of last expression
- Multi-line function definitions
  - use whitespace to indent; convention is 2 spaces

just like in JavaScript functions, all arguments can be accessed using the array-like **`arguments`** object

functions must be defined before they are called!

```
odds = (numbers) ->
  result = []
  for n in numbers
    result.push n if n % 2 is 1
  result
```

Function Definition Order

- Function definitions must appear before they are called
  - this works fine

```
f = (n) ->
  console.log 'in f'
  g(n - 1) if n > 0

g = (n) ->
  console.log 'in g'
  f(n - 1) if n > 0

f 5
```

Default Parameter Values

- Function parameters can have default values
  - not just on parameters at end
  - pass `null` to take default value for parameters not at end

```
# distance defaults to marathon.
# time defaults to one hour.
calculatePace = (distance = 26.2, time = 60) -> time / distance

console.log calculatePace 3.1, 17.6 # 5K in 17.6 minutes
console.log calculatePace 8 # 8 miles in 1 hour
console.log calculatePace null, 180 # marathon in 3 hours
console.log calculatePace() # wow, that's fast!
```

```
// generated JavaScript
var calculatePace;
calculatePace = function(distance, time) {
  if (distance == null) {
    distance = 26.2;
  }
  if (time == null) {
    time = 60;
  }
  return distance / time;
};
```

Splats

- Used in parameter lists to collect a variable number of arguments into a real JavaScript array
  - alternative to using **`arguments`** object
- Only one parameter can use splats, but it doesn't have to be the last one

```
sumEndsAndMiddle = (first, middle..., last) ->
  [ first + last, middle.reduce (a, b) -> a + b ]

console.log sumEndsAndMiddle(1, 2, 3, 4, 5) # [6, 9]
```

- Can also be used in a function call to expand an array into individual arguments

```
distance = (x1, y1, x2, y2) ->
  Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2))
point1 = [3, 4]
point2 = [1, 5]
console.log distance(point1..., point2...) # 2.236
```

Simulating Named Parameters

- Write function to accepted an object
- Pass key/value pairs in a literal hash

```
f = (params) ->
  console.log params.name if params.name

f color: 'yellow', name: 'Mark', number: 19

f
  color: 'yellow'
  name: 'Mark'
  number: 19
```

Chained Comparisons

- Can use more than one relational operator without a logical operator
- Instead of ...

```
validDiceRoll = x > 0 and x <= 6
```

- Can write ...

```
validDiceRoll = 0 < x <= 6
```

Equality

- JavaScript has many operators for testing equality
  - some perform type coercions and using them is discouraged
  - == and `!=` perform type coercions; === and `!==` do not
- CoffeeScript avoids this confusion
  - instead of ===, use **`is`**
  - instead of **`!==`**, use **`isnt`**
  - unfortunately CoffeeScript supports == and `!=`,
    but changes their meaning to be the same as === and `!==` in JavaScript
    which is confusing!
    - avoid those and always use **`is`** or **`isnt`**

Property Access

- Properties of an object are accessed just like in JavaScript
  - dot notation - ***`object.property`***
  - bracket notation - `object['property']`
- Inside a function where **`this`** refers to the object
  - can use `@property`

JSON ...

- CoffeeScript supports an alternative, indented style of JSON
- Instead of ...

```
person = {
  name: 'Mark Volkmann',
  address: {
    street: '123 Some Street',
    zip: 12345
  }
}
```

- Can write ...

```
person =
  name: 'Mark Volkmann'
  address:
    street: '123 Some Street'
    zip: 12345
```

can be on one line if the content of each of these lines is separated by commas

Can be tricky when passing to a function! Entries that overflow a line cannot have indentation.

```
# This doesn't parse.
f name: 'Mark'
  number: 19

# These do parse.
f name: 'Mark', number: 19

f
  name: 'Mark'
  number: 19

f name: 'Mark'
number: 19 # not preferred!
```

... JSON

- Can omit JSON keys if each of these is true
  - keys are valid names
  - values are in variables with same names as keys
  - braces are used

```
name = 'Mark Volkmann'
phone = '123-456-7890'
info = { name, phone }
# equivalent to the following
info = {
  name: name,
  phone: phone
}
```

Soaks ?

- Can write expressions that succeed even when the value of a variable is null or undefined, a function returns null, or an object doesn't have a given method
- Use `?` operator, also referred to as the existential operator

```
pujols = {}
pujols.swing = -> 'home run'
carpenter = {}
players =
  'Pujols': pujols
  'Carpenter': carpenter

# Object found and has swing method.
console.log players['Pujols']?.swing?() # home run
# Object found but doesn't have swing method.
console.log players['Carpenter']?.swing?()  # undefined
# Object not found.
console.log players['Molina']?.swing?() # undefined
```

Another Use
`x ?= y`
assigns `y` to `x`
only if `x` doesn't already have a value
(same as `x or= y`)

Ranges

- Can create arrays containing ranges of consecutive numbers (syntax borrowed from Ruby)
- Inclusive upper bound - `[`***`start`***`..`***`end`***`]`
- Exclusive upper bound - `[`***`start`***`...`***`end`***`]`

`start` can be greater than ***`end`***

- bounds must be integers, not variables
- Examples
  - `[2..5]` gives `[2, 3, 4, 5]`
  - `[2...5]` gives `[2, 3, 4]`
- Can create ranges that go backwards
  - `[5..2]` gives `[5, 4, 3, 2]`
  - `[5...2]` gives `[5, 4, 3]`
- Ranges can be used to “slice” values from arrays and strings
  - `s = 'abcdef'; s[2..4]` gives `'cde'`

Iteration ...

- Over array values

```
for value in array [by step]
  # use value
```

step can be negative only if the array was created by a range

- Over object properties

```
for [own] key of object
  # use key

for [own] key, value of object
  # use key and value
```

using the own keyword is equivalent to wrapping the loop body in
`if object.hasOwnProperty(key)`

key and ***value*** are variables in the current scope, not scoped to the **`for`** loop

- Each of the **`for`** lines above can end with **`when`** ***`condition`***
  - filters out iterations where condition evaluates to false
  - alternative to wrapping loop body in an **`if`** statement

```
for n in [1..100] when n % 3 is 0
  # process multiples of 3
```

… Iteration

- Can call a function on each iteration value in an array

```
function(value) for value in array [by step]
```

- While loop

```
s = 'test'
while s.length
  console.log s.substr(0, 1)
  s = s.substr 1
```

- Endless loop
  - only escape with **break** or **return**

```
i = 3
loop
  console.log i
  break if i is 0
  i--
```

Collection Content Testing

- `in` and `of` are also operators that evaluate to a boolean value
- To determine whether an array contains a given value,
  `value in array`
- To determine whether an object contains a given property,
  `property of object`
  - ***`property`*** can be the name of a function

```
console.log 4 in [1, 4, 7] # true

obj =
  foo: 1
  bar: 2
console.log 'bar' of obj # true
```

Comprehensions

- The value of each kind of loop is an array containing the value of the last expression in the body for each iteration

```
squares = for n in [1..3]
  compute? n # a no-op since function doesn't exist
  n * n
console.log squares # [1, 4, 9]
squares = n * n for n in [1..3] # same
```

- Comprehensions are another way to specify the value to be collected from each iteration
  - works with any kind of loop: **`for`**, **`while`**, **`until`** and **`loop`**

```
console.log n * 2 for n in [1..10] when n % 3 is 0
# [6, 12, 18]
```

Pattern Matching

in JavaScript 1.7 and already implemented in Firefox

- Provides an easy way to extract values from an array or object
- A.k.a. destructuring

```
values = ['St. Louis', 'Cardinals', 'baseball']
[city, team, sport] = values
console.log "The #{team} play #{sport} in #{city}."
```

- Can be used to swap values

```
x = 1
y = 2
[x, y] = [y, x]
console.log "x=#{x}, y=#{y}"
```

- Even works when arrays and objects are nested inside each other to any depth

```
obj =
  name: 'Mark Volkmann'
  address:
    street: '123 Street'
    zip: 12345

{name: n, address: {street: s, zip: z}} = obj
console.log "The person at #{s} is #{n}."

{name, address: {street, zip}} = obj
console.log "The person at #{street} is #{name}."
```

Adding Methods to a Prototype

- `obj::x` is the same as `obj.prototype.x`

```
// CoffeeScript
String::startsWith = (prefix) ->
  new RegExp("^#{prefix}").test this

console.log 'foobar'.startsWith('foo')
console.log 'barbaz'.startsWith('foo')
```

Output

```
true
false
```

```
// generated JavaScript
String.prototype.startsWith = function(prefix) {
  return new RegExp("^" + prefix).test(this);
};
console.log("foobar".startsWith("foo"));
console.log("barbaz".startsWith("foo"));
```

Classes

- Classes in CoffeeScript are compiled to a common JavaScript pattern for modeling them
- JavaScript refresher
  - global variables are actually properties of the "root object"
    - **`window`** in browsers, **`global`** in Node.js
  - constructors are functions whose name, by convention, starts uppercase
  - objects are created by calling a constructor with the **`new`** keyword
  - methods are added to a "class" by assigning them to the prototype object of the constructor function object

```javascript
// JavaScript
Rocket = function () {};
Rocket.prototype.launch = function () {
  console.log('3, 2, 1, Blast Off!');
};
var r1 = new Rocket();
r1.launch();
```

```coffeescript
// CoffeeScript
Rocket = ->
Rocket::launch = ->
  console.log '3, 2, 1, Blast Off!'
r1 = new Rocket
r1.launch()
```

CoffeeScript Classes

```
class Rocket
  constructor: (@name) ->
    @launchCount = 0 # object variable
    Rocket.count++ # class variable
  launch: ->
    console.log "#{@name} 3, 2, 1, Blast Off!"
    @launchCount++ # object variable
  report: ->
    console.log "#{@name} was launched #{@launchCount} time(s)"
  @count = 0 # class variable

r1 = new Rocket 'Helicat'
r2 = new Rocket 'Eggscaliber'
r1.launch(); r2.launch(); r1.launch()
console.log "#{Rocket.count} rockets were created"
r1.report(); r2.report()
```

can only have one constructor

when a constructor parameter name starts with `@`, it is automatically assigned to an object variable with the same name

Output

```
Helicat 3, 2, 1, Blast Off!
Eggscaliber 3, 2, 1, Blast Off!
Helicat 3, 2, 1, Blast Off!
2 rockets were created
Helicat was launched 2 time(s)
Eggscaliber was launched 1 time(s)
```

`@` is the same as `this`
`@x` is the same as `this.x`

Generated JavaScript

```
var Rocket, r1, r2;
Rocket = (function() {
  function Rocket(name) {
    this.name = name;
    this.launchCount = 0;
    Rocket.count++;
  }
  Rocket.prototype.launch = function() {
    console.log("" + this.name +
      " 3, 2, 1, Blast Off!");
    return this.launchCount++;
  };
  Rocket.prototype.report = function() {
    return console.log("" + this.name +
      " was launched " + this.launchCount + " time(s)");
  };
  Rocket.count = 0;
  return Rocket;
})();
```

```
r1 = new Rocket('Helicat');
r2 = new Rocket('Eggscaliber');
r1.launch();
r2.launch();
r1.launch();
console.log("" + Rocket.count +
  " rockets were created");
r1.report();
r2.report();
```

Class Inheritance

- Utilizes the prototype chain of objects, just like JavaScript
- Call **`super`** anywhere inside a constructor or method
  to call corresponding thing in superclass
  - with no parens or arguments, all arguments passed to it are passed on to superclass method
- **`instanceof`** operator
  - can be used to test whether an object is an instance of a given class
    or one that extends from a given class

```
class AdvancedRocket extends Rocket
  constructor: (name, @stages) ->
    super name + '!'
  launch: ->
    super
    for stage in [2..@stages]
      console.log("fire stage #{stage}");

r3 = new AdvancedRocket 'Epplin', 3
r3.launch()
```

The generated JavaScript for this is
NOT code you'd want to write yourself!

Output

```
Epplin! 3, 2, 1, Blast Off!
fire stage 2
fire stage 3
```

“Fat Arrow”

- **=>** instead of **->** to define a function
- Fixes value of **this** inside the function to its current value
- Useful for defining callback functions
inside constructors or instance methods
that need to refer to instance variables
or call instance methods
- Example using Node.js

```
events = require 'events'
class Alarm extends events.EventEmitter
  constructor: (ms) ->
    setTimeout (=> @.emit 'ring'), ms

alarm = new Alarm 1000
console.log 'alarm set'
alarm.on 'ring', -> console.log 'alarm rang'
```

need a better example that
shows notification of listeners
when instance state changes

`switch` Statement

- Uses **`when`** instead of **`case`** and **`else`** instead of **`default`**
  - like in Ruby
- Each **`when`** can be followed by a comma-separated list of values
- No colon after value(s)
  - can use **`then`** keyword to place code on same line
- Implicit break at end of code for each **`when`**
  - can't fall through
- Result of last expression evaluated is returned
  - can assign a **`switch`** statement to a variable

```
level = switch r3.stages
  when 1 then 'basic'
  when 2, 3 then 'advanced'
  when 4, 5, 6 then 'crazy'
  else 'highly unlikely'
console.log "level of #{r3.name} is #{level}"
```

same code spread across more lines

```
level = switch r3.stages
  when 1
    'basic'
  when 2, 3
    'advanced'
  when 4, 5, 6
    'crazy'
  else
    'highly unlikely'
```

Debugging

- Currently a challenge
  - compiler stops on first line it can’t parse
    - gives line number and a message that sometimes doesn’t accurately describe the issue
    - gives a stack trace into the compiler
    - expected to improve in the future
  - line numbers in stack traces refer to lines in generated JavaScript, not lines in CoffeeScript source
  - code displayed in debugger of browsers (like Firebug) is the generated JavaScript
- Being addressed
  - Mozilla and WebKit teams are working on adding support for debugging CoffeeScript and other JS-based languages in their browsers (Firefox, Chrome and Safari)
    - http://www.infoq.com/news/2011/08/debug-languages-on-javascript-vm/
- Can use
  - **`console.assert`**
  - a Node.js logging module or **`node-inspector`** for server-side code

Runtime Compilation in Browsers

- CoffeeScript compiler can be downloaded as part of web page
  - `<script src="coffee-script.js"></script>`
  - get from http://jashkenas.github.com/coffee-script/extras/coffee-script.js
- Allows CoffeeScript files to be referenced directly instead of pre-compiling them to JavaScript
  - `<script src="whatever.coffee"></script>`
- Fine for development ... too slow for production use
  - but may want to run compiler to check for syntax errors anyway before testing in browser
- In Chrome
  - may need to start browser with `-allow-file-access-from-files` option if `.coffee` files are local instead of being served via HTTP

Runtime Compilation in Node.js

- Can call **`require`** on a CoffeeScript file if **`require('coffee-script')`** has been called
  - don't need to specify **`.coffee`** file extension

mine.coffee

```
# This is a Node module written in CoffeeScript.
exports.shoutOut = -> console.log 'Hello from CoffeeScript!'
```

```
// This is a JavaScript Node client that uses the module.
require('coffee-script');
var mine = require('./mine');
mine.shoutOut();
```

client.js

run with "**`node client`**"

Running as Scripts

- On Unix-like systems, if first line is proper “shebang”, can run like a shell script
  - looks for `coffee` executable in PATH
  - file must have execute privilege
  - the file below is named “`script`”
  - run with `./script`

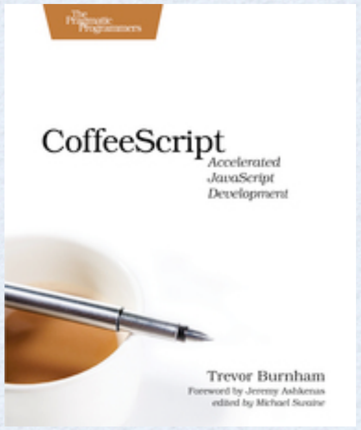
```
#!/usr/bin/env coffee
console.log 'The script ran!'
```

Won’t JS Skill Be Lost?

- Ability to read JS won’t be affected much
  - syntax is somewhat close
  - still use same methods on same core objects
    - `Array`, `Date`, `Function`, `Math`, `Number`, `Object`, `RegExp`, `String`
  - still need to learn about JS libraries that will be used with CoffeeScript so will be continually reading example JS code
- Ability to write JS will be affected more
  - but can write in CoffeeScript and compile to JS to see equivalent, good JS

Resources

- Books
  - CoffeeScript: Accelerated JavaScript Development
    - Trevor Burnham, Pragmatic Programmers, 2011



- Websites
  - main - http://jashkenas.github.com/coffee-script/
  - see "TRY COFFEESCRIPT" tab that allows entering CoffeeScript code in browser and viewing generated JavaScript as you type!
  - style guide - https://github.com/polarmobile/coffeescript-style-guide
  - Code School - http://coffeescript.codeschool.com/