CoffeeScript Cookbook

Welcome

Welcome to the CoffeeScript Cookbook! CoffeeScript recipes for the community *by* the community. Head over to the Contributing page and see what you can do to help out!

- Basic Server
- Basic HTTP Server

12. Design Patterns

- Builder Pattern
- Bridge Pattern
- Decorator Pattern
- Interpreter Pattern
- Command Pattern
- Factory Method Pattern
- Strategy Pattern
- Memento Pattern
- Singleton Pattern

13. Databases

- MongoDB
- SQLite

Don't see a recipe you want? See an entire missing chapter? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Syntax

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Embedding JavaScript

Problem

You want to include some found/pre-written JavaScript code inline with your CoffeeScript.

Solution

Wrap the JavaScript with backticks:

```
`function greet(name) {
return "Hello "+name;
}`

# Back to CoffeeScript
greet "Coffee"
# => "Hello Coffee"
```

Discussion

This is a simple way to integrate small snippets of JavaScript code into your CoffeeScript without converting it over to use CoffeeScript syntax. As shown in the CoffeeScript Language Reference you can mix to the two languages to a certain extent:

```
hello = `function (name) {
return "Hello "+name
}`
hello "Coffee"
# => "Hello Coffee"
```

Here the `hello` variable is still in CoffeeScript, but is assigned a function written in JavaScript.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Comparing Ranges

Problem

You want to know if a variable is inside a given range.

Solution

Use CoffeeScript's chained comparison syntax.

```
maxDwarfism = 147
minAcromegaly = 213

height = 180

normalHeight = maxDwarfism < height < minAcromegaly
# => true
```

Discussion

This is a nice feature lifted from Python. Instead of writing out the full comparison like

```
normalHeight = height > maxDwarfism && height < minAcromegaly
```

CoffeeScript allows us to chain the two comparisons together in a form that more closely matches the way a mathematician would write it.

CoffeeScript Cookbook

For Loops

Problem

You need to iterate over an array, object or range with a for loop.

Solution

```
# for(i = 1; i<= 10; i++)
x for x in [1..10]
# => [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

# To count by 2
# for(i=1; i<= 10; i=i+2)
x for x in [1..10] by 2
# => [ 1, 3, 5, 7, 9 ]

# Perform a simple operation like squaring each item.
x * x for x in [1..10]
# = > [1,4,9,16,25,36,49,64,81,100]
```

Discussion

Comprehensions replace for loops in CoffeeScript, but they simply compile into the traditional javascript equivalent for-loop.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Classes and Objects

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Class Methods and Instance Methods

Problem

You want to create a class methods and instance methods.

Solution

```
class Songs
  @_titles: 0    # Although it's directly accessible, the leading _ defines it by convention as private property.

  @get_count: ->
    @_titles

  constructor: (@artist, @title) ->
    Songs._titles++

Songs.get_count()
# => 0

song = new Songs("Rick Astley", "Never Gonna Give You Up")
Songs.get_count()
# => 1

song.get_count()
# => TypeError: Object #<Songs> has no method 'get_count'
```

Discussion

Coffeescript will store class methods (also called static methods) on the object itself rather than on the object prototype (and thus on individual object instances), which conserves memory and gives a central location to store class-level values.

Cloning an object (deep copy)

Problem

You want to clone an object with all its sub-objects.

Solution

```
clone = (obj) ->
  if not obj? or typeof obj isnt 'object'
    return obj

  newInstance = new obj.constructor()

  for key of obj
    newInstance[key] = clone obj[key]

  return newInstance

x =
  foo: 'bar'
  bar: 'foo'

y = clone(x)

y.foo = 'test'

console.log x.foo isnt y.foo, x.foo, y.foo
# => true, bar, test
```

Discussion

The difference between copying an object through assignment and through this clone-function is how they handle references. The assignment only copies the object's reference, whereas the clone-function creates a complete new object by

- creating a new object like the source object,
- copying all attributes form the source object to the new object and
- repeating these steps for all sub-objects by calling the clone-function recursively.

Example of an assignment copy:

```coffee
x =
  foo: 'bar'
  bar: 'foo'

y = x

y.foo = 'test'

console.log x.foo isnt y.foo, x.foo, y.foo
# => false, test, test
```

As you can see, when you change `y` after the copy, you also change `x`.

Create an object literal if it does not already exist

Problem

You want to initialize an object literal, but you do not want to overwrite the object if it already exists.

Solution

Use the Existential operator

```
window.MY_NAMESPACE ?= {}
```

Discussion

This is equivalent to the following JavaScript:

```
window.MY_NAMESPACE = window.MY_NAMESPACE || {};
```

Common JavaScript technique, using object literal to define a namespace. This saves us from clobbering the namespace if it already exists.

CoffeeScript Cookbook

A CoffeeScript type function

Problem

You'd like to know the type of a function without using typeof. (See http://javascript.crockford.com/remedial.html for more information on why typeof is pretty inferior.)

Solution

Use the following function:

```
type = (obj) ->
  if obj == undefined or obj == null
    return String obj
  classToType = new Object
  for name in "Boolean Number String Function Array Date RegExp".split(" ")
    classToType["[object " + name + "]"] = name.toLowerCase()
  myClass = Object.prototype.toString.call obj
  if myClass of classToType
    return classToType[myClass]
  return "object"
```

Discussion

This function was modeled on jQuery's $.type function. (http://api.jquery.com/jQuery.type/)

Note that, as an alternative to type checking, you can often use duck typing and the existential operator together to eliminating the need to examine an object's type, in certain cases. For example, here is exception-free code that pushes an element to an array, if myArray is in fact an array (or array-like, with a push function), and does nothing otherwise.

```
myArray?.push? myValue
```


Class Variables

Problem

You want to create a class variable.

Solution

```coffeescript
class Zoo
  @MAX_ANIMALS: 50
  MAX_ZOOKEEPERS: 3

Zoo.MAX_ANIMALS
# => 50

Zoo.MAX_ZOOKEEPERS
# => undefined (it is an instance variable)

zoo = new Zoo
zoo.MAX_ZOOKEEPERS
# => 3
```

Discussion

Coffeescript will store these values on the object itself rather than on the object prototype (and thus on individual object instances), which conserves memory and gives a central location to store class-level values.

Chaining Calls to an Object

Problem

You want to call multiple methods on a single object without having to reference that object each time.

Solution

Return the `this` (i.e. `@`) object after every chained method.

```
class CoffeeCup
 properties:
  strength: 'medium'
  cream: false
  sugar: false
 strength: (newStrength) ->
  @properties.strength = newStrength
  @
 cream: (newCream) ->
  @properties.cream = newCream
  @
 sugar: (newSugar) ->
  @properties.sugar = newSugar
  @

morningCup = new CoffeeCup()

morningCup.properties # => { strength: 'medium', cream: false, sugar: false }

eveningCup = new CoffeeCup().strength('dark').cream(true).sugar(true)

eveningCup.properties # => { strength: 'dark', cream: true, sugar: true }
```

Discussion

The jQuery library uses a similar approach by returning a selector object from every relevant method, modifying it as subsequent methods tweak the selection:

```
$('p').filter('.topic').first()
```

For your own objects, a touch of metaprogramming can automate the setup process and explicitly state the purpose of returning *this*.

```coffeescript
addChainedAttributeAccessor = (obj, propertyAttr, attr) ->
 obj[attr] = (newValues...) ->
  if newValues.length == 0
   obj[propertyAttr][attr]
  else
   obj[propertyAttr][attr] = newValues[0]
   obj

class TeaCup
 properties:
  size: 'medium'
  type: 'black'
  sugar: false
  cream: false

addChainedAttributeAccessor(TeaCup.prototype, 'properties', attr) for attr of TeaCup.prototype.properties

earlgrey = new TeaCup().size('small').type('Earl Grey').sugar('false')

earlgrey.properties # => { size: 'small', type: 'Earl Grey', sugar: false }

earlgrey.sugar true

earlgrey.sugar() # => true
```

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Strings

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Repeating a String

Problem

You want to repeat a string.

Solution

Create an array of n+1 nulls, and then join it with the repetition string as the glue:

```
# create a string of 10 foos
Array(11).join 'foo'

# => "foofoofoofoofoofoofoofoofoo"
```

Discussion

JavaScript lacks a string repeat function, as does CoffeeScript. List comprehensions and maps can be pressed into service here, but in the case of a simple string repeat it's easier to simply build an array of n+1 nulls and then glue them together.

CoffeeScript Cookbook

Lowercasing a String

Problem

You want to lowercase a string.

Solution

Use JavaScript's String toLowerCase() method:

```
"ONE TWO THREE".toLowerCase()
# => 'one two three'
```

Discussion

`toLowerCase()` is a standard JavaScript method. Don't forget the parentheses.

Syntax Sugar

You can add some Ruby-like syntax sugar with the following shortcut:

```
String::downcase = -> @toLowerCase()
"ONE TWO THREE".downcase()
# => 'one two three'
```

The snippet above demonstrates a few features of CoffeeScript:

- The double-colon `::` is shorthand for saying `.prototype.`
- The "at" sign `@` is shorthand for saying `this.`

The code above compiles in to the following JavaScript:

```
String.prototype.downcase = function() {
  return this.toLowerCase();
};
"ONE TWO THREE".downcase();
```

Note: Although it's quite common in languages like Ruby, extending native objects is often considered bad practice in JavaScript (see: Maintainable JavaScript: Don't modify objects you don't own; Extending built-in native objects. Evil or not?).

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Uppercasing a String

Problem

You want to uppercase a string.

Solution

Use JavaScript's String toUpperCase() method:

```
"one two three".toUpperCase()
# => 'ONE TWO THREE'
```

Discussion

`toUpperCase()` is a standard JavaScript method. Don't forget the parentheses.

Syntax Sugar

You can add some Ruby-like syntax sugar with the following shortcut:

```
String::upcase = -> @toUpperCase()
"one two three".upcase()
# => 'ONE TWO THREE'
```

The snippet above demonstrates a few features of CoffeeScript:

- The double-colon `::` is shorthand for saying `.prototype.`
- The "at" sign `@` is shorthand for saying `this.`

The code above compiles in to the following JavaScript:

```
String.prototype.upcase = function() {
  return this.toUpperCase();
};
"one two three".upcase();
```

Note: Although it's quite common in languages like Ruby, extending native objects is often considered bad practice in JavaScript (see: Maintainable JavaScript: Don't modify objects you don't own; Extending built-in native objects. Evil or not?).

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Finding Substrings

Problem

You need to find the first or last occurrence of a search string within a message.

Solution

Use Javascript's indexOf() and lastIndexOf() to find the first and last occurrences of a string, respectively.
Syntax: string.indexOf searchstring, start

```
message = "This is a test string. This has a repeat or two. This might even have a third."
message.indexOf "This", 0
# => 0

# Modifying the start parameter
message.indexOf "This", 5
# => 23

message.lastIndexOf "This"
# => 49
```

Discussion

Still need recipe to count occurrences of a given string within a message.

Trimming whitespace from a String

Problem

You want to trim whitespace from a string.

Solution

Use JavaScript's Regular Expression support to replace whitespace.

To trim leading and trailing whitespace, use the following:

```
"  padded string  ".replace /^\s+|\s+$/g, ""
# => 'padded string'
```

To trim only leading whitespace, use the following:

```
"  padded string  ".replace /^\s+/g, ""
# => 'padded string  '
```

To trim only trailing whitespace, use the following:

```
"  padded string  ".replace /\s+$/g, ""
# => '  padded string'
```

Discussion

Opera, Firefox and Chrome all have a native string prototype `trim` method, and the other browsers could add one as well. For this particular method, I would use the built-in method where possible, otherwise create a polyfill:

```
unless String::trim then String::trim = -> @replace /^\s+|\s+$/g, ""

"  padded string  ".trim
# => 'padded string'
```

Syntax Sugar

You can add some Ruby-like syntax sugar with the following shortcuts:

```coffeescript
String::strip = -> if String::trim? then @trim() else @replace /^\s+|\s+$/g, ""
String::lstrip = -> @replace /^\s+/g, ""
String::rstrip = -> @replace /\s+$/g, ""

"  padded string  ".strip()
# => 'padded string'
"  padded string  ".lstrip()
# => 'padded string  '
"  padded string  ".rstrip()
# => '  padded string'
```

For an interesting discussion and benchmarks of JavaScript `trim` performance, see this blog post by Steve Levithan.

Capitalizing Words

Problem

You want to capitalize the first letter of every word in a string.

Solution

Use the split, map, join pattern: Split the string into words, then use a map to capitalize the first letter and lowercase all other letters of each word before gluing the string back together with join.

```
("foo bar baz".split(' ').map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '
# => 'Foo Bar Baz'
```

Or do the same thing using a list comprehension:

```
(word[0].toUpperCase() + word[1..-1].toLowerCase() for word in "foo   bar   baz".split /\s+/).join ' '
# => 'Foo Bar Baz'
```

Discussion

Split, map, join is a common scripting pattern dating back to perl. This function may benefit from being placed directly onto the String class by Extending Classes.

Be aware that two wrinkles can appear in the split, map, join pattern. The first is that the split text works best when it is constant. If the source string has multiple spaces in it, the split will need to take this into account to prevent getting extra, empty words. One way to do this is with a regular expression to split on runs of whitespace instead of a single space:

```
("foo    bar    baz".split(/\s+/).map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '
# => 'Foo Bar Baz'
```

…but this leads us to the second wrinkle: notice that the runs of whitespace are now compressed down to a single character by the join.

Quite often one or both of these wrinkles is acceptable, however, so the split, map, join pattern can be a powerful tool.

Splitting a String

Problem

You want to split a string.

Solution

Use JavaScript's String split() method:

```
"foo bar baz".split " "
# => [ 'foo', 'bar', 'baz' ]
```

Discussion

String's split() is a standard JavaScript method. It can be used to split a string on any delimiter, including regular expressions. It also accepts a second parameter that specifies the number of splits to return.

```
"foo-bar-baz".split "-"
# => [ 'foo', 'bar', 'baz' ]
```

```
"foo   bar  \t baz".split /\s+/
# => [ 'foo', 'bar', 'baz' ]
```

```
"the sun goes down and I sit on the old broken-down river pier".split " ", 2
# => [ 'the', 'sun' ]
```

String Interpolation

Problem

You want to create a string that contains a text representation of a CoffeeScript Variable.

Solution

Use CoffeeScript's ruby-like string interpolation instead of JavaScript's string addition.

Interpolation:

```
muppet = "Beeker"
favorite = "My favorite muppet is #{muppet}!"

# => "My favorite muppet is Beeker!"
```

```
square = (x) -> x * x
message = "The square of 7 is #{square 7}."

# => "The square of 7 is 49."
```

Discussion

CoffeeScript interpolates strings in similar fashion to ruby. Most expressions are valid inside the `#{...}` interpolation syntax.

CoffeeScript permits multiple expressions inside the interpolation which can have side effects, but this is discouraged. Only the last value will be returned.

```
# You can do this, but don't. YOU WILL GO MAD.
square = (x) -> x * x
muppet = "Beeker"
message = "The square of 10 is #{muppet='Animal'; square 10}. Oh, and your favorite muppet is now #{muppet}."

# => "The square of 10 is 100. Oh, and your favorite muppet is now Animal."
```

Matching Strings

Problem

You want to match two or more strings.

Solution

Calculate the edit distance, or number of operations required to transform one string into the other.

```
Levenshtein =
  (str1, str2) ->

    l1 = str1.length
    l2 = str2.length

    Math.max l1, l2 if Math.min l1, l2 == 0

    i = 0; j = 0; distance = []

    for i in [0...l1 + 1]
     distance[i] = []
     distance[i][0] = i

    distance[0][j] = j for j in [0...l2 + 1]

    for i in [1...l1 + 1]
     for j in [1...l2 + 1]
       distance[i][j] = Math.min distance[i - 1][j] + 1,
         distance[i][j - 1] + 1,
         distance[i - 1][j - 1] +
           if (str1.charAt i - 1) == (str2.charAt j - 1) then 0 else 1

    distance[l1][l2]
```

Discussion

You can use either Hirschberg or Wagner–Fischer's algorithm to calculate a Levenshtein distance. This example uses Wagner–Fischer's algorithm.

Is this recipe wrong, incomplete, or non-idiomatic? Help fix it by reading the Contributor's Guide!

Generating a Unique ID

Problem

You want to generate a random unique identifier.

Solution

You can create a Base 36 encoded string from a random number.

```
uniqueId = (length=8) ->
  id = ""
  id += Math.random().toString(36).substr(2) while id.length < length
  id.substr 0, length

uniqueId()     # => n5yjla3b
uniqueId(2)    # => 0d
uniqueId(20)   # => ox9eo7rt3ej0pb9kqlke
uniqueId(40)   # => xu2vo4xjn4g0t3xr74zmndshrqlivn291d584alj
```

Discussion

There are other possible techniques, but this is relatively performant and flexible.

CoffeeScript Cookbook

Arrays

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Creating a String from an Array

Problem

You want to create a string from an array.

Solution

Use JavaScript's Array toString() method:

```
["one", "two", "three"].toString()
# => 'one,two,three'
```

Discussion

`toString()` is a standard JavaScript method. Don't forget the parentheses.

Python-like zip function

Problem

You want to zip together multiple arrays into an array of arrays, similar to Python's zip function. Python's zip function returns an array of tuples, where each tuple contains the i-th element from each of the argument arrays.

Solution

Use the following CoffeeScript code:

```
# Usage: zip(arr1, arr2, arr3, ...)
zip = () ->
  lengthArray = (arr.length for arr in arguments)
  length = Math.max.apply(Math, lengthArray)
  argumentLength = arguments.length
  results = []
  for i in [0...length]
    semiResult = []
    for arr in arguments
      semiResult.push arr[i]
    results.push semiResult
  return results

zip([0, 1, 2, 3], [0, -1, -2, -3])
# => [[0, 0], [1, -1], [2, -2], [3, -3]]
```


Max Array Value

Problem

You need to find the largest value contained in an array.

Solution

You can use Math.max() JavaScript method along with splats.

```
Math.max [12, 32, 11, 67, 1, 3]...
# => 67
```

Alternatively, it's possible to use ES5 `reduce` method. For backward compatibility with older JavaScript implementations, use Math.max.apply:

```
# ECMAScript 5
[12,32,11,67,1,3].reduce (a,b) -> Math.max a, b
# => 67

# Pre-ES5
Math.max.apply(null, [12,32,11,67,1,3])
# => 67
```

Discussion

`Math.max` compares every argument and returns the largest number from arguments. The ellipsis (`...`) converts every array value into argument which is given to the function. You can also use it with other functions which take variable ammount of arguments, such as `console.log`.

Using Arrays to Swap Variables

Problem

You want to use an array to swap variables.

Solution

Use CoffeeScript's destructuring assignment syntax:

```
a = 1
b = 3

[a, b] = [b, a]

a
# => 3

b
# => 1
```

Discussion

Destructuring assignment allows swapping two values without the use of a temporary variable.

This can be useful when traversing arrays and ensuring iteration only happens over the shortest one:

```
ray1 = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
ray2 = [ 5, 9, 14, 20 ]

intersection = (a, b) ->
  [a, b] = [b, a] if a.length > b.length
  value for value in a when value in b

intersection ray1, ray2
# => [ 5, 9 ]

intersection ray2, ray1
# => [ 5, 9 ]
```

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Define Ranges Array

Problem

You want to define a range in an array.

Solution

There are two ways to define a range of array elements in CoffeeScript.

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

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

We can also reverse the range of element by writing it this way.

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

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

Discussion

Inclusive range always define by '..' operator.

Exclusive range define by '…', and always omit the last value.

Shuffling Array Elements

Problem

You want to shuffle the elements in an array.

Solution

The JavaScript Array `sort()` method accepts a custom sort function. We can write a `shuffle()` method to add some convenience.

```
Array::shuffle = -> @sort -> 0.5 - Math.random()

[1..9].shuffle()
# => [ 3, 1, 5, 6, 4, 8, 2, 9, 7 ]
```

Discussion

For more background on how this shuffle logic works, see this discussion at StackOverflow.

Note: Although it's quite common in languages like Ruby, extending native objects is often considered bad practice in JavaScript (see: Maintainable JavaScript: Don't modify objects you don't own; Extending built-in native objects. Evil or not?).

CoffeeScript Cookbook

Mapping Arrays

Problem

You have an array of objects and want to map them to another array, similar to Ruby's map.

Solution

Use map() with an anonymous function, but don't forget about list comprehensions.

```
electric_mayhem = [ { name: "Doctor Teeth", instrument: "piano" },
                    { name: "Janice", instrument: "lead guitar" },
                    { name: "Sgt. Floyd Pepper", instrument: "bass" },
                    { name: "Zoot", instrument: "sax" },
                    { name: "Lips", instrument: "trumpet" },
                    { name: "Animal", instrument: "drums" } ]

names = electric_mayhem.map (muppet) -> muppet.name
# => [ 'Doctor Teeth', 'Janice', 'Sgt. Floyd Pepper', 'Zoot', 'Lips', 'Animal' ]
```

Discussion

Because CoffeeScript has clean support for anonymous functions, mapping an array in CoffeeScript is nearly as easy as it is in Ruby.

Maps are are good way to handle complicated transforms and chained mappings in CoffeeScript. If your transformation is as simple as the one above, however, it may read more cleanly as a list comprehension.

Reducing Arrays

Problem

You have an array of objects and want to reduce them to a value, similar to Ruby's `reduce()` and `reduceRight()`.

Solution

You can simply use Array's `reduce()` and `reduceRight()` methods along with an anonoymous function, keeping the code clean and readable. The reduction may be something simple such as using the + operator with numbers or strings.

```
[1,2,3,4].reduce (x,y) -> x + y
# => 10
```

```
["words", "of", "bunch", "A"].reduceRight (x, y) -> x + " " + y
# => 'A bunch of words'
```

Or it may be something more complex such as aggregating elements from a list into a combined object.

```
people =
    { name: 'alec', age: 10 }
    { name: 'bert', age: 16 }
    { name: 'chad', age: 17 }

people.reduce (x, y) ->
    x[y.name]= y.age
    x
, {}
# => { alec: 10, bert: 16, chad: 17 }
```

Discussion

Javascript introduced `reduce` and `reduceRight` in version 1.8. Coffeescript provides a natural and simple way to express anonymous functions. Both go together cleanly in the problem of merging a collection's items into a combined result.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Filtering Arrays

Problem

You want to be able to filter arrays based on a boolean condition.

Solution

Use Array.filter (ECMAScript 5):

```
array = [1..10]

array.filter (x) -> x > 5
# => [6,7,8,9,10]
```

In pre-EC5 implementations, extend the Array prototype to add a filter function which will take a callback and perform a comprension over itself, collecting all elements for which the callback is true. Be sure to check if the function is already implemented before overwriting it:

```
# Extending Array's prototype
unless Array::filter
  Array::filter = (callback) ->
    element for element in this when callback(element)

array = [1..10]

# Filter odd elements
filtered_array = array.filter (x) -> x % 2 == 0
# => [2,4,6,8,10]

# Filter elements less than or equal to 5:
gt_five = (x) -> x > 5
filtered_array = array.filter gt_five
# => [6,7,8,9,10]
```

Discussion

This is similar to using ruby's Array#select method.

Reversing Arrays

Problem

You want to reverse an array.

Solution

Use JavaScript's Array reverse() method:

```
["one", "two", "three"].reverse()
# => ["three", "two", "one"]
```

Discussion

`reverse()` is a standard JavaScript method. Don't forget the parentheses.

Concatenating Arrays

Problem

You want to join two arrays together.

Solution

There are two standard options for concatenating arrays in JavaScript.

The first is to use JavaScript's Array `concat()` method:

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
array3 = array1.concat array2
# => [1, 2, 3, 4, 5, 6]
```

Note that `array1` is *not* modified by the operation. The concatenated array is returned as a new object.

If you want to merge two arrays without creating a new object, you can use the following technique:

```
array1 = [1, 2, 3]
array2 = [4, 5, 6]
Array::push.apply array1, array2
array1
# => [1, 2, 3, 4, 5, 6]
```

In the example above, the `Array.prototype.push.apply(a, b)` approach modifies `array1` in place without creating a new array object.

We can simplify the pattern above using CoffeeScript by creating a new `merge()` method for Arrays.

```
Array::merge = (other) -> Array::push.apply @, other

array1 = [1, 2, 3]
array2 = [4, 5, 6]
array1.merge array2
array1
# => [1, 2, 3, 4, 5, 6]
```

Discussion

CoffeeScript lacks a special syntax for joining arrays, but `concat()` and `push()` are standard JavaScript methods.

CoffeeScript Cookbook

Testing Every Element

Problem

You want to be able to check that every element in an array meets a particular condition.

Solution

Use Array.every (ECMAScript 5):

```
evens = (x for x in [1..10] by 2)

evens.every (x)-> x % 2 == 0
# => true
```

Array.every was addded to Mozilla's Javascript 1.6 and made standard with EcmaScript 5. If you to support browsers that do not implement EC5 then check out `_.all` from underscore.js.

For a real world example, prentend you have a multiple select list that looks like:

```
<select multiple id="my-select-list">
  <option>1</option>
  <option>2</option>
  <option>Red Car</option>
  <option>Blue Car</option>
</select>
```

Now you want to verify that the user selected only numbers. Let's use Array.every:

```
validateNumeric = (item)->
  parseFloat(item) == parseInt(item) && !isNaN(item)

values = $("#my-select-list").val()

values.every validateNumeric
```

Discussion

This is similar to using ruby's Array#all? method.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

List Comprehensions

Problem

You have an array of objects and want to map them to another array, similar to Python's list comprehensions.

Solution

Use a list comprehension, but don't forget about mapping arrays.

```
electric_mayhem = [ { name: "Doctor Teeth", instrument: "piano" },
                    { name: "Janice", instrument: "lead guitar" },
                    { name: "Sgt. Floyd Pepper", instrument: "bass" },
                    { name: "Zoot", instrument: "sax" },
                    { name: "Lips", instrument: "trumpet" },
                    { name: "Animal", instrument: "drums" } ]

names = (muppet.name for muppet in electric_mayhem)
# => [ 'Doctor Teeth', 'Janice', 'Sgt. Floyd Pepper', 'Zoot', 'Lips', 'Animal' ]
```

Discussion

Because CoffeeScript directly support list comprehensions, they work pretty much as advertised wherever you would use one in Python. For simple mappings, list comprehensions are much more readable. For complicated transformations or for chained mappings, mapping arrays might be more elegant.

Removing duplicate elements from Arrays

Problem

You want to remove duplicate elements from an array.

Solution

```
Array::unique = ->
  output = {}
  output[@[key]] = @[key] for key in [0...@length]
  value for key, value of output

[1,1,2,2,2,3,4,5,6,6,6,"a","a","b","d","b","c"].unique()
# => [ 1, 2, 3, 4, 5, 6, 'a', 'b', 'd', 'c' ]
```

Discussion

There are many implementations of the `unique` method in JavaScript. This one is based on "The fastest method to find unique items in array" found here.

Note: Although it's quite common in languages like Ruby, extending native objects is often considered bad practice in JavaScript (see: Maintainable JavaScript: Don't modify objects you don't own; Extending built-in native objects. Evil or not?).

CoffeeScript Cookbook

Dates and Times

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Calculate the date of Easter Sunday

Problem

You need to find the month and day of the Easter Sunday for given year.

Solution

The following function returns array with two elements: month (1-12) and day of the Easter Sunday. If no arguments are given result is for the current year. This is an implementation of Anonymous Gregorian algorithm in CoffeeScript.

```
gregorianEaster = (year = (new Date).getFullYear()) ->
  a = year % 19
  b = ~~(year / 100)
  c = year % 100
  d = ~~(b / 4)
  e = b % 4
  f = ~~((b + 8) / 25)
  g = ~~((b - f + 1) / 3)
  h = (19 * a + b - d - g + 15) % 30
  i = ~~(c / 4)
  k = c % 4
  l = (32 + 2 * e + 2 * i - h - k) % 7
  m = ~~((a + 11 * h + 22 * l) / 451)
  n = h + l - 7 * m + 114
  month = ~~(n / 31)
  day = (n % 31) + 1
  [month, day]
```

Discussion

NB! Javascript numbers months from 0 to 11 so .getMonth() for date in March will return 2, this function will return 3. You can modify the function if you want this to be consistent.

The function uses ~~ trick instead of Math.floor().

```
gregorianEaster()    # => [4, 24] (April 24th in 2011)
gregorianEaster 1972 # => [4, 2]
```

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Calculate the date of Thanksgiving (USA and Canada)

Problem

You need to calculate when is Thanksgivinig in given year.

Solution

The following functions return the day of Thanksgiving for a given year. If no year is given then current year is used.

In the USA Thanksgiving is celebrated on the fourth Thursday in November:

```
thanksgivingDayUSA = (year = (new Date).getFullYear()) ->
  first = new Date year, 10, 1
  day_of_week = first.getDay()
  22 + (11 - day_of_week) % 7
```

In Canada it is the second Monday in October:

```
thanksgivingDayCA = (year = (new Date).getFullYear()) ->
    first = new Date year, 9, 1
    day_of_week = first.getDay()
    8 + (8 - day_of_week) % 7
```

Discussion

```
thanksgivingDayUSA() #=> 24 (November 24th, 2011)

thanksgivingDayCA() # => 10 (October 10th, 2011)

thanksgivingDayUSA(2012) # => 22 (November 22nd)

thanksgivingDayCA(2012) # => 8 (October 8th)
```

The idea is very simple:

1. Find out what day of the week is the first day of respective month (November for USA, October for Canada).

2. Calculate offset from that day to the next occurrence of weekday required (Thursday for USA, Monday for Canada).
3. Add that offset to the first possible date of the holiday (22nd for USA Thanksgiving, 8th for Canada).

Chapter Index | Contributing | Authors | License

Finding the Last Day of the Month

Problem

You need to find the last day of the month, but don't want to keep a lookup table of the number of days in each month of the year.

Solution

Use JavaScript's Date underflow to find the -1th day of the following month:

```
now = new Date
lastDayOfTheMonth = new Date(1900+now.getYear(), now.getMonth()+1, -1)
```

Discussion

JavaScript's Date constructor cheerfully handles overflow and underflow conditions, which makes date math very easy. Given this ease of manipulation, it doesn't make sense to worry about how many days are in a given month; just nudge the math around. In December, the solution above will actually ask for the -1th day of the 13th month of the current year, which works out to the -1th day of January of NEXT year, which works out to the 31st day of December of the current year.

Get Days Between two Dates

Problem

You need to find how much seconds minutes, hours, days, months or years has passed between two dates.

Solution

Use JavaScript's Date function getTime(). Which provides how much time in miliseconds has passed since 01/01/1970:

```
DAY = 1000 * 60 * 60  * 24

d1 = new Date('02/01/2011')
d2 = new Date('02/06/2011')

days_passed = Math.round((d2.getTime() - d1.getTime()) / DAY)
```

Discussion

Using miliseconds makes the life easier to avoid overflow mistakes with Dates. So we first calculate how much miliseconds has a day. Then, given two distincit dates, just get the diference in miliseconds betwen two dates and then divide by how much miliseconds has a day. It will get you the days between two distinct dates.

If you'd like to calculate the hours between two dates objects you can do that just by dividing the diference in miliseconds by the convertion of miliseconds to hours. The same goes to minutes and seconds.

```
HOUR = 1000 * 60 * 60

d1 = new Date('02/01/2011 02:20')
d2 = new Date('02/06/2011 05:20')

hour_passed = Math.round((d2.getTime() - d1.getTime()) / HOUR)
```

Finding Last (or Next) Month

Problem

You need to calculate a relative date range like “last month” or “next month”.

Solution

Add or subtract from the current month, secure in the knowledge that JavaScript’s Date constructor will fix up the math.

```
# these examples were written in GMT-6
# Note that these examples WILL work in January!
now = new Date
# => "Sun, 08 May 2011 05:50:52 GMT"

lastMonthStart = new Date 1900+now.getYear(), now.getMonth()-1, 1
# => "Fri, 01 Apr 2011 06:00:00 GMT"

lastMonthEnd = new Date 1900+now.getYear(), now.getMonth(), 0
# => "Sat, 30 Apr 2011 06:00:00 GMT"
```

Discussion

JavaScript Date objects will cheerfully handle underflows and overflows in the month and day fields, and will adjust the date object accordingly. You can ask for the 42nd of March, for example, and will get the 11th of April.

JavaScript Date objects store the year as the number of years since 1900, the month as an integer from 0 to 11, and the date (day of month) as an integer from 1 to 31. In the solution above, last_month_start is obtained by asking for the first day of a month in the current year, but the month is -1 to 10. If month is -1 the Date object will actually return December of the previous year:

```
lastNewYearsEve = new Date 1900+now.getYear(), -1, 31
# => "Fri, 31 Dec 2010 07:00:00 GMT"
```

The same is true for overflows:

```
thirtyNinthOfFourteember = new Date 1900+now.getYear(), 13, 39
# => "Sat, 10 Mar 2012 07:00:00 GMT"
```

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Math

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Faster Fibonacci algorithm

Problem

You would like to calculate a number N in the Fibonacci sequence but want to do it quickly.

Solution

The following solution (which can still be improved on) was originally talked about on Robin Houston's blog.

Here are a few links talking about the algorithm and ways to improve it:

- http://bosker.wordpress.com/2011/04/29/the-worst-algorithm-in-the-world/
- http://www.math.rutgers.edu/~erowland/fibonacci
- http://jsfromhell.com/classes/bignumber
- http://www.math.rutgers.edu/~erowland/fibonacci
- http://bigintegers.blogspot.com/2010/11/square-division-power-square-root
- http://bugs.python.org/issue3451

This code is in gist form here: https://gist.github.com/1032685

```
###
Author: Jason Giedymin <jasong _a_t_ apache -dot- org>
        http://www.jasongiedymin.com
        https://github.com/JasonGiedymin

This CoffeeScript Javascript Fast Fibonacci code is
based on the python code from Robin Houston's blog.
See below links.

A few things I want to introduce in time are implementions of
Newtonian, Burnikel / Ziegler, and Binet's algorithms on top
of a Big Number framework.

Todo:
- https://github.com/substack/node-bigint
- BZ and Newton mods.
- Timing

###

MAXIMUM_JS_FIB_N = 1476

fib_bits = (n) ->
 #Represent an integer as an array of binary digits.
```

```
 bits = []
 while n > 0
     [n, bit] = divmodBasic n, 2
     bits.push bit

   bits.reverse()
   return bits

fibFast = (n) ->
 #Fast Fibonacci

 if n < 0
  console.log "Choose an number >= 0"
  return

 [a, b, c] = [1, 0, 1]

 for bit in fib_bits n
     if bit
      [a, b] = [(a+c)*b, b*b + c*c]
     else
      [a, b] = [a*a + b*b, (a+c)*b]

     c = a + b
    return b

divmodNewton = (x, y) ->
 throw new Error "Method not yet implemented yet."

divmodBZ = () ->
 throw new Error "Method not yet implemented yet."

divmodBasic = (x, y) ->
 ###
 Absolutely nothing special here. Maybe later versions will be Newtonian or
 Burnikel / Ziegler _if_ possible...
 ###

 return [(q = Math.floor x/y), (r = if x < y then x else x % y)]

start = (new Date).getTime();
calc_value = fibFast(MAXIMUM_JS_FIB_N)
diff = (new Date).getTime() - start;
console.log "[#{calc_value}] took #{diff} ms."
```

Discussion

Questions?

Questions?

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

A random integer function

Problem

You'd like to get a random integer between two integers, inclusive.

Solution

Use the following function.

{% highlight coffeescript %} randomInt = (lower, upper=0) -> start = Math.random() if not lower? lower, upper = 0, lower if lower > upper lower, upper = upper, lower return Math.floor(start * (upper - lower + 1) + lower)

(randomInt(1) for i in 0…10)

=> 0,1,1,0,0,0,1,1,1,0

(randomInt(1, 10) for i in 0…10)

=> 7,3,9,1,8,5,4,10,10,8

Discussion

Questions?

CoffeeScript Cookbook

Generating Random Numbers

Problem

You need to generate a random number in a certain range.

Solution

Use JavaScript's Math.random() to get floating-point numbers from 0 <= x < 1.0. Use multiplication and Math.floor to get a number in a certain range.

```
probability = Math.random()
0.0 <= probability < 1.0
# => true

# Note that percentile does NOT ever reach 100. A full range of 0 to 100 is actually a span of 101.
percentile = Math.floor(Math.random() * 100)
0 <= percentile < 100
# => true

dice = Math.floor(Math.random() * 6) + 1
1 <= dice <= 6
# => true
```

Discussion

This is a straight lift from JavaScript.

Note that JavaScripts's Math.random() does not allow you to seed the random number generator to force certain values. See Generating Predictable Random Numbers for that.

To generate a number from 0 up to (but not including) n, multiply by n. To generate a number from 1 to n (inclusive), multiply by n and add 1.

Math Constants

Problem

You need to use common mathematical constants like pi or e.

Solution

Use Javascript's Math object to provide commonly needed mathematical constants.

```
Math.PI
# => 3.141592653589793

# Note: Capitalization matters! This produces no output, it's undefined.
Math.Pi
# =>

Math.E
# => 2.718281828459045

Math.SQRT2
# => 1.4142135623730951

Math.SQRT1_2
# => 0.7071067811865476

# Natural log of 2. ln(2)
Math.LN2
# => 0.6931471805599453

Math.LN10
# => 2.302585092994046

Math.LOG2E
# => 1.4426950408889634

Math.LOG10E
# => 0.4342944819032518
```

Discussion

For another example of how a math constant is used in a real world problem, refer to the Converting Radians and Degrees section of this Math chapter.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Generating Predictable Random Numbers

Problem

You need to generate a random number in a certain range, but you also need to be able to "seed" the generator to deliver predictable values.

Solution

Write your own random number generator. There are a LOT of ways to do this. Here's a simple one. *This generator is +ABSOLUTELY NOT+ acceptable for cryptographic purposes!*

```
class Rand
  # if created without a seed, uses current time as seed
  constructor: (@seed) ->
    # Knuth and Lewis' improvements to Park and Miller's LCPRNG
    @multiplier = 1664525
    @modulo = 4294967296 # 2**32-1;
    @offset = 1013904223
    unless @seed? && 0 <= seed < @modulo
      @seed = (new Date().valueOf() * new Date().getMilliseconds()) % @modulo

  # sets new seed value
  seed: (seed) ->
    @seed = seed

  # return a random integer 0 <= n < @modulo
  randn: ->
    # new_seed = (a * seed + c) % m
    @seed = (@multiplier*@seed + @offset) % @modulo

 # return a random float 0 <= f < 1.0
  randf: ->
    this.randn() / @modulo

  # return a random int 0 <= f < n
  rand: (n) ->
    Math.floor(this.randf() * n)

  # return a random int min <= f < max
  rand2: (min, max) ->
    min + this.rand(max-min)
```

Discussion

JavaScript and CoffeeScript do not provide a seedable random number generator. Writing your own will be an exercise in trading off the amount of randomness with the simplicity of the generator. A full discussion of randomness is beyond the scope of this cookbook; for further reading consult Donald Knuth's *The Art of Computer Programming*, Volume II, Chapter 3, "Random Numbers", and *Numerical Recipes in C*, 2nd Edition, Chapter 7, "Random Numbers".

A brief explanation of this random number generator is in order, however. It is a Linear Congruential Pseudorandom Number Generator. LCPRNG's operate on the mathematical formula `I<sub>j+1</sub> = (aI<sub>j</sub>+c) % m`, where a is the multiplier, c is the addition offset, and m is the modulus. Each time a random number is requested, a very large multiplication and addition are performed – "very large" relative to the key space – and the resulting number is modulused back down into the keyspace.

This generator has a period of 232. It is absolutely unacceptable for cryptographic purposes, but for most simple randomness requirements it is quite adequate. `randn()` will traverse the entire keyspace before repeating itself, and the next number is determined by the previous one.

If you want to tinker with this generator, you are *strongly* encouraged to read Chapter 3 of Knuth's *The Art of Computer Programming*. Random number generation is VERY easy to screw up, and Knuth explains how to tell a good RNG from a bad one.

Avoid the temptation to modulus the output of this generator. If you need an integer range, use division. Linear Congruential generators are very nonrandom in their lower bits. This one in particular always generates an odd number from an even seed, and vice versa. So if you need a random 0 or 1, do NOT use

```
# NOT random! Do not do this!
r.randn() % 2
```

because you will most definitely not get random digits. Use `r.randi(2)` instead.

CoffeeScript Cookbook

Fast Inverse Square Root

Problem

You would like to calculate a the inverse square root of a number quickly.

Solution

Appearing in the Quake III Arena source code, this strange algorithm uses integer operations along with a 'magic number' to calculate floating point approximation values of inverse square roots.

In this CoffeeScript variant I supply the original classic, and newer optimal 32 bit magic numbers found by Chris Lomont. Also supplied is the 64-bit sized magic number.

Another feature included is the ability to alter the level of precision. This is done by controling the number of iterations for performing Newton's method.

Depending on the machine and level of percision this algorithm may still provide performance increases over the classic.

To run this, compile the script with coffee: coffee -c script.coffee

Then copy & paste the compiled js code in to the JavaSript console of your browser.

Note: You will need a browser which supports typed-arrays.

References:

1. ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip
2. http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
3. http://en.wikipedia.org/wiki/Newton%27s_method
4. https://developer.mozilla.org/en/JavaScript_typed_arrays
5. http://en.wikipedia.org/wiki/Fast_inverse_square_root

This code is in gist form here: https://gist.github.com/1036533

```
###

Author: Jason Giedymin <jasong _a_t_ apache -dot- org>
        http://www.jasongiedymin.com
        https://github.com/JasonGiedymin

Appearing in the Quake III Arena source code[1], this strange algorithm uses
integer operations along with a 'magic number' to calculate floating point
approximation values of inverse square roots[5].
```

```
In this CoffeeScript variant I supply the original classic, and newer optimal
32 bit magic numbers found by Chris Lomont[2]. Also supplied is the 64-bit
sized magic number.

Another feature included is the ability to alter the level of precision.
This is done by controling the number of iterations for performing Newton's
method[3].

Depending on the machine and level of percision this algorithm may still
provide performance increases over the classic.

To run this, compile the script with coffee:
    coffee -c <this script>.coffee

Then copy & paste the compiled js code in to the JavaSript console of your
browser.

Note: You will need a browser which supports typed-arrays[4].

References:
[1] ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip
[2] http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf
[3] http://en.wikipedia.org/wiki/Newton%27s_method
[4] https://developer.mozilla.org/en/JavaScript_typed_arrays
[5] http://en.wikipedia.org/wiki/Fast_inverse_square_root

###

approx_const_quake_32 = 0x5f3759df # See [1]
approx_const_32 = 0x5f375a86 # See [2]
approx_const_64 = 0x5fe6eb50c7aa19f9 # See [2]

fastInvSqrt_typed = (n, precision=1) ->
 # Using typed arrays. Right now only works in browsers.
 # Node.JS version coming soon.

    y = new Float32Array(1)
    i = new Int32Array(y.buffer)

    y[0] = n
    i[0] = 0x5f375a86 - (i[0] >> 1)

    for iter in [1...precision]
        y[0] = y[0] * (1.5 - ((n * 0.5) * y[0] * y[0]))

    return y[0]

### Sample single runs ###
testSingle = () ->
```

```
    example_n = 10

    console.log("Fast InvSqrt of 10, precision 1: #{fastInvSqrt_typed(example_n)}")
    console.log("Fast InvSqrt of 10, precision 5: #{fastInvSqrt_typed(example_n, 5)}")
    console.log("Fast InvSqrt of 10, precision 10: #{fastInvSqrt_typed(example_n, 10)}")
    console.log("Fast InvSqrt of 10, precision 20: #{fastInvSqrt_typed(example_n, 20)}")
    console.log("Classic of 10: #{1.0 / Math.sqrt(example_n)}")

testSingle()
```

Discussion

Questions?

Chapter Index | Contributing | Authors | License

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Converting Radians and Degrees

Problem

You need to convert between radians and degrees.

Solution

Use Javascript's Math.PI and a simple formula to convert between the two.

```
# To convert from radians to degrees
radiansToDegrees = (radians) ->
    degrees = radians * 180 / Math.PI

radiansToDegrees(1)
# => 57.29577951308232

# To convert from degrees to radians
degreesToRadians = (degrees) ->
    radians = degrees * Math.PI / 180

degreesToRadians(1)
# => 0.017453292519943295
```

Discussion

Questions?

CoffeeScript Cookbook

Functions

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Recursive Functions

Problem

You want to call a function from within that same function.

Solution

With a named function:

```
ping = ->
 console.log "Pinged"
 setTimeout ping, 1000
```

With an unnamed function, using @arguments.callee@:

```
delay = 1000

setTimeout((->
 console.log "Pinged"
 setTimeout arguments.callee, delay
 ), delay)
```

Discussion

While `arguments.callee` allows for the recursion of anonymous functions and might have the advantage in a very memory-intensive application, named functions keep their purpose more explicit and make for more maintainable code.

When Function Parentheses are not Optional

Problem

You want to call a function that takes no arguments, but don't want to use parentheses.

Solution

Use parentheses anyway.

Another alternative is to utilize the do-notation like so:

```
notify = -> alert "Hello, user!"
do notify if condition
```

This compiles to the following JavaScript:

```
var notify;
notify = function() {
 return alert("Hello, user!");
};
if (condition) {
 notify();
}
```

Discussion

Like Ruby, CoffeeScript allows you to drop parentheses to method calls. Unlike Ruby, however, CoffeeScript treats a bare function name as the pointer to the function. The practical upshot of this is that if you give no arguments to a method, CoffeeScript cannot tell if you want to call the function or use it as a reference.

Is this good or bad? It's just different. It creates an unexpected syntax case – parentheses aren't *always* optional – but in exchange it gives you the ability to pass and receive functions fluently by name, something that's a bit klunky in Ruby.

This usage of the do-notation is a neat approach for CoffeeScript with parenphobia. Some people simply prefer to write out the parentheses in the function call, though.

Splat Arguments

Problem

Your function will be called with a varying number of arguments.

Solution

Use *splats*.

```
loadTruck = (firstDibs, secondDibs, tooSlow...) ->
 truck:
  driversSeat: firstDibs
  passengerSeat: secondDibs
  trunkBed: tooSlow

loadTruck("Amanda", "Joel")
# => { truck: { driversSeat: "Amanda", passengerSeat: "Joel", trunkBed: [] } }

loadTruck("Amanda", "Joel", "Bob", "Mary", "Phillip")
# => { truck: { driversSeat: "Amanda", passengerSeat: "Joel", trunkBed: ["Bob", "Mary", "Phillip"] } }
```

With a trailing argument:

```
loadTruck = (firstDibs, secondDibs, tooSlow..., leftAtHome) ->
 truck:
  driversSeat: firstDibs
  passengerSeat: secondDibs
  trunkBed: tooSlow
 taxi:
  passengerSeat: leftAtHome

loadTruck("Amanda", "Joel", "Bob", "Mary", "Phillip", "Austin")
# => { truck: { driversSeat: 'Amanda', passengerSeat: 'Joel', trunkBed: [ 'Bob', 'Mary', 'Phillip' ] }, taxi: { passengerSeat: 'Austin' } }

loadTruck("Amanda")
# => { truck: { driversSeat: "Amanda", passengerSeat: undefined, trunkBed: [] }, taxi: undefined }
```

Discussion

By adding an ellipsis (`...`) next to no more than one of a function's arguments, CoffeeScript will combine all of the argument values not captured by other named arguments into a list. It will serve up an empty list even if some of the named arguments were not supplied.

CoffeeScript Cookbook

Metaprogramming

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Detecting and Creating Missing Functions

Problem

You want to detect if a function exists and create it if it does not (such as an ECMAScript 5 function in Internet Explorer 8).

Solution

Use `::` to detect the function, and assign to it if it does not exist.

```
unless Array::filter
  Array::filter = (callback) ->
    element for element in this when callback element

array = [1..10]

array.filter (x) -> x > 5
# => [6,7,8,9,10]
```

Discussion

Objects in JavaScript (and thus, in CoffeeScript) have a prototype member that defines what member functions should be available on all objects based on that prototype. In CoffeeScript, you can access the prototype directly via the `::` operator.

Extending Built-in Objects

Problem

You want to extend a class to add new functionality or replace old.

Solution

Use `::` to assign your new function to the prototype of the object or class.

```
String::capitalize = () ->
  (this.split(/\s+/).map (word) -> word[0].toUpperCase() + word[1..-1].toLowerCase()).join ' '

"foo bar     baz".capitalize()
# => 'Foo Bar Baz'
```

Discussion

Objects in JavaScript (and thus, in CoffeeScript) have a prototype member that defines what member functions should be available on all objects based on that prototype. In CoffeeScript, you can access the prototype directly via the `::` operator.

Note: Although it's quite common in languages like Ruby, extending native objects is often considered bad practice in JavaScript (see: Maintainable JavaScript: Don't modify objects you don't own; Extending built-in native objects. Evil or not?).

CoffeeScript Cookbook

jQuery

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

AJAX

Problem

You want to make AJAX calls using jQuery.

Solution

```
$ ?= require 'jquery' # For Node.js compatibility

$(document).ready ->
 # Basic Examples
 $.get '/', (data) ->
  $('body').append "Successfully got the page."

 $.post '/',
  userName: 'John Doe'
  favoriteFlavor: 'Mint'
  (data) -> $('body').append "Successfully posted to the page."

 # Advanced Settings
 $.ajax '/',
  type: 'GET'
  dataType: 'html' error: (jqXHR, textStatus, errorThrown) ->
   $('body').append "AJAX Error: #{textStatus}"
  success: (data, textStatus, jqXHR) ->
   $('body').append "Successful AJAX call: #{data}"
```

jQuery 1.5 and later have added a new, supplemental API for handling different callbacks.

```
request = $.get '/'
request.success (data) -> $('body').append "Successfully got the page again."
request.error (jqXHR, textStatus, errorThrown) -> $('body').append "AJAX Error: ${textStatus}."
```

Discussion

The jQuery and $ variables can be used interchangeably. See also Callback bindings.

Create a jQuery plugin

Problem

You'd like to create jQuery plugin using CoffeeScript

Solution

```
# Reference jQuery
$ = jQuery

# Adds plugin object to jQuery
$.fn.extend
  # Change pluginName to your plugin's name.
  pluginName: (options) ->
    # Default settings
    settings =
      option1: true
      option2: false
      debug: false

    # Merge default settings with options.
    settings = $.extend settings, options

    # Simple logger.
    log = (msg) ->
      console?.log msg if settings.debug

    # _Insert magic here._
    return @each ()->
      log "Preparing magic show."
      # You can use your settings in here now.
      log "Option 1 value: #{settings.option1}"
```

Discussion

Here are a couple of examples of how to use your new plugin.

JavaScript

```
$("body").pluginName({
  debug: true
```

```
  debug: true
};
```

CoffeeScript:

```
$("body").pluginName
  debug: true
```

Chapter Index | Contributing | Authors | License

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

Callback bindings

Problem

You want to bind a callback function to an object.

Solution

```
$ ->
  class Basket
    constructor: () ->
      @products = []

      $('.product').click (event) =>
        @add $(event.currentTarget).attr 'id'

    add: (product) ->
      @products.push product
      console.log @products

  new Basket()
```

Discussion

By using the fat arrow (=>) instead of the normal arrow (->) the function gets automatically bound to the object and can access the `@-variable`.

CoffeeScript Cookbook

Regular Expressions

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

Replacing substrings

Problem

You need to replace a portion of a string with another value.

Solution

Use the JavaScript `replace` method. `replace` matches with the given string, and returns the edited string.

The first version takes 2 arguments: *pattern* and *string replacement*

```
"JavaScript is my favorite!".replace /Java/, "Coffee"
# => 'CoffeeScript is my favorite!'

"foo bar baz".replace /ba./, "foo"
# => 'foo foo baz'

"foo bar baz".replace /ba./g, "foo"
# => 'foo foo foo'
```

The second version takes 2 arguments: *pattern* and *callback function*

```
"CoffeeScript is my favorite!".replace /(\w+)/g, (match) ->
  match.toUpperCase()
# => 'COFFEESCRIPT IS MY FAVORITE!'
```

The callback function is invoked for each match, and the match value is passed as the argument to the callback.

Discussion

Regular Expressions are a powerful way to match and replace strings.

Replacing HTML tags with HTML named entities

Problem

You need to replace HTML tags with named entities:

```
<br/> => &lt;br/&gt;
```

Solution

```
htmlEncode = (str) ->
  str.replace /[&<>"']/g, ($0) ->
    "&" + {"&":"amp", "<":"lt", ">":"gt", '"':"quot", "'":"#39"}[$0] + ";"

htmlEncode('<a href="http://bn.com">Barnes & Noble</a>')
# => '&lt;a href=&quot;http://bn.com&quot;&gt;Barnes &amp; Noble&lt;/a&gt;'
```

Discussion

There are probably better ways to implement the above method.

CoffeeScript Cookbook

Using Heregexes

Problem

You need to write a complex regular expression.

Solution

Use Coffeescript's "heregexes" – extended regular expressions that ignore internal whitespace and can contain comments.

```
pattern = ///
  ^\(?(\d{3})\)? # Capture area code, ignore optional parens
  [-\s]?(\d{3})  # Capture prefix, ignore optional dash or space
  -?(\d{4})      # Capture line-number, ignore optional dash
///
[area_code, prefix, line] = "(555)123-4567".match(pattern)[1..3]
# => ['555', '123', '4567']
```

Discussion

Breaking up your complex regular expressions and commenting key sections makes them a lot more decipherable and maintainable. For example, changing this regex to allow an optional space between the prefix and line number would now be fairly obvious.

Whitespace characters in heregexes

Whitespace is ignored in heregexes – so what do you do if you need to match a literal ASCII space?

One solution is to use the @\s@ character class, which will match spaces, tabs and line breaks. If you only want to match a space, though, you'll need to use `\x20` to denote a literal ASCII space.

CoffeeScript Cookbook

Searching for substrings

Problem

You need to search for a substring, and return either the starting position of the match or the matching value itself.

Solution

There are several ways to accomplish this using regular expressions. Some methods are called on a `RegExp` pattern or object and some are called on `String` objects.

`RegExp` objects

The first way is to call the `test` method on a `RegExp` pattern or object. The `test` method returns a boolean value:

```
match = /sample/.test("Sample text")
# => false

match = /sample/i.test("Sample text")
# => true
```

The next way to is to call the `exec` method on a `RegExp` pattern or object. The `exec` method returns an array an array with the match information or `null`:

```
match = /s(amp)le/i.exec "Sample text"
# => [ 'Sample', 'amp', index: 0, input: 'Sample text' ]

match = /s(amp)le/.exec "Sample text"
# => null
```

`String` objects

The `match` method matches a given string with the `RegExp`. With 'g' flag returns an array containing the matches, without 'g' flag returns just the first match or if no match is found returns `null`.

```
"Watch out for the rock!".match(/r?or?/g)
# => [ 'o', 'or', 'ro' ]

"Watch out for the rock!".match(/r?or?/)
# => [ 'o', index: 6, input: 'Watch out for the rock!' ]
```

```
"Watch out for the rock!".match(/ror/)
# => null
```

The `search` method matches `RegExp` with string and returns the index of the beginning of the match if found, -1 if not.

```
"Watch out for the rock!".search /for/
# => 10

"Watch out for the rock!".search /rof/
# => -1
```

Discussion

Regular Expressions are a powerful way to test and match substrings.

CoffeeScript Cookbook

Networking

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Bi-Directional Server

Problem

You want to provide a persistent service over a network, one which maintains an on-going connection with a client.

Solution

Create a bi-directional TCP server.

In Node.js

```
net = require 'net'

domain = 'localhost'
port = 9001

server = net.createServer (socket) ->
 console.log "New connection from #{socket.remoteAddress}"

 socket.on 'data', (data) ->
  console.log "#{socket.remoteAddress} sent: #{data}"
  others = server.connections - 1
  socket.write "You have #{others} #{others == 1 and "peer" or "peers"} on this server"

console.log "Listening to #{domain}:#{port}"
server.listen port, domain
```

Example Usage

Accessed by the Bi-Directional Client:

```
$ coffee bi-directional-server.coffee
Listening to localhost:9001
New connection from 127.0.0.1
127.0.0.1 sent: Ping
127.0.0.1 sent: Ping
127.0.0.1 sent: Ping
[...]
```

Discussion

The bulk of the work lies in the @socket.on 'data'@ handler, which processes all of the input from the client. A real server would likely pass the data onto another function to process it and generate any responses so that the original handler.

See also the Bi-Directional Client, Basic Client, and Basic Server recipes.

Exercises

- Add support for choosing the target domain and port based on command-line arguments or on a configuration file.

Bi-Directional Client

Problem

You want to a persistent service over a network, one which maintains an on-going connection with its clients.

Solution

Create a bi-directional TCP client.

In Node.js

```
net = require 'net'

domain = 'localhost'
port = 9001

ping = (socket, delay) ->
 console.log "Pinging server"
 socket.write "Ping"
 nextPing = -> ping(socket, delay)
 setTimeout nextPing, delay

connection = net.createConnection port, domain

connection.on 'connect', () ->
 console.log "Opened connection to #{domain}:#{port}"
 ping connection, 2000

connection.on 'data', (data) ->
 console.log "Received: #{data}"

connection.on 'end', (data) ->
 console.log "Connection closed"
 process.exit()
```

Example Usage

Accessing the Bi-Directional Server:

```
$ coffee bi-directional-client.coffee
Opened connection to localhost:9001
Pinging server
Received: You have 0 peers on this server
```

```
Pinging server
Received: You have 0 peers on this server
Pinging server
Received: You have 1 peer on this server
[...]
Connection closed
```

Discussion

This particular example initiates contact with the server and starts the conversation in the @connection.on ‘connect’@ handler. The bulk of the work in a real client, however, will lie in the @connection.on ‘data’@ handler, which processes output from the server. The @ping@ function only recurses in order to illustrate continuous communication with the server and can be removed from a real client.

See also the Bi-Directional Server, Basic Client, and Basic Server recipes.

Exercises

- Add support for choosing the target domain and port based on command-line arguments or from a configuration file.

Basic Client

Problem

You want to access a service provided over the network.

Solution

Create a basic TCP client.

In Node.js

```
net = require 'net'

domain = 'localhost'
port = 9001

connection = net.createConnection port, domain

connection.on 'connect', () ->
 console.log "Opened connection to #{domain}:#{port}."

connection.on 'data', (data) ->
 console.log "Received: #{data}"
 connection.end()
```

Example Usage

Accessing the Basic Server:

```
$ coffee basic-client.coffee
Opened connection to localhost:9001
Received: Hello, World!
```

Discussion

The most important work takes place in the *connection.on 'data'* handler, where the client receives its response from the server and would most likely arrange for responses to it.

See also the Basic Server, Bi-Directional Client, and Bi-Directional Server recipes.

Exercises

- Add support for choosing the target domain and port based on command-line arguments or from a

configuration file.

Basic HTTP Client

Problem

You want to create a HTTP client.

Solution

In this recipe, we'll use node.js's HTTP library. We'll go from a simple GET request example to a client which returns the external IP of a computer.

GET something

```
http = require 'http'

http.get { host: 'www.google.com' }, (res) ->
    console.log res.statusCode
```

The `get` function, from node.js's `http` module, issues a GET request to a HTTP server. The response comes in the form of a callback, which we can handle in a function. This example merely prints the response status code. Check it out:

```
$ coffee http-client.coffee
200
```

What's my IP?

If you are inside a network which relies on NAT such as a LAN, you probably have faced the issue of finding out what's your external IP address. Let's write a small coffeescript for this.

```
http = require 'http'

http.get { host: 'checkip.dyndns.org' }, (res) ->
    data = ''
    res.on 'data', (chunk) ->
        data += chunk.toString()
    res.on 'end', () ->
        console.log data.match(/([0-9]+\.){3}[0-9]+/)[0]
```

We can get the data from the result object by listening on its `'data'` event; and know that it has come to an end once the `'end'` event has been fired. When that happens, we can do a simple regular expression match to

extract our IP address. Try it:

```
$ coffee http-client.coffee
123.123.123.123
```

Discussion

Note that `http.get` is a shortcut of `http.request`. The latter allows you to issue HTTP requests with different methods, such as POST or PUT.

For API and overall information on this subject, check node.js's http and https documentation pages. Also, the HTTP spec might come in handy.

Exercises

- Create a client for the key-value store HTTP server, from the Basic HTTP Server recipe.

CoffeeScript Cookbook

Basic Server

Problem

You want to provide a service over a network.

Solution

Create a basic TCP server.

In Node.js

```
net = require 'net'

domain = 'localhost'
port = 9001

server = net.createServer (socket) ->
 console.log "Received connection from #{socket.remoteAddress}"
 socket.write "Hello, World!\n"
 socket.end()

console.log "Listening to #{domain}:#{port}"
server.listen port, domain
```

Example Usage

Accessed by the Basic Client:

```
$ coffee basic-server.coffee
Listening to localhost:9001
Received connection from 127.0.0.1
Received connection from 127.0.0.1
[...]
```

Discussion

The function passed to @net.createServer@ receives the new socket provided for each new connection to a client. This basic server simply socializes with its visitors but a hard-working server would pass this socket along to a dedicated handler and then return to the task of waiting for the next client.

See also the Basic Client, Bi-Directional Server, and Bi-Directional Client recipes.

Exercises

- Add support for choosing the target domain and port based on command-line arguments or from a configuration file.

Basic HTTP Server

Problem

You want to create a HTTP server over a network. Over the course of this recipe, we'll go step by step from the smallest server possible to a functional key-value store.

Solution

We'll use node.js's HTTP library to our own selfish purposes and create the simplest web server possible in Coffeescript.

Say 'hi\n'

We can start by importing node.js's HTTP module. This contains `createServer` which, given a simple request handler, returns a HTTP server. We can use that server to listen on a TCP port.

```
http = require 'http'
server = http.createServer (req, res) -> res.end 'hi\n'
server.listen 8000
```

To run this example, simply put in a file and run it. You can kill it with `Ctrl-C`. We can test it using the `curl` command, available on most *nix platforms:

```
$ curl -D - http://localhost:8000/
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

hi
```

What's going on?

Let's get a little bit more feedback on what's happening on our server. While we're at it, we could also be friendlier to our clients and provide them some HTTP headers.

```
http = require 'http'

server = http.createServer (req, res) ->
    console.log req.method, req.url
    data = 'hi\n'
    res.writeHead 200,
        'Content-Type':     'text/plain'
```

```
        'Content-Length':   data.length
    res.end data

server.listen 8000
```

Try to access it once again, but this time use different URL paths, such as `http://localhost:8000/coffee`. You'll see something like this on the server console:

```
$ coffee http-server.coffee
GET /
GET /coffee
GET /user/1337
```

GETting stuff

What if our webserver was able to hold some data? We'll try to come up with a simple key-value store in which elements are retrievable via GET requests. Provide a key on the request path and the server will return the corresponding value — or 404 if it doesn't exist.

```
http = require 'http'

store = # we'll use a simple object as our store
    foo:    'bar'
    coffee: 'script'

server = http.createServer (req, res) ->
    console.log req.method, req.url

    value = store[req.url[1..]]

    if not value
        res.writeHead 404
    else
        res.writeHead 200,
            'Content-Type': 'text/plain'
            'Content-Length': value.length + 1
        res.write value + '\n'

    res.end()

server.listen 8000
```

We can try several URLs to see how it responds:

```
$ curl -D - http://localhost:8000/coffee
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 7
Connection: keep-alive

script

$ curl -D - http://localhost:8000/oops
HTTP/1.1 404 Not Found
Connection: keep-alive
Transfer-Encoding: chunked
```

Use your head(ers)

Let's face it, `text/plain` is kind of lame. How about if we use something hip like `application/json` or `text/xml`? Also, our store retrieval process could use a bit of refactoring — how about some exception throwing & handling? Let's see what we can come up with:

```
http = require 'http'

# known mime types
[any, json, xml] = ['*/*', 'application/json', 'text/xml']

# gets a value from the db in format [value, contentType]
get = (store, key, format) ->
    value = store[key]
    throw 'Unknown key' if not value
    switch format
        when any, json then [JSON.stringify({ key: key, value: value }), json]
        when xml then ["<key>#{ key }</key>\n<value>#{ value }</value>", xml]
        else throw 'Unknown format'

store =
    foo:    'bar'
    coffee: 'script'

server = http.createServer (req, res) ->
    console.log req.method, req.url

    try
        key = req.url[1..]
        [value, contentType] = get store, key, req.headers.accept
        code = 200
    catch error
        contentType = 'text/plain'
        value = error
        code = 404

    res.writeHead code,
```

```
        'Content-Type': contentType
        'Content-Length': value.length + 1
    res.write value + '\n'
    res.end()

server.listen 8000
```

This server will still return the value which matches a given key, or 404 if non-existent. But it will structure the response either in JSON or XML, according to the `Accept` header. See for yourself:

```
$ curl http://localhost:8000/
Unknown key

$ curl http://localhost:8000/coffee
{"key":"coffee","value":"script"}

$ curl -H "Accept: text/xml" http://localhost:8000/coffee
<key>coffee</key>
<value>script</value>

$ curl -H "Accept: image/png" http://localhost:8000/coffee
Unknown format
```

You gotta give to get back

The obvious last step in our adventure is to provide the client the ability to store data. We'll keep our RESTiness by listening to POST requests for this purpose.

```
http = require 'http'

# known mime types
[any, json, xml] = ['*/*', 'application/json', 'text/xml']

# gets a value from the db in format [value, contentType]
get = (store, key, format) ->
    value = store[key]
    throw 'Unknown key' if not value
    switch format
        when any, json then [JSON.stringify({ key: key, value: value }), json]
        when xml then ["<key>#{ key }</key>\n<value>#{ value }</value>", xml]
        else throw 'Unknown format'

# puts a value in the db
put = (store, key, value) ->
    throw 'Invalid key' if not key or key is ''
    store[key] = value
```

```
store =
    foo:    'bar'
    coffee: 'script'

# helper function that responds to the client
respond = (res, code, contentType, data) ->
    res.writeHead code,
        'Content-Type': contentType
        'Content-Length': data.length
    res.write data
    res.end()

server = http.createServer (req, res) ->
    console.log req.method, req.url
    key = req.url[1..]
    contentType = 'text/plain'
    code = 404

    switch req.method
        when 'GET'
            try
                [value, contentType] = get store, key, req.headers.accept
                code = 200
            catch error
                value = error
            respond res, code, contentType, value + '\n'

        when 'POST'
            value = ''
            req.on 'data', (chunk) -> value += chunk
            req.on 'end', () ->
                try
                    put store, key, value
                    value = ''
                    code = 200
                catch error
                    value = error + '\n'
                respond res, code, contentType, value

server.listen 8000
```

Notice how the data is received in a POST request. By attaching some handlers on the `'data'` and `'end'` events of the request object, we're able to buffer and finally save the data from the client in the `store`.

```
$ curl -D - http://localhost:8000/cookie
HTTP/1.1 404 Not Found # ...
Unknown key
```

```
$ curl -D - -d "monster" http://localhost:8000/cookie
HTTP/1.1 200 OK # ...

$ curl -D - http://localhost:8000/cookie
HTTP/1.1 200 OK # ...
{"key":"cookie","value":"monster"}
```

Discussion

Give `http.createServer` a function in the shape of `(request, response) -> ...` and it will return a server object, which we can use to listen on a port. Interact with the `request` and `response` objects to give the server its behaviour. Listen on port 8000 using `server.listen 8000`.

For API and overall information on this subject, check node.js's http and https documentation pages. Also, the HTTP spec might come in handy.

Exercises

- Create a layer in between the server and the developer which would allow the developer to do something like:

```
server = layer.createServer
    'GET /': (req, res) ->
        ...
    'GET /page': (req, res) ->
        ...
    'PUT /image': (req, res) ->
        ...
```

CoffeeScript Cookbook

Design Patterns

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

Builder Pattern

Problem

You need to prepare a complicated, multi-part object, but you expect to do it more than once or with varying configurations.

Solution

Create a Builder to encapsulate the object production process.

The Todo.txt format provides an advanced but still plain-text method for maintaining lists of to-do items. Typing out each item by hand would provide exhausting and error-prone, however, so a TodoTxtBuilder class could save us the trouble:

```
class TodoTxtBuilder
    constructor: (defaultParameters={ }) ->
        @date = new Date(defaultParameters.date) or new Date
        @contexts = defaultParameters.contexts or [ ]
        @projects = defaultParameters.projects or [ ]
        @priority =  defaultParameters.priority or undefined
    newTodo: (description, parameters={ }) ->
        date = (parameters.date and new Date(parameters.date)) or @date
        contexts = @contexts.concat(parameters.contexts or [ ])
        projects = @projects.concat(parameters.projects or [ ])
        priorityLevel = parameters.priority or @priority
        createdAt = [date.getFullYear(), date.getMonth()+1, date.getDate()].join("-")
        contextNames = ("@#{context}" for context in contexts when context).join(" ")
        projectNames = ("+#{project}" for project in projects when project).join(" ")
        priority = if priorityLevel then "(#{priorityLevel})" else ""
        todoParts = [priority, createdAt, description, contextNames, projectNames]
        (part for part in todoParts when part.length > 0).join " "

builder = new TodoTxtBuilder(date: "10/13/2011")

builder.newTodo "Wash laundry"

# => '2011-10-13 Wash laundry'

workBuilder = new TodoTxtBuilder(date: "10/13/2011", contexts: ["work"])

workBuilder.newTodo "Show the new design pattern to Lucy", contexts: ["desk", "xpSession"]

# => '2011-10-13 Show the new design pattern to Lucy @work @desk @xpSession'

workBuilder.newTodo "Remind Sean about the failing unit tests", contexts: ["meeting"], projects: ["compilerRefactor"], priority: 'A'

# => '(A) 2011-10-13 Remind Sean about the failing unit tests @work @meeting +compilerRefactor'
```

Discussion

The TodoTxtBuilder class takes care of all the heavy lifting of text generation and lets the programmer focus on the unique elements of each to-do item. Additionally, a command line tool or GUI could plug into this code and still retain support for later, more advanced versions of the format with ease.

Pre-Construction

Instead of creating a new instance of the needed object from scratch every time, we shift the burden to a separate object that we can then tweak during the object creation process.

```
builder = new TodoTxtBuilder(date: "10/13/2011")

builder.newTodo "Order new netbook"

# => '2011-10-13 Order new netbook'

builder.projects.push "summerVacation"

builder.newTodo "Buy suntan lotion"

# => '2011-10-13 Buy suntan lotion +summerVacation'

builder.contexts.push "phone"

builder.newTodo "Order tickets"

# => '2011-10-13 Order tickets @phone +summerVacation'

delete builder.contexts[0]

builder.newTodo "Fill gas tank"

# => '2011-10-13 Fill gas tank +summerVacation'
```

Exercises

- Expand the project- and context-tag generation code to filter out duplicate entries.
- Some Todo.txt users like to insert project and context tags inside the description of their to-do items. Add code to identify these tags and filter them out of the end tags.

Bridge Pattern

Problem

You need to maintain a reliable interface for code that can change frequently or change between multiple implementations.

Solution

Use the Bridge pattern as an intermediate between the different implementations and the rest of the code.

Assume that you developed an in-browser text editor that saves to the cloud. Now, however, you need to port it to a stand-alone client that saves locally.

```
class TextSaver
 constructor: (@filename, @options) ->
 save: (data) ->

class CloudSaver extends TextSaver
 constructor: (@filename, @options) ->
  super @filename, @options
 save: (data) ->
  # Assuming jQuery
  # Note the fat arrows
  $( =>
   $.post "#{@options.url}/#{@filename}", data, =>
    alert "Saved '#{data}' to #{@filename} at #{@options.url}."
  )

class FileSaver extends TextSaver
 constructor: (@filename, @options) ->
  super @filename, @options
  @fs = require 'fs'
 save: (data) ->
  @fs.writeFile @filename, data, (err) => # Note the fat arrow
   if err? then console.log err
   else console.log "Saved '#{data}' to #{@filename} in #{@options.directory}."

filename = "temp.txt"
data = "Example data"

saver = if window?
 new CloudSaver filename, url: 'http://localhost' # => Saved "Example data" to temp.txt at http://localhost
else if root?
 new FileSaver filename, directory: './' # => Saved "Example data" to temp.txt in ./

saver.save data
```

Discussion

The Bridge pattern helps you to move the implementation-specific code out of sight so that you can focus on your program's specific code. In the above example, the rest of your application can call `saver.save data` without regard for where the file ultimately ends up.

Decorator Pattern

Problem

You have a set of data that you need to process in multiple, possibly varying ways.

Solution

Use the Decorator pattern in order to structure how you apply the changes.

```
miniMarkdown = (line) ->
    if match = line.match /^(#+)\s*(.*)$/
        headerLevel = match[1].length
        headerText = match[2]
        "<h#{headerLevel}>#{headerText}</h#{headerLevel}>"
    else
        if line.length > 0
            "<p>#{line}</p>"
        else
            ''

stripComments = (line) ->
    line.replace /\s*\/\/.*$/, '' # Removes one-line, double-slash C-style comments

TextProcessor = (@processors) ->
    reducer: (existing, processor) ->
        if processor
            processor(existing or '')
        else
            existing
    processLine: (text) ->
        @processors.reduce @reducer, text
    processString: (text) ->
        (@processLine(line) for line in text.split("\n")).join("\n")

exampleText = '''
              # A level 1 header
              A regular line
              // a comment
              ## A level 2 header
              A line // with a comment
              '''

processor = new TextProcessor [stripComments, miniMarkdown]

processor.processString exampleText

# => "<h1>A level 1 header</h1>\n<p>A regular line</p>\n\n<h2>A level 2 header</h2>\n<p>A line</p>"
```

Results

```
<h1>A level 1 header</h1>
<p>A regular line</p>

<h2>A level 1 header</h2>
<p>A line</p>
```

Discussion

The TextProcessor serves the role of Decorator by binding the individual, specialized text processors together. This frees up the miniMarkdown and stripComments components to focus on handling nothing but a single line of text. Future developers only have to write functions that return a string and add it to the array of processors.

We can even modify the existing Decorator object on the fly:

```
smilies =
    ':)' : "smile"
    ':D' : "huge_grin"
    ':(' : "frown"
    ';)' : "wink"

smilieExpander = (line) ->
    if line
        (line = line.replace symbol, "<img src='#{text}.png' alt='#{text}' />") for symbol, text of smilies
    line

processor.processors.unshift smilieExpander

processor.processString "# A header that makes you :) // you may even laugh"

# => "<h1>A header that makes you <img src='smile.png' alt='smile' /></h1>"

processor.processors.shift()

# => "<h1>A header that makes you :)</h1>"
```

Interpreter Pattern

Problem

Someone else needs to run parts of your code in a controlled fashion. Alternately, your language of choice cannot express the problem domain in a concise fashion.

Solution

Use the Interpreter pattern to create a domain-specific language that you translate into specific code.

Assume, for example, that the user wants to perform math inside of your application. You could let them forward code to *eval* but that would let them run arbitrary code. Instead, you can provide a miniature "stack calculator" language that you parse separately in order to only run mathematical operations while reporting more useful error messages.

```
class StackCalculator
 parseString: (string) ->
  @stack = [ ]
  for token in string.split /\s+/
   @parseToken token

  if @stack.length > 1
   throw "Not enough operators: numbers left over"
  else
   @stack[0]

 parseToken: (token, lastNumber) ->
  if isNaN parseFloat(token) # Assume that anything other than a number is an operator
   @parseOperator token
  else
   @stack.push parseFloat(token)

 parseOperator: (operator) ->
  if @stack.length < 2
   throw "Can't operate on a stack without at least 2 items"

  right = @stack.pop()
  left = @stack.pop()

  result = switch operator
   when "+" then left + right
   when "-" then left - right
   when "*" then left * right
   when "/"
    if right is 0
     throw "Can't divide by 0"
    else
     left / right
   else
    throw "Unrecognized operator: #{operator}"

  @stack.push result

calc = new StackCalculator

calc.parseString "5 5 +" # => { result: 10 }
```

```
calc.parseString "4.0 5.5 +" # => { result: 9.5 }

calc.parseString "5 5 + 5 5 + *" # => { result: 100 }

try
 calc.parseString "5 0 /"
catch error
 error # => "Can't divide by 0"

try
 calc.parseString "5 -"
catch error
 error # => "Can't operate on a stack without at least 2 items"

try
 calc.parseString "5 5 5 -"
catch error
 error # => "Not enough operators: numbers left over"

try
 calc.parseString "5 5 5 foo"
catch error
 error # => "Unrecognized operator: foo"
```

Discussion

As an alternative to writing our own interpreter, you can co-op the existing CoffeeScript interpreter in a such a way that its normal syntax makes for more natural (and therefore more comprehensible) expressions of your algorithm.

```
class Sandwich
 constructor: (@customer, @bread='white', @toppings=[], @toasted=false)->

white = (sw) ->
 sw.bread = 'white'
 sw

wheat = (sw) ->
 sw.bread = 'wheat'
 sw

turkey = (sw) ->
 sw.toppings.push 'turkey'
 sw

ham = (sw) ->
 sw.toppings.push 'ham'
 sw

swiss = (sw) ->
 sw.toppings.push 'swiss'
 sw

mayo = (sw) ->
 sw.toppings.push 'mayo'
 sw

toasted = (sw) ->
 sw.toasted = true
 sw

sandwich = (customer) ->
```

```
  new Sandwich customer

to = (customer) ->
 customer

send = (sw) ->
 toastedState = sw.toasted and 'a toasted' or 'an untoasted'

 toppingState = ''
 if sw.toppings.length > 0
  if sw.toppings.length > 1
   toppingState = " with #{sw.toppings[0..sw.toppings.length-2].join ', '} and #{sw.toppings[sw.toppings.length-1]}"
  else
   toppingState = " with #{sw.toppings[0]}"
 "#{sw.customer} requested #{toastedState}, #{sw.bread} bread sandwich#{toppingState}"

send sandwich to 'Charlie' # => "Charlie requested an untoasted, white bread sandwich"
send turkey sandwich to 'Judy' # => "Judy requested an untoasted, white bread sandwich with turkey"
send toasted ham turkey sandwich to 'Rachel' # => "Rachel requested a toasted, white bread sandwich with turkey and ham"
send toasted turkey ham swiss sandwich to 'Matt' # => "Matt requested a toasted, white bread sandwich with swiss, ham and turkey"
```

This example allows for layers of functions by how it returns the modified object so that outer functions can modify it in turn. By borrowing a very and the particle *to*, the example lends natural grammar to the construction and ends up reading like an actual sentence when used correctly. This way, both your CoffeeScript skills and your existing language skills can help catch code problems.

Command Pattern

Problem

You need to let another object handle when your private code is executed.

Solution

Use the Command pattern to pass along references to your functions.

```
# Using a private variable to simulate external scripts or modules
incrementers = (() ->
 privateVar = 0

 singleIncrementer = () ->
  privateVar += 1

 doubleIncrementer = () ->
  privateVar += 2

 commands =
  single: singleIncrementer
  double: doubleIncrementer
  value: -> privateVar
)()

class RunsAll
 constructor: (@commands...) ->
 run: -> command() for command in @commands

runner = new RunsAll(incrementers.single, incrementers.double, incrementers.single, incrementers.double)
runner.run()
incrementers.value() # => 6
```

Discussion

With functions as first-class objects and with the function-bound variable scope inherited from Javascript, the CoffeeScript language makes the pattern nearly invisible. In fact, any function passed along as callbacks can act as a *Command*.

The `jqXHR` object returned by jQuery AJAX methods uses this pattern.

```
jqxhr = $.ajax
 url: "/"
```

```coffeescript
logMessages = ""

jqxhr.success -> logMessages += "Success!\n"
jqxhr.error -> logMessages += "Error!\n"
jqxhr.complete -> logMessages += "Completed!\n"

# On a valid AJAX request:
# logMessages == "Success!\nCompleted!\n"
```

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Factory Method Pattern

Problem

You don't know what kind of object you will need until runtime.

Solution

Use the Factory Method pattern and choose the object to be generated dynamically.

Say that you need to load a file into an editor but you don't know its format until the user chooses the file. A class using the Factory Method pattern can serve up different parsers depending on the file's extension.

```
class HTMLParser
 constructor: ->
  @type = "HTML parser"
class MarkdownParser
 constructor: ->
  @type = "Markdown parser"
class JSONParser
 constructor: ->
  @type = "JSON parser"

class ParserFactory
 makeParser: (filename) ->
  matches = filename.match /\.(\w*)$/
  extension = matches[1]
  switch extension
   when "html" then new HTMLParser
   when "htm" then new HTMLParser
   when "markdown" then new MarkdownParser
   when "md" then new MarkdownParser
   when "json" then new JSONParser

factory = new ParserFactory

factory.makeParser("example.html").type # => "HTML parser"

factory.makeParser("example.md").type # => "Markdown parser"

factory.makeParser("example.json").type # => "JSON parser"
```

Discussion

In the example, you can ignore the specifics of the file's format and focus on the parsed content. A more advanced Factory Method might, for instance, also search for versioning data within the file itself before returning a more precise parser (e.g. an HTML5 parser instead of an HTML v4 parser).

CoffeeScript Cookbook

Strategy Pattern

Problem

You have more than one way to solve a problem but you need to choose (or even switch) between these methods at run time.

Solution

Encapsulate your algorithms inside of Strategy objects.

Given an unsorted list, for example, we can change the sorting algorithm under different circumstances.

The base class:

```
StringSorter = (@algorithm) ->
    sort: (list) -> @algorithm list
```

The strategies:

```
bubbleSort = (list) ->
    anySwaps = false
    swapPass = ->
        for r in [0..list.length-1]
            if list[r] > list[r+1]
                anySwaps = true
                [list[r], list[r+1]] = [list[r+1], list[r]]

    swapPass()
    while anySwaps
        anySwaps = false
        swapPass()
    list

reverseBubbleSort = (list) ->
    anySwaps = false
    swapPass = ->
        for r in [list.length-1..1]
            if list[r] < list[r-1]
                anySwaps = true
                [list[r], list[r-1]] = [list[r-1], list[r]]

    swapPass()
```

```
    while anySwaps
        anySwaps = false
        swapPass()
    list
```

Using the strategies:

```
sorter = new StringSorter bubbleSort

unsortedList = ['e', 'b', 'd', 'c', 'x', 'a']

sorter.sort unsortedList

# => ['a', 'b', 'c', 'd', 'e', 'x']

unsortedList.push 'w'

# => ['a', 'b', 'c', 'd', 'e', 'x', 'w']

sorter.algorithm = reverseBubbleSort

sorter.sort unsortedList

# => ['a', 'b', 'c', 'd', 'e', 'w', 'x']
```

Discussion

"No plan survives first contact with the enemy", nor users, but we can use the knowledge gained from changing circumstances to adapt. Near the end of the example, for instance, the newest item in the array now lies out of order. Knowing that detail, we can then speed the sort up by switching to an algorithm optimized for that exact scenario with nothing but a simple reassignment.

Exercises

- Expand `StringSorter` into an `AlwaysSortedArray` class that implements all of the functionality of a regular array but which automatically sorts new items based on the method of insertion (e.g. `push` vs. `shift`).

CoffeeScript Cookbook

Memento Pattern

Problem

You want to anticipate the reversion of changes to an object.

Solution

Use the Memento pattern to track changes to an object. The class using the pattern will export a `memento` object stored elsewhere.

If you have application where the user can edit a text file, for example, they may want to undo their last action. You can save the current state of the file before the user changes it and then roll back to that at a later point.

```
class PreserveableText
 class Memento
  constructor: (@text) ->

 constructor: (@text) ->
 save: (newText) ->
  memento = new Memento @text
  @text = newText
  memento
 restore: (memento) ->
  @text = memento.text

pt = new PreserveableText "The original string"
pt.text # => "The original string"

memento = pt.save "A new string"
pt.text # => "A new string"

pt.save "Yet another string"
pt.text # => "Yet another string"

pt.restore memento
pt.text # => "The original string"
```

Discussion

The Memento object returned by `PreserveableText#save` stores the important state information separately for safe-keeping. You could even serialize this Memento in order to maintain an “undo” buffer on the hard disk or remotely for such data-intensive objects as edited images.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!

CoffeeScript Cookbook

Singleton Pattern

Problem

Many times you only want one, and only one, instance of a class. For example, you may only need one class that creates server resources and you want to ensure that the one object can control those resources. Beware, however, because the singleton pattern can be easily abused to mimic unwanted global variables.

Solution

The publicly available class only contains the method to get the one true instance. The instance is kept within the closure of that public object and is always returned.

The actual definition of the singleton class follows.

Note that I am using the idiomatic module export feature to emphasize the publicly accessible portion of the module. Remember coffeescript wraps all files in a function block to protect the global namespace.

```
root = exports ? this # http://stackoverflow.com/questions/4214731/coffeescript-global-variables

# The publicly accessible Singleton fetcher
class root.Singleton
  _instance = undefined # Must be declared here to force the closure on the class
  @get: (args) -> # Must be a static method
    _instance ?= new _Singleton args

# The actual Singleton class
class _Singleton
  constructor: (@args) ->

  echo: ->
    @args

a = root.Singleton.get 'Hello A'
a.echo()
# => 'Hello A'

b = root.Singleton.get 'Hello B'
a.echo()
# => 'Hello A'

b.echo()
# => 'Hello A'

root.Singleton._instance
```

```
root.Singleton._instance
# => undefined

root.Singleton._instance = 'foo'

root.Singleton._instance
# => 'foo'

c = root.Singleton.get 'Hello C'
c.foo()
# => 'Hello A'

a.foo()
# => 'Hello A'
```

Discussion

See in the above example how all instances are outputting from the same instance of the Singleton class.

Note how incredibly simple coffeescript makes this design pattern. For reference and discussion on nice javascript implementations, check out Essential JavaScript Design Patterns For Beginners.

CoffeeScript Cookbook

Databases

Don't see the recipe you want? Add it yourself by reading the Contributor's Guide, or request it by adding it to Wanted Recipes.

CoffeeScript Cookbook

MongoDB

Problem

You need to interface with a MongoDB database.

Solution

For Node.js

Setup

- Install MongoDB on your computer if you have not already.
- Install the native MongoDB module.

Saving Records

```
mongo = require 'mongodb'

server = new mongo.Server "127.0.0.1", 27017, {}

client = new mongo.Db 'test', server

# save() updates existing records or inserts new ones as needed
exampleSave = (dbErr, collection) ->
 console.log "Unable to access database: #{dbErr}" if dbErr
 collection.save { _id: "my_favorite_latte", flavor: "honeysuckle" }, (err, docs) ->
  console.log "Unable to save record: #{err}" if err
  client.close()

client.open (err, database) ->
 client.collection 'coffeescript_example', exampleSave
```

Finding Records

```
mongo = require 'mongodb'

server = new mongo.Server "127.0.0.1", 27017, {}

client = new mongo.Db 'test', server

exampleFind = (dbErr, collection) ->
 console.log "Unable to access database: #{dbErr}" if dbErr
 collection.find({ _id: "my_favorite_latte" }).nextObject (err, result) ->
```

```
  if err
   console.log "Unable to find record: #{err}"
  else
   console.log result # => {  id: "my_favorite_latte", flavor: "honeysuckle" }
  client.close()

client.open (err, database) ->
 client.collection 'coffeescript_example', exampleFind
```

For Browsers

A REST-based interface is in the works. This will provide AJAX-based access.

Discussion

This recipe breaks the *save* and *find* into separate examples in order to separate the MongoDB-specific concerns from the task of connection and callback management. The async module can help with that.

CoffeeScript Cookbook

SQLite

Problem

You need to interface with a SQLite database from inside of Node.js.

Solution

Use the SQLite module.

```
sqlite = require 'sqlite'

db = new sqlite.Database

# The module uses asynchronous methods,
# so we chain the calls the db.execute
exampleCreate = ->
 db.execute "CREATE TABLE snacks (name TEXT(25), flavor TEXT(25))",
  (exeErr, rows) ->
   throw exeErr if exeErr
   exampleInsert()

exampleInsert = ->
 db.execute "INSERT INTO snacks (name, flavor) VALUES ($name, $flavor)",
  { $name: "Potato Chips", $flavor: "BBQ" },
  (exeErr, rows) ->
   throw exeErr if exeErr
   exampleSelect()

exampleSelect = ->
 db.execute "SELECT name, flavor FROM snacks",
  (exeErr, rows) ->
   throw exeErr if exeErr
   console.log rows[0] # => { name: 'Potato Chips', flavor: 'BBQ' }

# :memory: creates a DB in RAM
# You can supply a filepath (like './example.sqlite') to create/open one on disk
db.open ":memory:", (openErr) ->
 throw openErr if openErr
 exampleCreate()
```

Discussion

You can also prepare your SQL queries beforehand:

You can also prepare your SQL queries beforehand:

```
sqlite = require 'sqlite'
async = require 'async' # Not required but added to make the example more concise

db = new sqlite.Database

createSQL = "CREATE TABLE drinks (name TEXT(25), price NUM)"

insertSQL = "INSERT INTO drinks (name, price) VALUES (?, ?)"

selectSQL = "SELECT name, price FROM drinks WHERE price < ?"

create = (onFinish) ->
 db.execute createSQL, (exeErr) ->
  throw exeErr if exeErr
  onFinish()

prepareInsert = (name, price, onFinish) ->
 db.prepare insertSQL, (prepErr, statement) ->
  statement.bindArray [name, price], (bindErr) ->
   statement.fetchAll (fetchErr, rows) -> # Called so that it executes the insert
    onFinish()

prepareSelect = (onFinish) ->
 db.prepare selectSQL, (prepErr, statement) ->
  statement.bindArray [1.00], (bindErr) ->
   statement.fetchAll (fetchErr, rows) ->
    console.log rows[0] # => { name: "Mia's Root Beer", price: 0.75 }
    onFinish()

db.open ":memory:", (openErr) ->
 async.series([
  (onFinish) -> create onFinish,
  (onFinish) -> prepareInsert "LunaSqueeze", 7.95, onFinish,
  (onFinish) -> prepareInsert "Viking Sparkling Grog", 4.00, onFinish,
  (onFinish) -> prepareInsert "Mia's Root Beer", 0.75, onFinish,
  (onFinish) -> prepareSelect onFinish
 ])
```

The SQLite version of SQL and the node-sqlite module documentation provide more complete information.

Is this recipe wrong, incomplete, or non idiomatic? Help fix it by reading the Contributor's Guide!