



Ruby: Collections

Juan “Harek” Urrios
@xharekx33

What are collections?

Often we need to store variables that hold more than a single value.

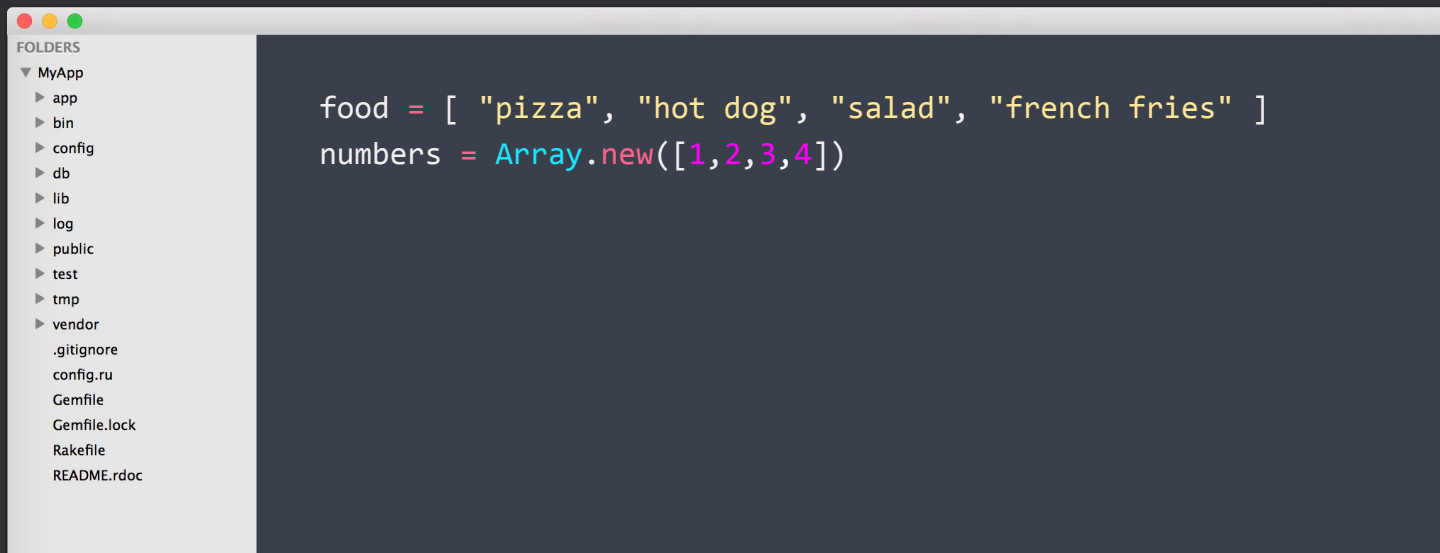
This lists of values are called Collections and allow for the organizing of large amounts of data.

The most common types are: **Arrays** and **Hashes**



Arrays

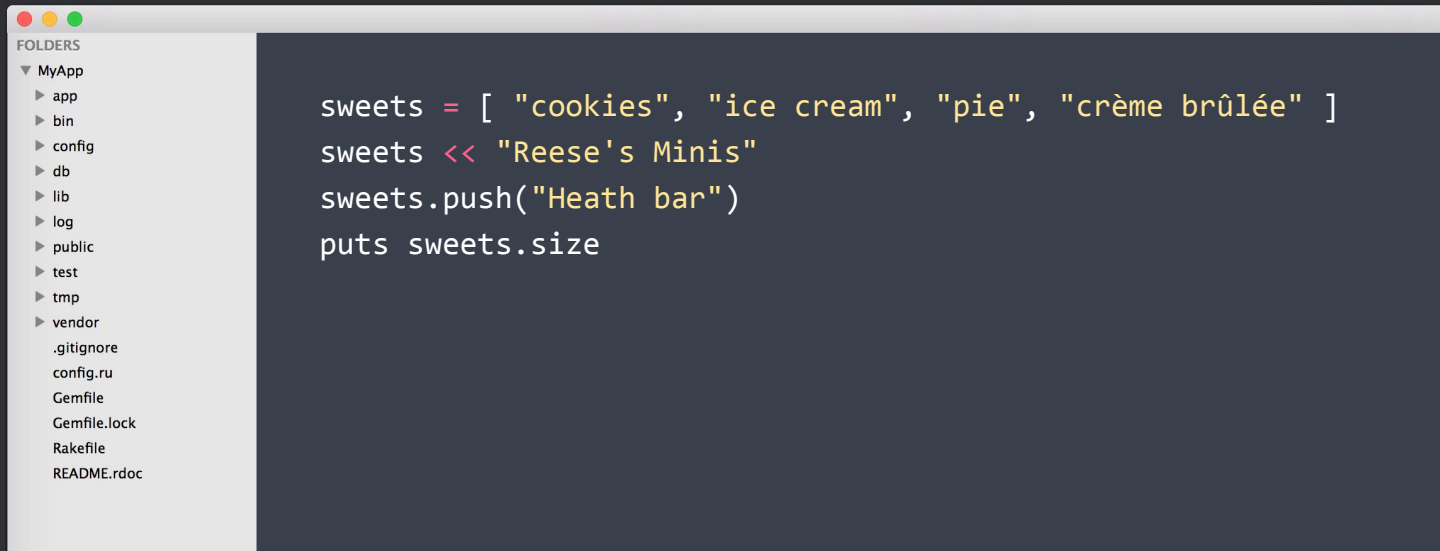
Arrays can be used to store many different kinds of data including strings, numbers, and almost any other kind of Ruby object.



@xharekx33

Arrays: Adding items

Arrays are dynamic, so it isn't necessary to preallocate space for them and you can add items after the fact with the **push** method or the **<<** operator.



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor displays Ruby code that demonstrates adding items to an array.

```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée" ]  
sweets << "Reese's Minis"  
sweets.push("Heath bar")  
puts sweets.size
```



@xharekx33

Arrays: Accessing items

You can access elements in the array directly by their assigned number, also known as the **index**. The index starts at 0, but you can also start from the end of the array using negative numbers!

```
$ irb
2.2-head :001 > food = [ "pizza", "hot dog", "salad", "french
fries" ]
=> ["pizza", "hot dog", "salad", "french fries"]
2.2-head :002 > food[0]
=> "pizza"
2.2-head :003 > food[3]
=> "french fries"
2.2-head :003 > food[-1]
=> "french fries"
```



Arrays: Accessing items

Be careful, nothing stops you from accessing an index that doesn't exist!

```
$ irb
2.2-head :001 > food = [ "pizza", "hot dog", "salad", "french
fries" ]
=> ["pizza", "hot dog", "salad", "french fries"]
2.2-head :002 > food[900]
=> nil
```



Arrays: stacks and queues

You can use **pop** to get the last element from the array and remove it, using the Array as a stack. Or you can use **shift** to get the first element from the array and remove it, using the Array as a queue

```
$ irb
2.2-head :001 > food = [ "pizza", "hot dog", "salad", "french
fries" ]
=> ["pizza", "hot dog", "salad", "french fries"]
2.2-head :002 > food.pop
=> "french fries"
2.2-head :003 > food.shift
=> "pizza"
2.2-head :004 > food
=> ["hot dog", "salad"]
```



Arrays: Deleting specific items

If you want to delete an item with a certain index instead of using `pop` or `shift`, you can use `delete_at` and specify an index

```
$ irb
2.2-head :001 > food = [ "pizza", "hot dog", "salad", "french
fries" ]
=> ["pizza", "hot dog", "salad", "french fries"]
2.2-head :002 > food.delete_at(1)
=> "hot dog"
2.2-head :003 > food.delete_at(-2)
=> "salad"
2.2-head :004 > food
=> ["pizza", "french fries"]
```



Arrays: Finding items

If you want to find out whether an item is in an array or not you can use `include?` or you can use `index` to get the index of the first instance of that value

```
$ irb
2.2-head :001 > food = [ "pizza", "hot dog", "salad", "french
fries" ]
=> ["pizza", "hot dog", "salad", "french fries"]
2.2-head :002 > food.include?("salad")
=> true
2.2-head :003 > food.index("hot dog")
=> 1
2.2-head :003 > food.index("burger")
=> nil
```



@xharekx33

Arrays: Multidimensional

Arrays items do not need to be all of the same data type. They can be heterogeneous. This is rarely useful though.

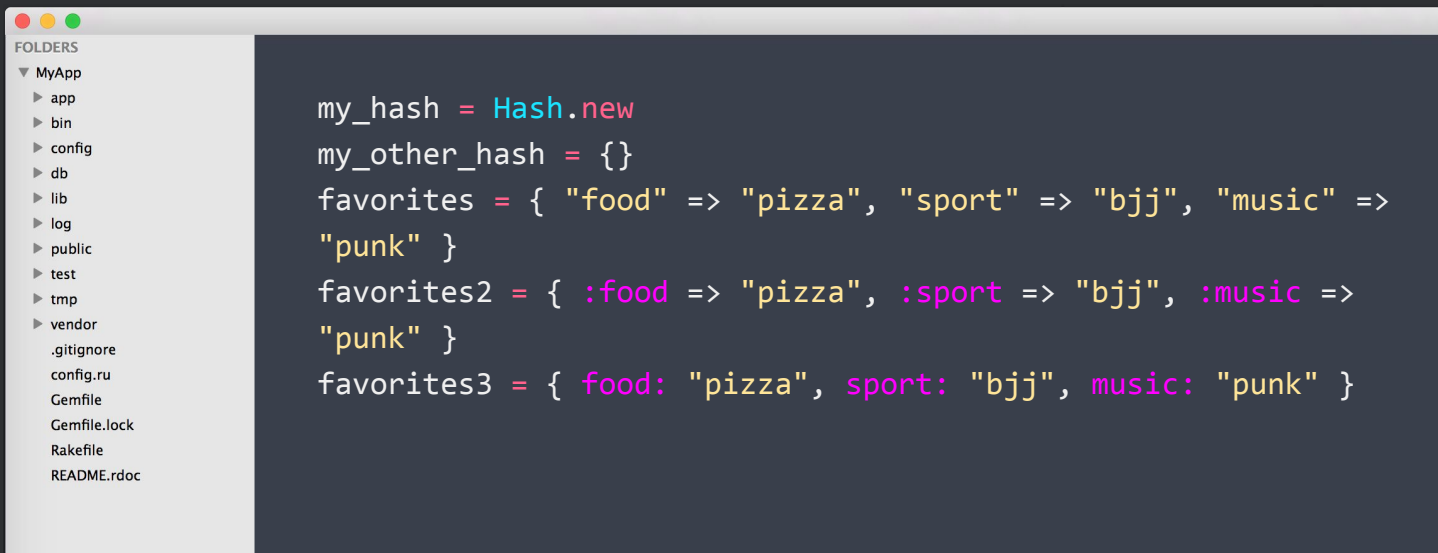
What can actually be quite useful is to store an Array inside an Array!

```
$ irb
2.2-head :001 > random = [ "yolo", 45, "weird", 7.0, nil ]
=> [ "yolo", 45, "weird", 7.0, nil ]
2.2-head :002 > my_array1 = Array.new(5)
=> => [nil, nil, nil, nil, nil]
2.2-head :003 > matrix = Array.new(5, Array.new(2))
=> [[nil, nil], [nil, nil], [nil, nil], [nil, nil], [nil,
nil]]
```



Hashes

Hashes are also containers for data, like arrays, but instead of storing data based on numeric indices, you use "keys" which can be strings or symbols

A screenshot of a code editor window. On the left is a sidebar with a 'FOLDERS' section containing a tree view of a project named 'MyApp'. The tree includes folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area has a dark background and displays four lines of Ruby code. The code defines a new hash, an empty hash, and two hashes using different syntaxes: one with string keys and one with symbol keys.

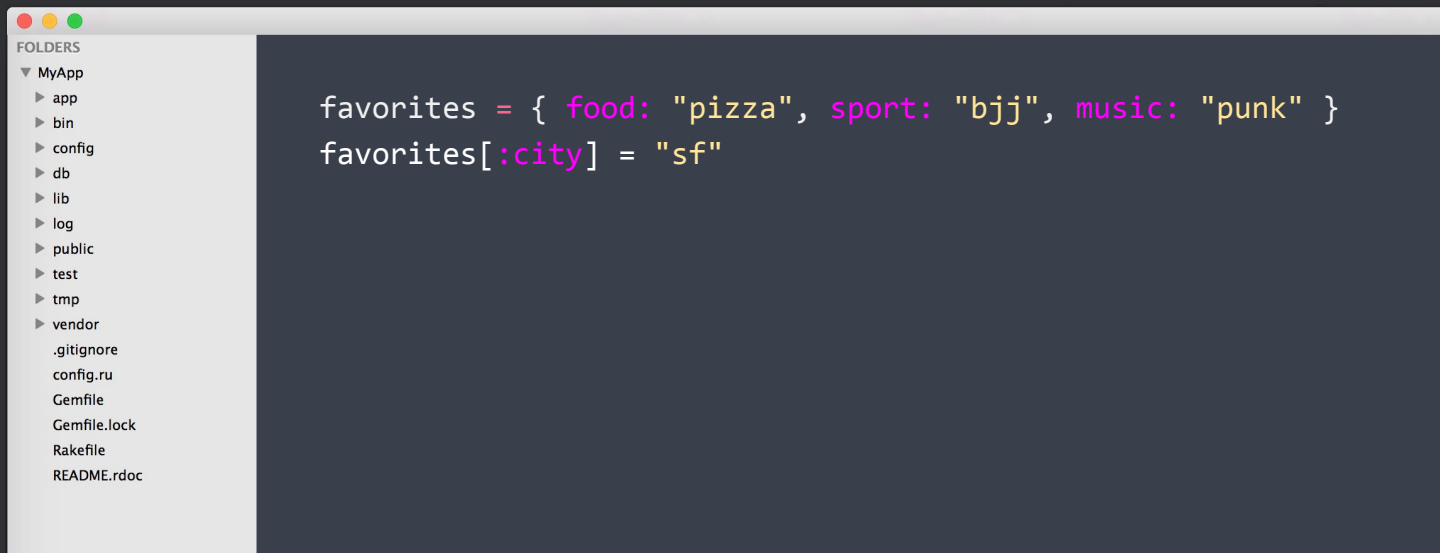
```
my_hash = Hash.new
my_other_hash = {}
favorites = { "food" => "pizza", "sport" => "bjj", "music" => "punk" }
favorites2 = { :food => "pizza", :sport => "bjj", :music => "punk" }
favorites3 = { food: "pizza", sport: "bjj", music: "punk" }
```



@xharekx33

Hashes: Adding items

To add items to a hash you simply define a new key and assign a value.



@xharekx33

Hashes: Accessing items

You can access elements in a hash directly by their key. Don't try to use indexes as you would do with arrays.

```
$ irb
2.2-head :001 > favorites = { food: "pizza", sport: "bjj",
music: "punk" }
=> { food: "pizza", sport: "bjj", music: "punk" }
2.2-head :002 > favorites[:food]
=> "pizza"
2.2-head :003 > favorites[0]
=> nil
```



Hashes: Deleting specific items

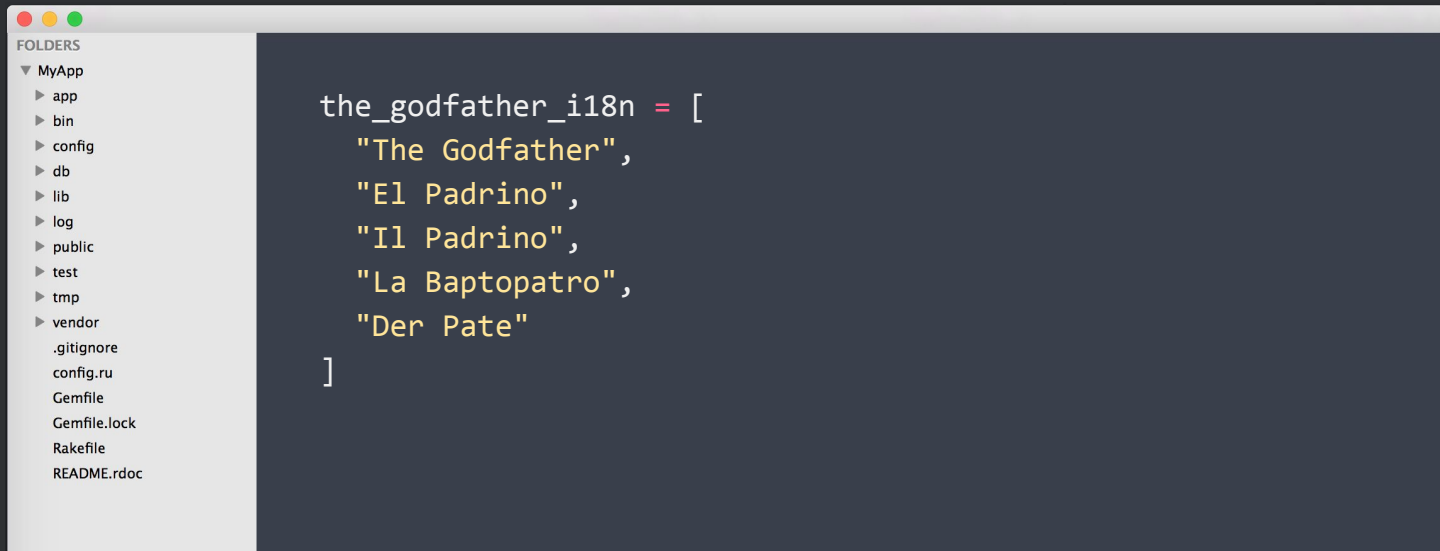
If you want to delete an item from the Hash simply use the **delete** method and specify the key that you want to delete from the hash

```
$ irb
2.2-head :001 > favorites = { food: "pizza", sport: "bjj",
music: "punk" }
=> { food: "pizza", sport: "bjj", music: "punk" }
2.2-head :002 > favorites.delete(:food)
=> "pizza"
2.2-head :003 > favorites
=> { sport: "bjj", music: "punk" }
```



Why use hashes?

What if you're storing lists with no inherent order, but instead need some other property to make sense of the values?



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor displays a Ruby hash definition. The hash is named 'the_godfather_i18n' and contains five string values: 'The Godfather', 'El Padrino', 'Il Padrino', 'La Baptopatro', and 'Der Pate'.

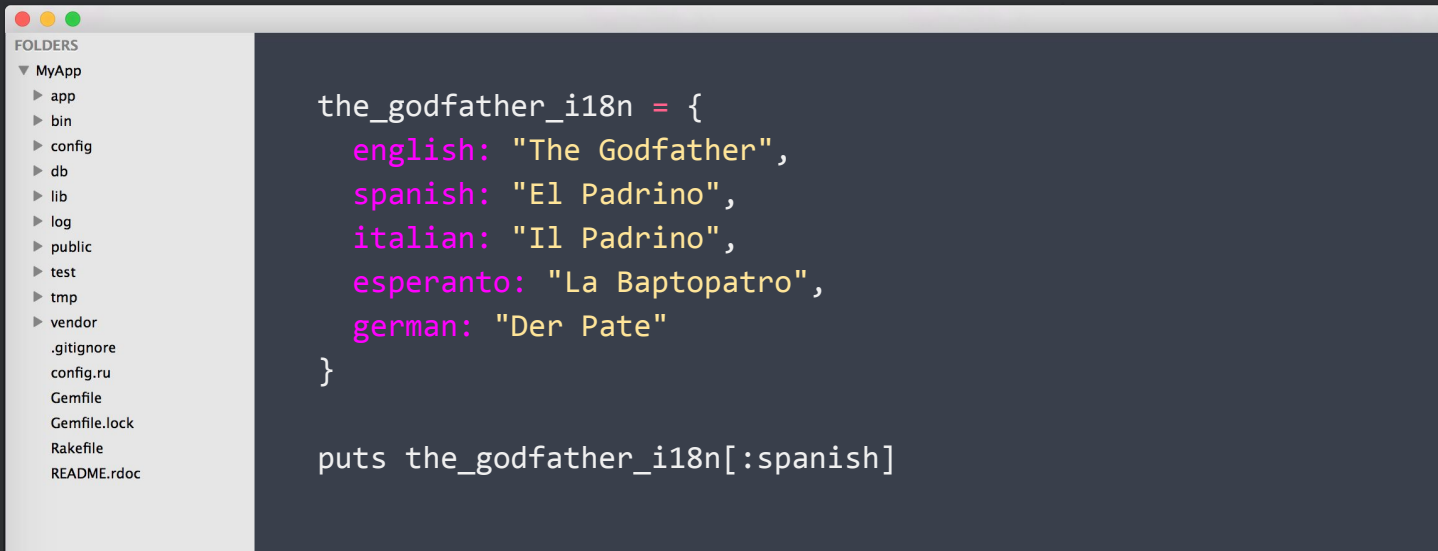
```
the_godfather_i18n = [  
  "The Godfather",  
  "El Padrino",  
  "Il Padrino",  
  "La Baptopatro",  
  "Der Pate"  
]
```



@xharekx33

Why use hashes?

To know what translation matches what language using a hash is more convenient and intuitive than having to remember what numeric index corresponds to each language.

A screenshot of a code editor window. On the left is a sidebar with a 'FOLDERS' section containing a tree view of a project named 'MyApp'. The tree includes folders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area displays Ruby code. It defines a hash named 'the_godfather_i18n' with keys for 'english', 'spanish', 'italian', 'esperanto', and 'german', each mapped to a specific translation. Below the hash definition, there is a line of code that puts the Spanish translation into the console.

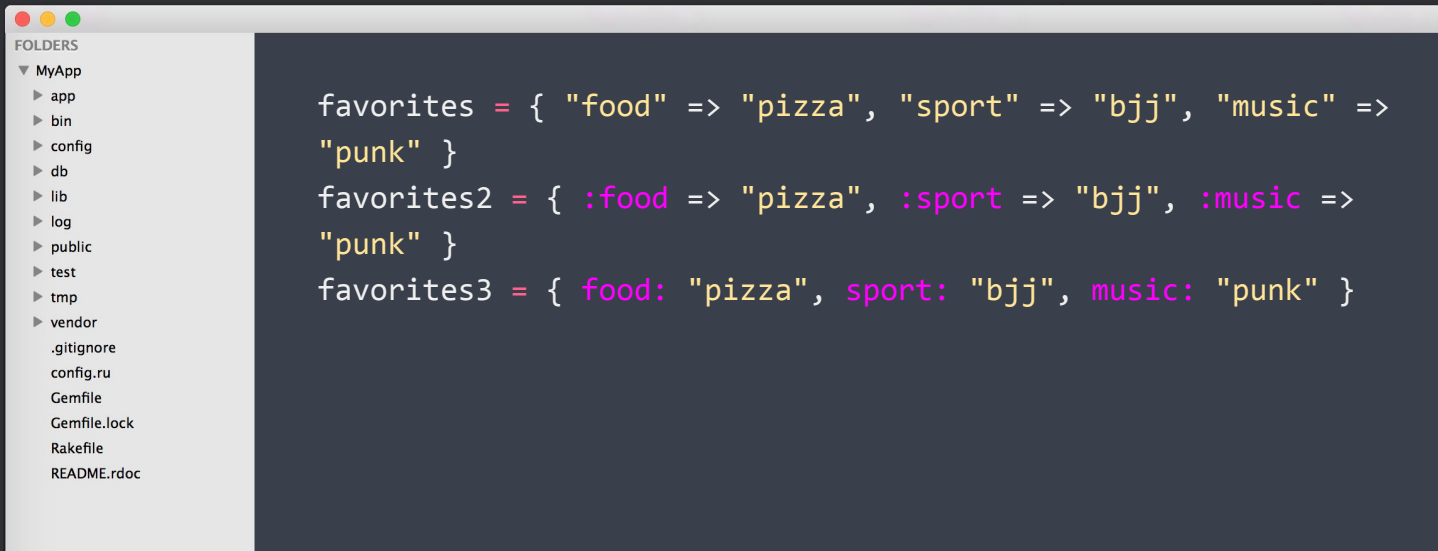
```
the_godfather_i18n = {  
  english: "The Godfather",  
  spanish: "El Padrino",  
  italian: "Il Padrino",  
  esperanto: "La Baptopatro",  
  german: "Der Pate"  
}  
  
puts the_godfather_i18n[:spanish]
```



@xharekx33

Symbols

There's no quick and easy definition for a symbol (here are 13 ways of looking at them: <http://bit.ly/1LkJjbb>) For now, let's just say that they are immutable identifiers, usually used as hash keys instead of strings.



```
FOLDERS
▼ MyApp
  ► app
  ► bin
  ► config
  ► db
  ► lib
  ► log
  ► public
  ► test
  ► tmp
  ► vendor
.gitignore
config.ru
Gemfile
Gemfile.lock
Rakefile
README.rdoc

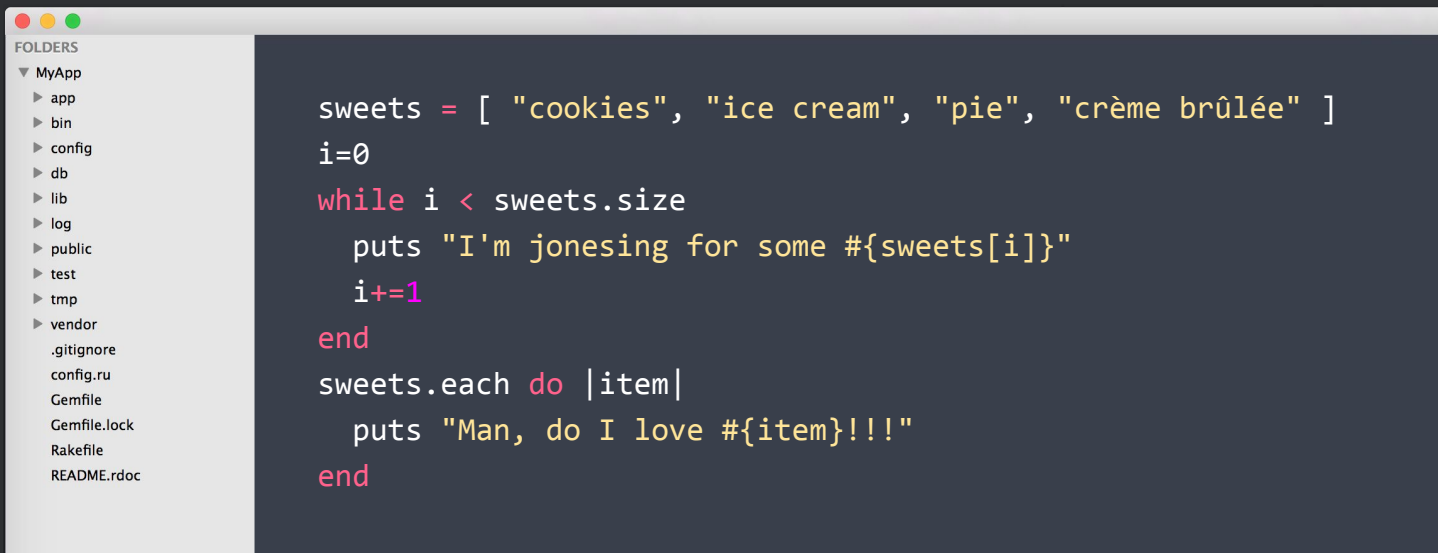
favorites = { "food" => "pizza", "sport" => "bjj", "music" =>
  "punk" }
favorites2 = { :food => "pizza", :sport => "bjj", :music =>
  "punk" }
favorites3 = { food: "pizza", sport: "bjj", music: "punk" }
```



@xharekx33

Iteration: each

To go over all the elements in an Array you could build a while loop... but it's cleaner and easier to use the **each** method, that will let you run some code for each item in the array.



The image shows a screenshot of a code editor window. On the left, there is a sidebar with a file explorer titled 'FOLDERS'. It shows a project named 'MyApp' with several subfolders: 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor'. Below these are several files: '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main area of the editor displays Ruby code. The code defines an array 'sweets' containing 'cookies', 'ice cream', 'pie', and 'crème brûlée'. It then uses a 'while' loop to iterate over the array by index, printing a message for each item. Finally, it uses the 'each' method to iterate over the array directly, printing a message for each item. The code is as follows:

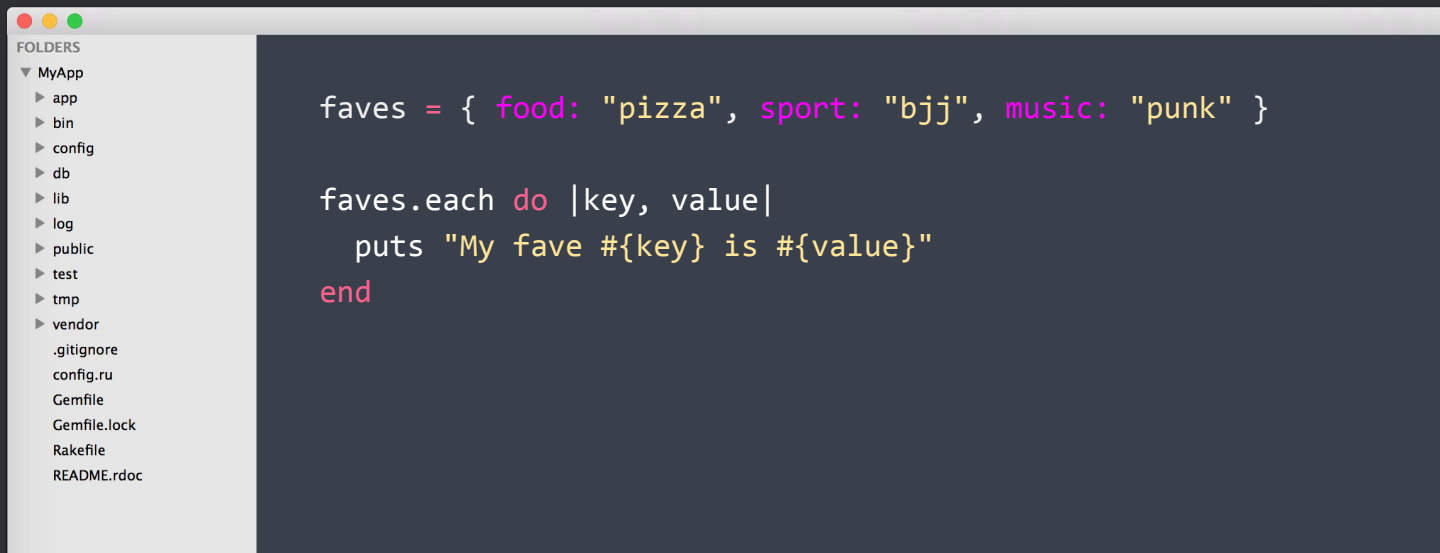
```
sweets = [ "cookies", "ice cream", "pie", "crème brûlée" ]
i=0
while i < sweets.size
  puts "I'm jonesing for some #{sweets[i]}"
  i+=1
end
sweets.each do |item|
  puts "Man, do I love #{item}!!!"
end
```



@xharekx33

Iteration: each

You can also use **each** with Hashes and get the key, value pair. That's way more convenient than coming up with a regular loop that does the same thing!



The image shows a screenshot of a code editor window. On the left, there is a sidebar titled 'FOLDERS' showing a directory structure for 'MyApp'. The main area of the editor displays Ruby code that defines a hash 'faves' and iterates over it using the 'each' method. The code is as follows:

```
faves = { food: "pizza", sport: "bjj", music: "punk" }

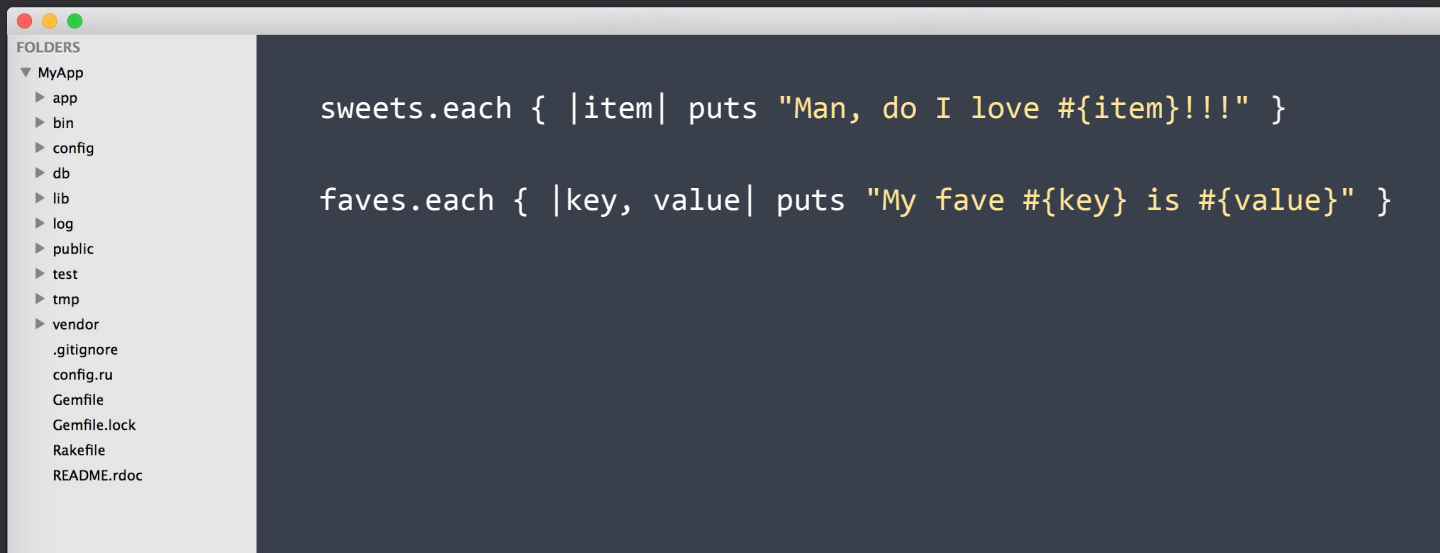
faves.each do |key, value|
  puts "My fave #{key} is #{value}"
end
```



@xharekx33

Iteration: each in a single line

Our previous calls to **each** can be written in a single line.



@xharekx33

Exercise: each

Given an array with all the names of the people in class, tell them "good morning"



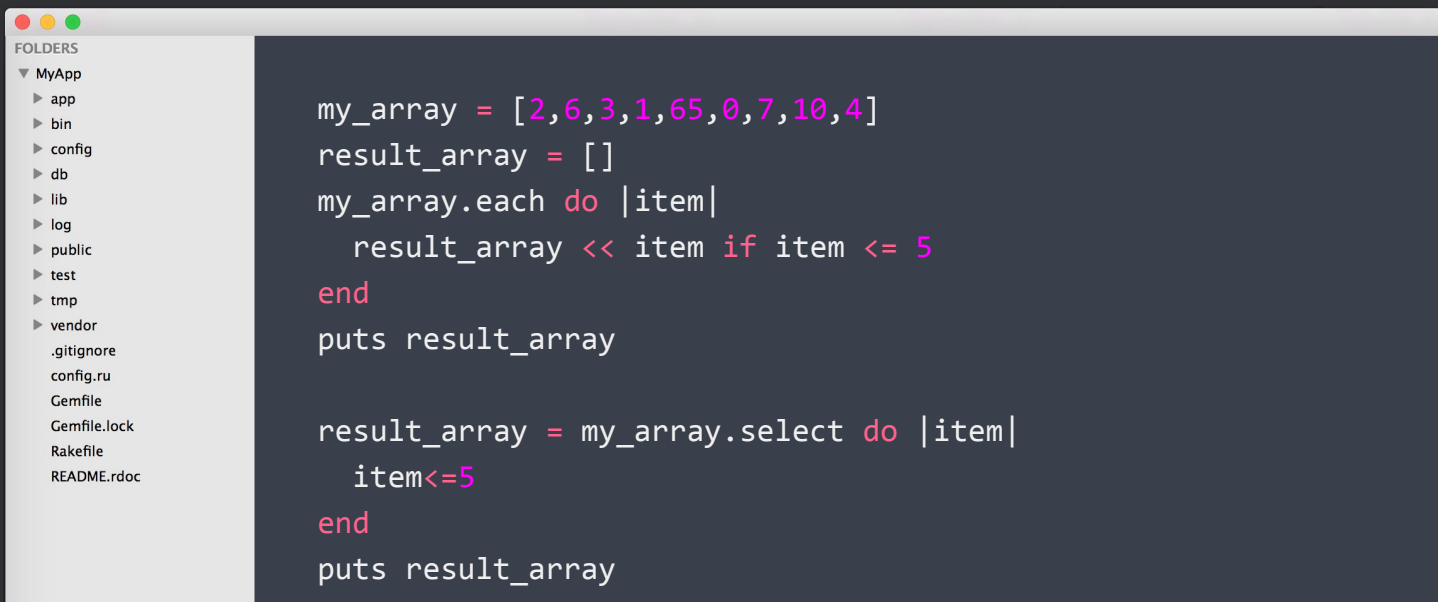
Transforming collections

Next we will see other methods we can use to iterate over collections and transform them and how all these methods have **each** at the heart of their functionality.



Iteration: select

With `select` you can extract only the values you want from an Array and get a new array. The original is left untouched.



The image shows a screenshot of a code editor with a dark theme. On the left, there is a sidebar titled 'FOLDERS' showing a project structure for 'MyApp' with subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', along with files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains two snippets of Ruby code. The first snippet uses a loop to build a new array from an existing one. The second snippet uses the `select` method to filter an array based on a condition.

```
my_array = [2,6,3,1,65,0,7,10,4]
result_array = []
my_array.each do |item|
  result_array << item if item <= 5
end
puts result_array

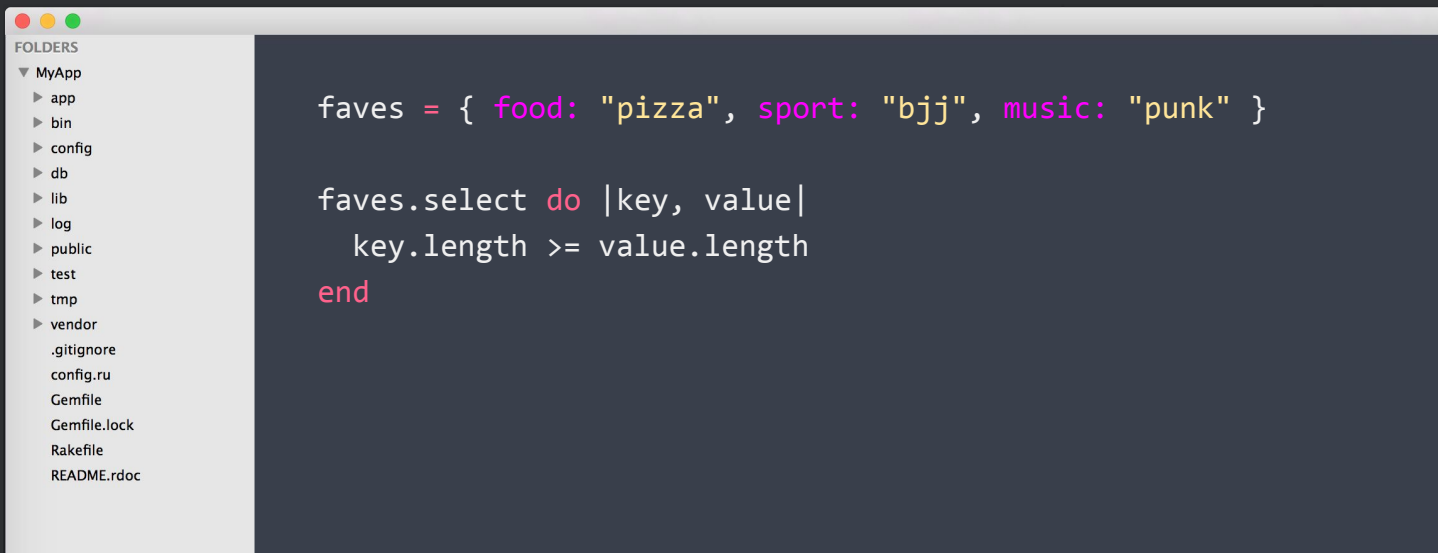
result_array = my_array.select do |item|
  item<=5
end
puts result_array
```



@xharekx33

Iteration: select

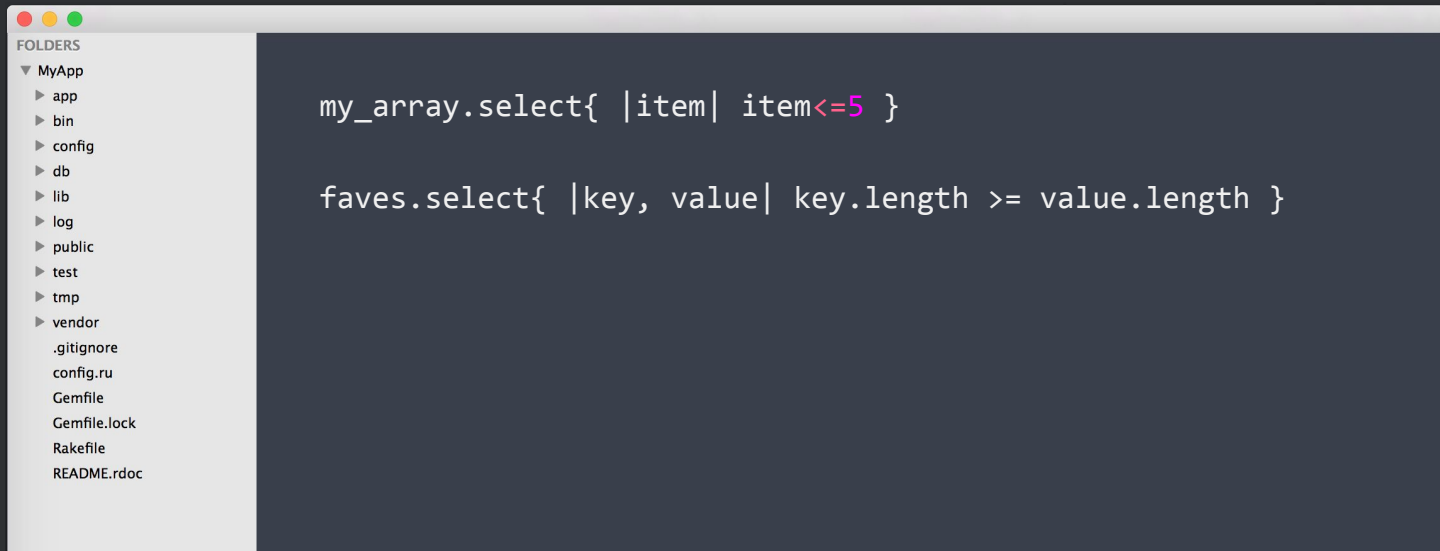
Select can be used with hashes too



@xharekx33

Iteration: select in a single line

Our previous `select` calls can be also written in a single line.



@xharekx33

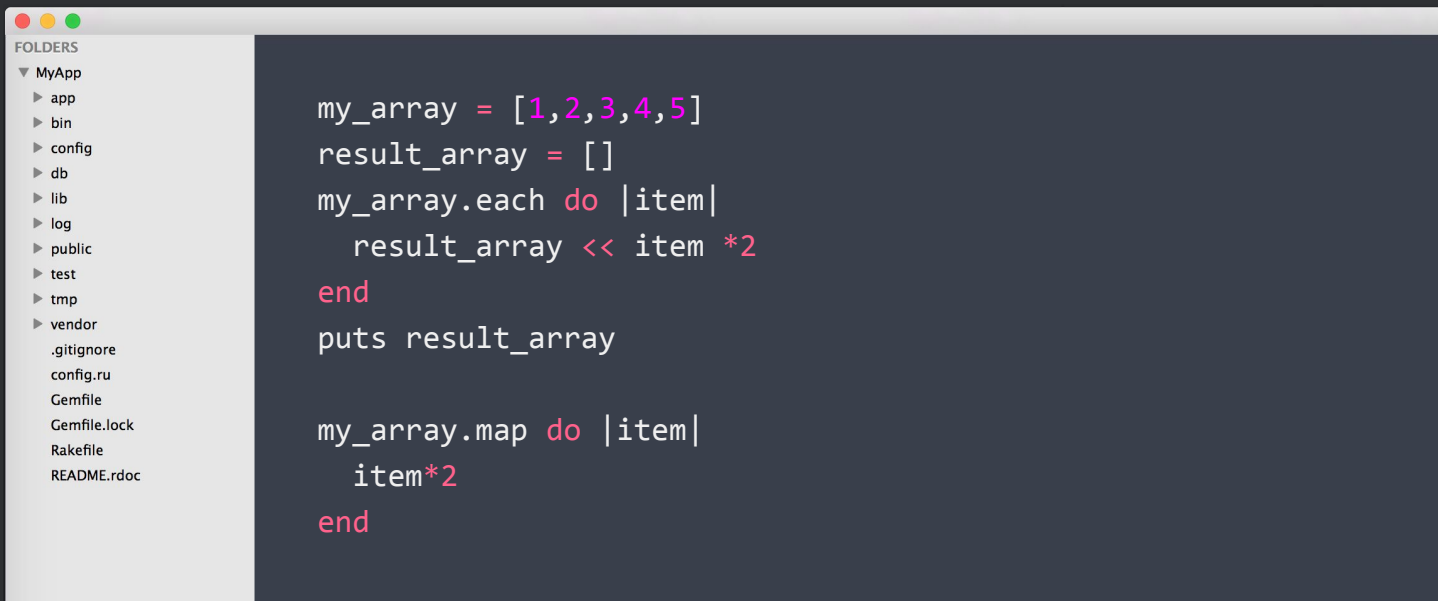
Exercise: select

Given a hash with all the names of the people in class and their ages, return those who are older than 25



Iteration: map

Map returns a new array with the results of running a block once for every element in enum. Again, the original is left untouched.



The image shows a screenshot of a code editor with a dark theme. On the left, there is a sidebar titled 'FOLDERS' showing a project structure for 'MyApp' with subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', along with files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains two Ruby code snippets. The first snippet uses `each` to iterate over `my_array` and build a new `result_array`. The second snippet uses `map` to achieve the same result more concisely.

```
my_array = [1,2,3,4,5]
result_array = []
my_array.each do |item|
  result_array << item *2
end
puts result_array

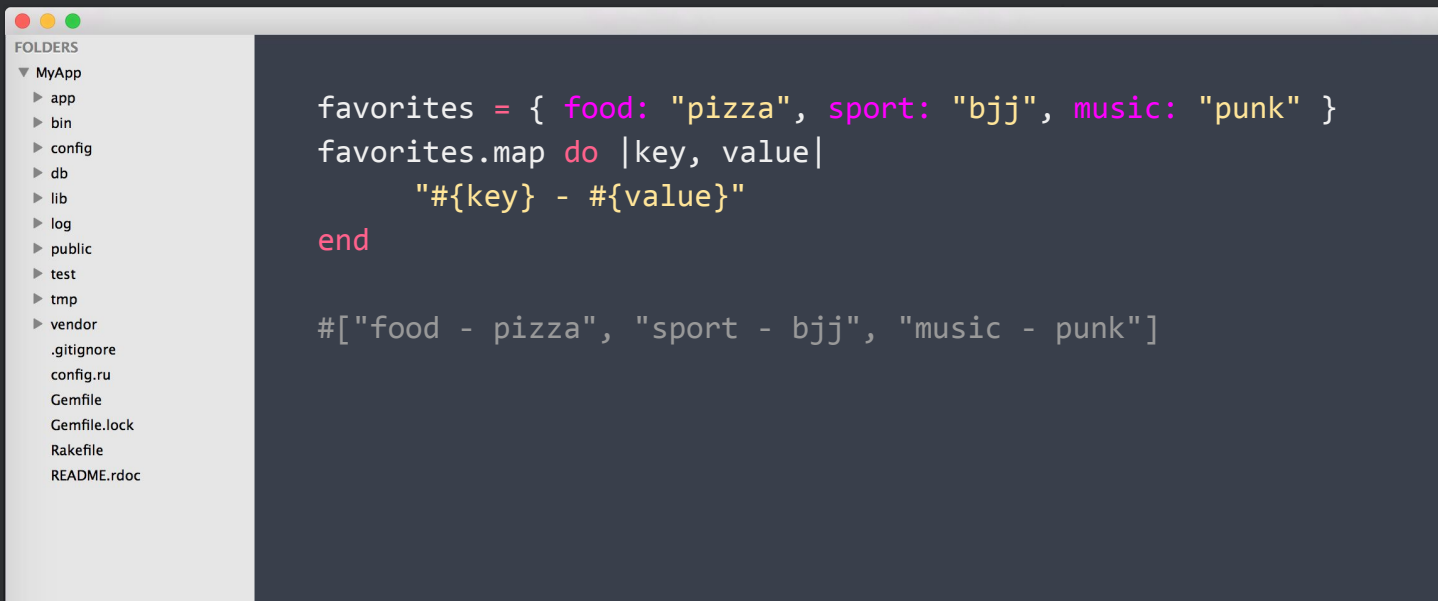
my_array.map do |item|
  item*2
end
```



@xharekx33

Iteration: map

You can use **Map** with Hashes too, but remember that it will return an Array!



The screenshot shows a code editor with a sidebar on the left labeled 'FOLDERS' containing a tree view of a project named 'MyApp'. The main editor area displays Ruby code that defines a hash 'favorites' and iterates over it using the 'map' method. The code is as follows:

```
FOLDERS
▼ MyApp
  ► app
  ► bin
  ► config
  ► db
  ► lib
  ► log
  ► public
  ► test
  ► tmp
  ► vendor
  .gitignore
  config.ru
  Gemfile
  Gemfile.lock
  Rakefile
  README.rdoc

favorites = { food: "pizza", sport: "bjj", music: "punk" }
favorites.map do |key, value|
  "#{key} - #{value}"
end

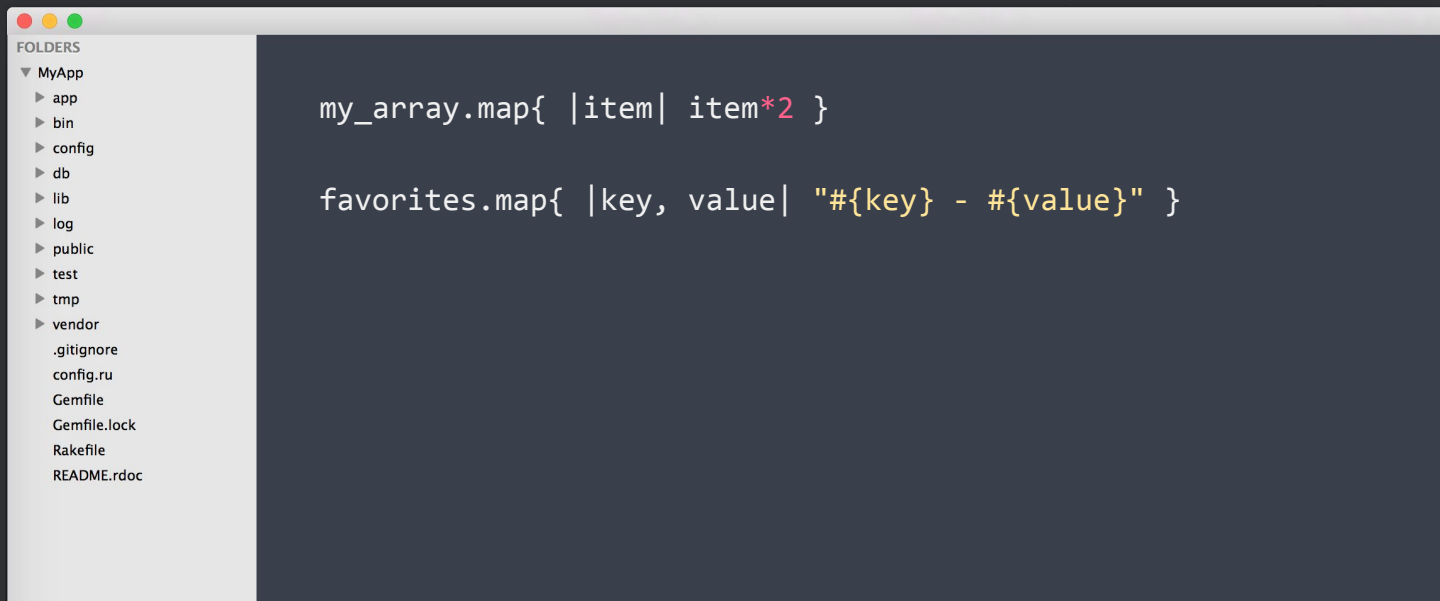
#["food - pizza", "sport - bjj", "music - punk"]
```



@xharekx33

Iteration: map in a single line

Our previous **map** calls can be also written in a single line.



@xharekx33

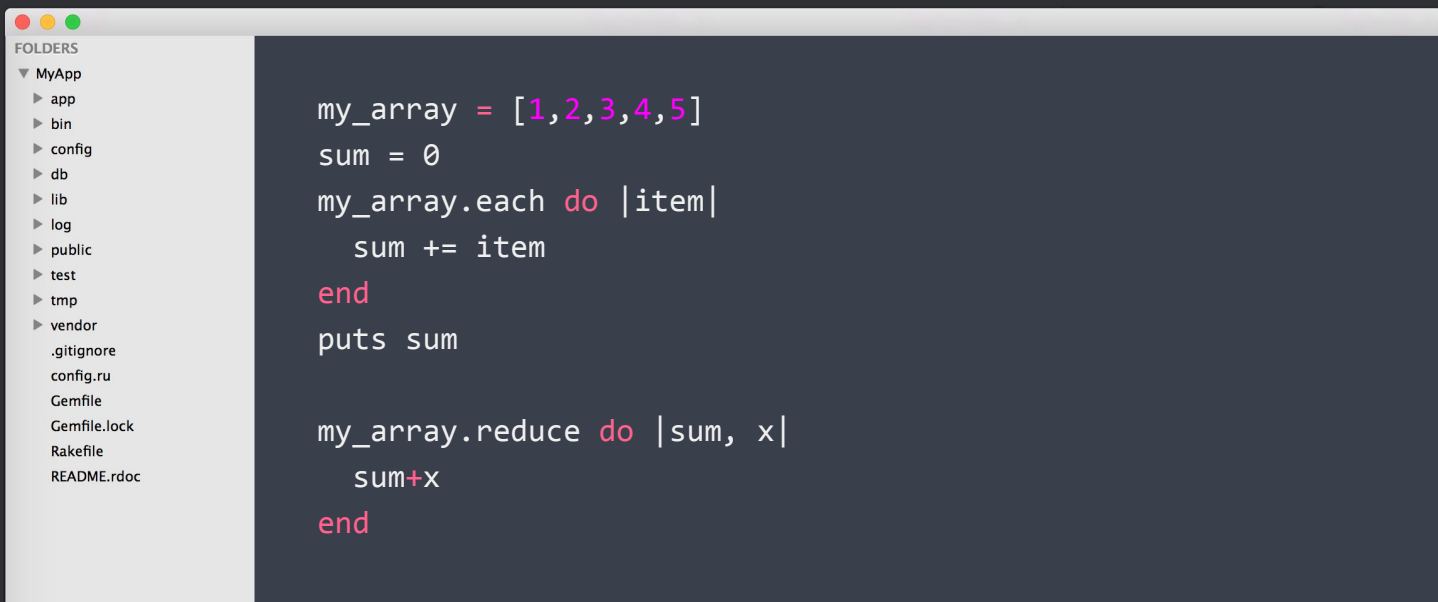
Exercise: map

Given an Array with city names all in downcase, return another with those city names properly capitalized.



Iteration: reduce

Reduce (aka "**inject**") transforms a Collection into a single value, resulting from applying a function recursively to each element in the collection.



The image shows a screenshot of a code editor with a dark theme. On the left, there is a sidebar titled 'FOLDERS' showing a project structure for 'MyApp' with subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', along with files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main editor area contains two Ruby code snippets. The first snippet demonstrates a manual iteration using 'each' to calculate the sum of an array. The second snippet demonstrates the 'reduce' method, which also calculates the sum of the array.

```
my_array = [1,2,3,4,5]
sum = 0
my_array.each do |item|
  sum += item
end
puts sum

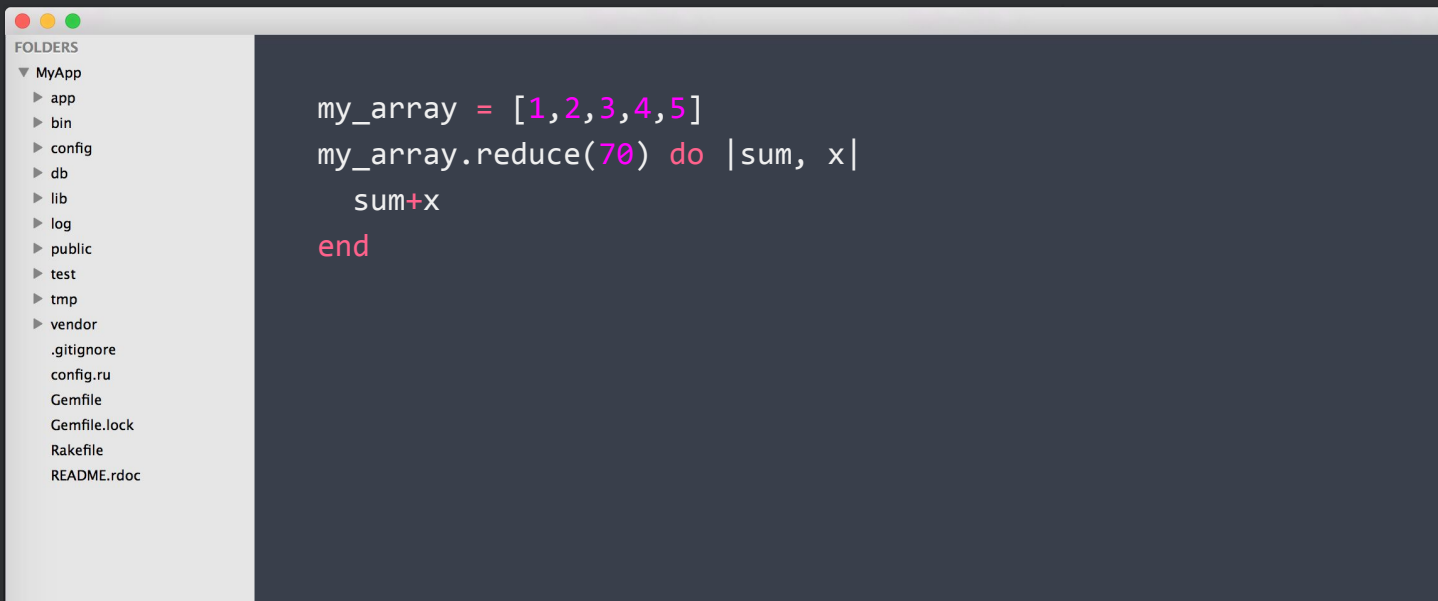
my_array.reduce do |sum, x|
  sum+x
end
```



@xharekx33

Iteration: reduce

You can initialize the accumulator ("memo") with any value you want. If you don't specify a initial value for it, then the first element of collection will be used.



The image shows a screenshot of a code editor with a dark theme. On the left, there is a sidebar titled 'FOLDERS' showing a project structure. The main area contains Ruby code demonstrating the use of the `reduce` method with an initial value.

```
FOLDERS
▼ MyApp
  ► app
  ► bin
  ► config
  ► db
  ► lib
  ► log
  ► public
  ► test
  ► tmp
  ► vendor
  .gitignore
  config.ru
  Gemfile
  Gemfile.lock
  Rakefile
  README.rdoc

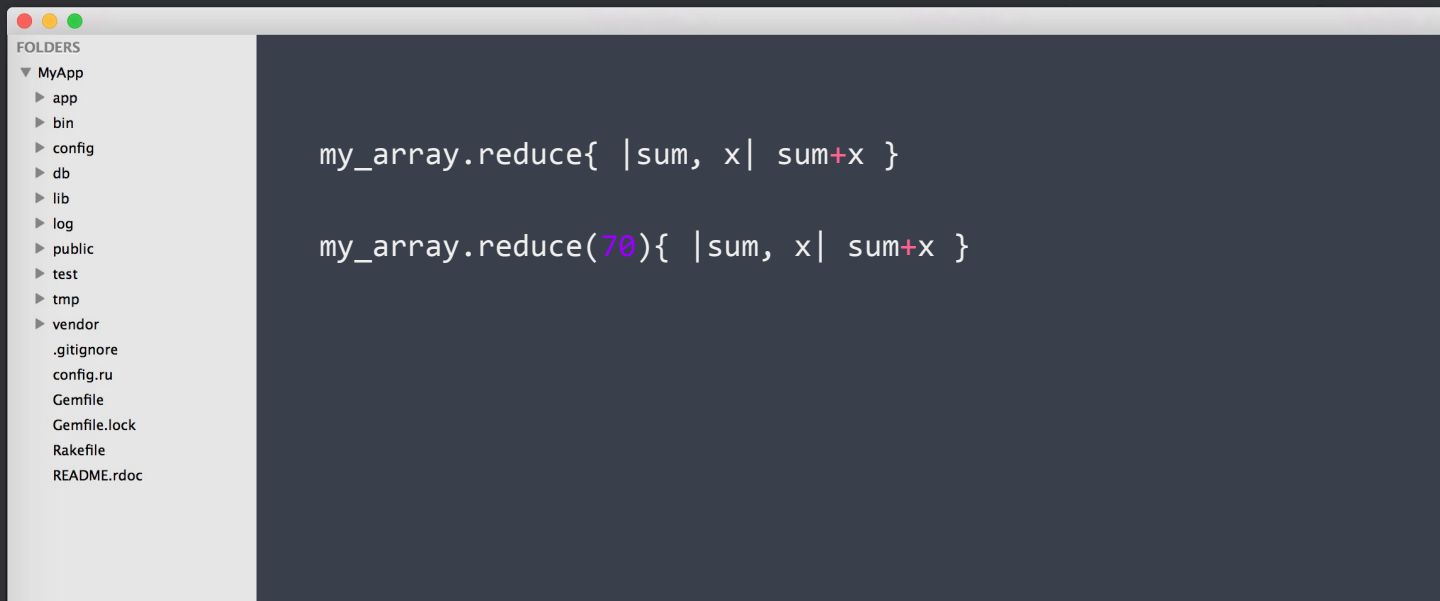
my_array = [1,2,3,4,5]
my_array.reduce(70) do |sum, x|
  sum+x
end
```



@xharekx33

Iteration: reduce in a single line

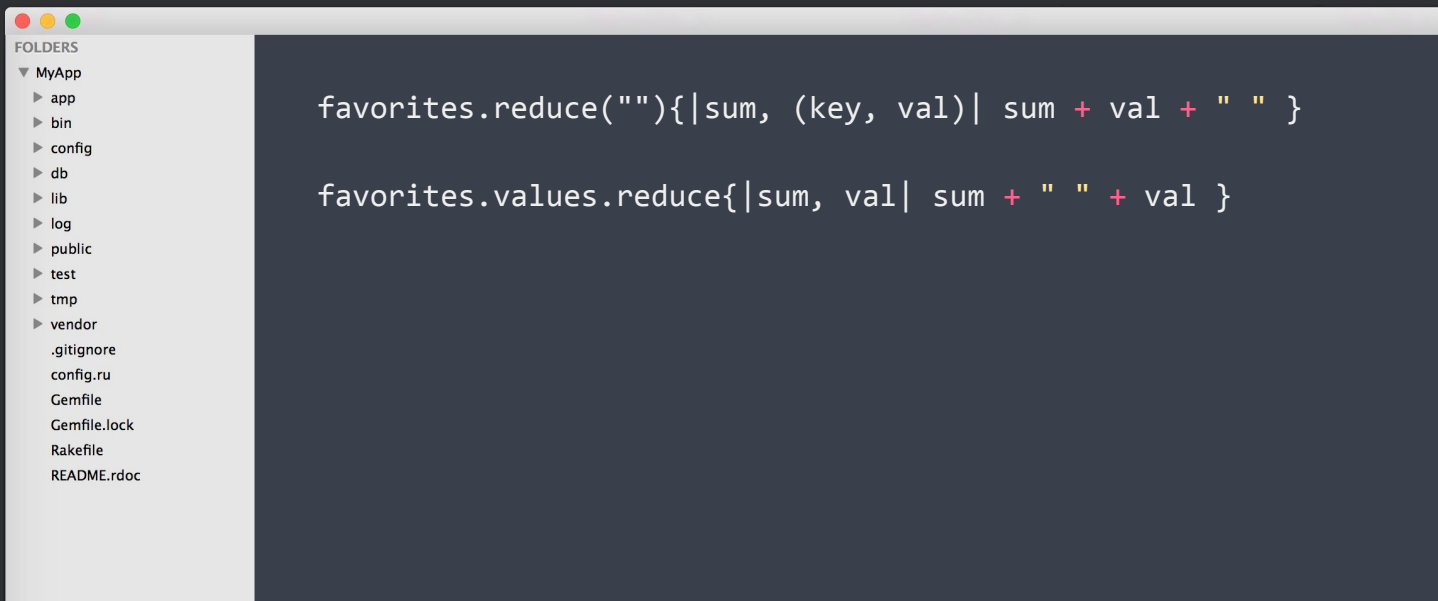
Once again, our previous calls to **reduce** can be written in a single line



@xharekx33

Iteration: reduce

You can use reduce with Hashes, but it's often just simpler to use the Hash **values** method instead and work with the Array it returns.



The image shows a screenshot of a code editor with a sidebar on the left and a main code area on the right. The sidebar, titled 'FOLDERS', shows a project structure with a 'MyApp' folder containing subfolders like 'app', 'bin', 'config', 'db', 'lib', 'log', 'public', 'test', 'tmp', and 'vendor', as well as files like '.gitignore', 'config.ru', 'Gemfile', 'Gemfile.lock', 'Rakefile', and 'README.rdoc'. The main code area contains two lines of Ruby code:

```
favorites.reduce(""){|sum, (key, val)| sum + val + " " }  
  
favorites.values.reduce{|sum, val| sum + " " + val }
```



@xharekx33

Exercise: reduce

Given an Array with city names, return the longest.



@xharekx33

Iteration methods quick summary

As a simple summary, so it's easy to remember what each method does:

- **each**: Traverse
- **select**: Filter
- **map**: Transform
- **reduce**: Accumulate



Enumerable

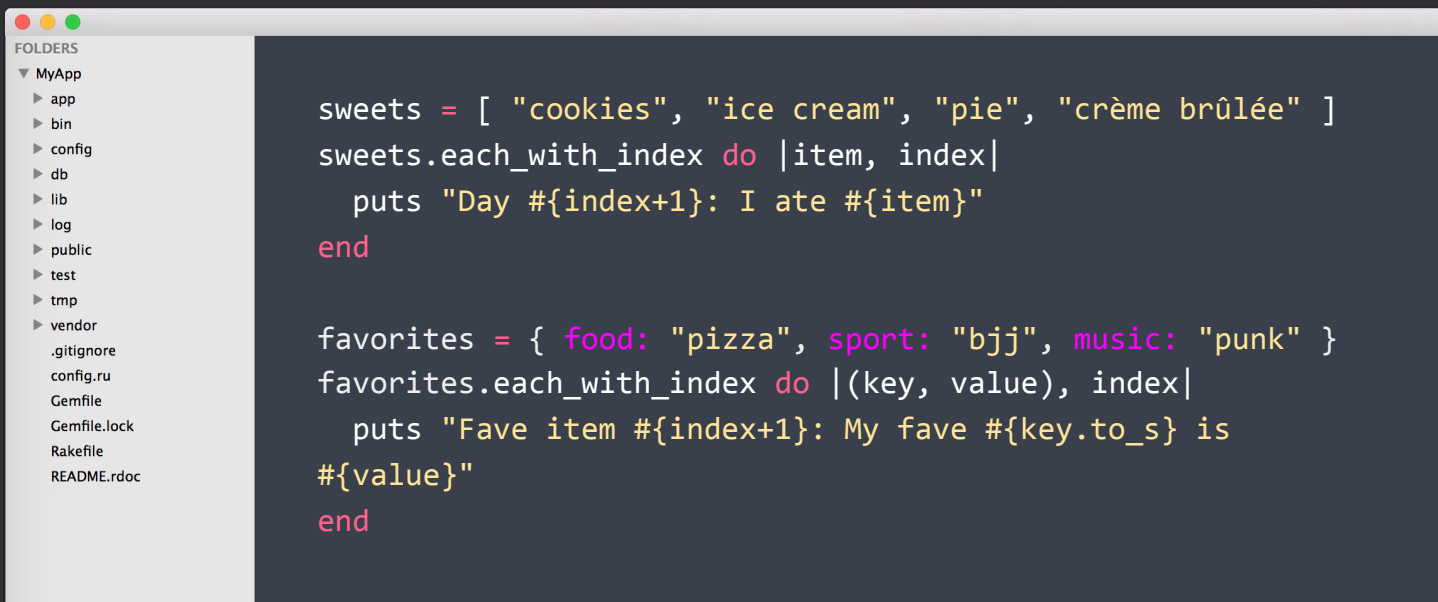
Collection classes implement an **each** method to get successive members of the collection. Because of it they can take advantage of the methods in **Enumerable**.

This module provides several traversing, searching and sorting methods that we can use with our Arrays and Hashes, and that are included by default.



More methods: each with index

If you want to use each, but you need the index along with the item you can use the `each_with_index` method



```
FOLDERS
▼ MyApp
  ► app
  ► bin
  ► config
  ► db
  ► lib
  ► log
  ► public
  ► test
  ► tmp
  ► vendor
  .gitignore
  config.ru
  Gemfile
  Gemfile.lock
  Rakefile
  README.rdoc

sweets = [ "cookies", "ice cream", "pie", "crème brûlée" ]
sweets.each_with_index do |item, index|
  puts "Day #{index+1}: I ate #{item}"
end

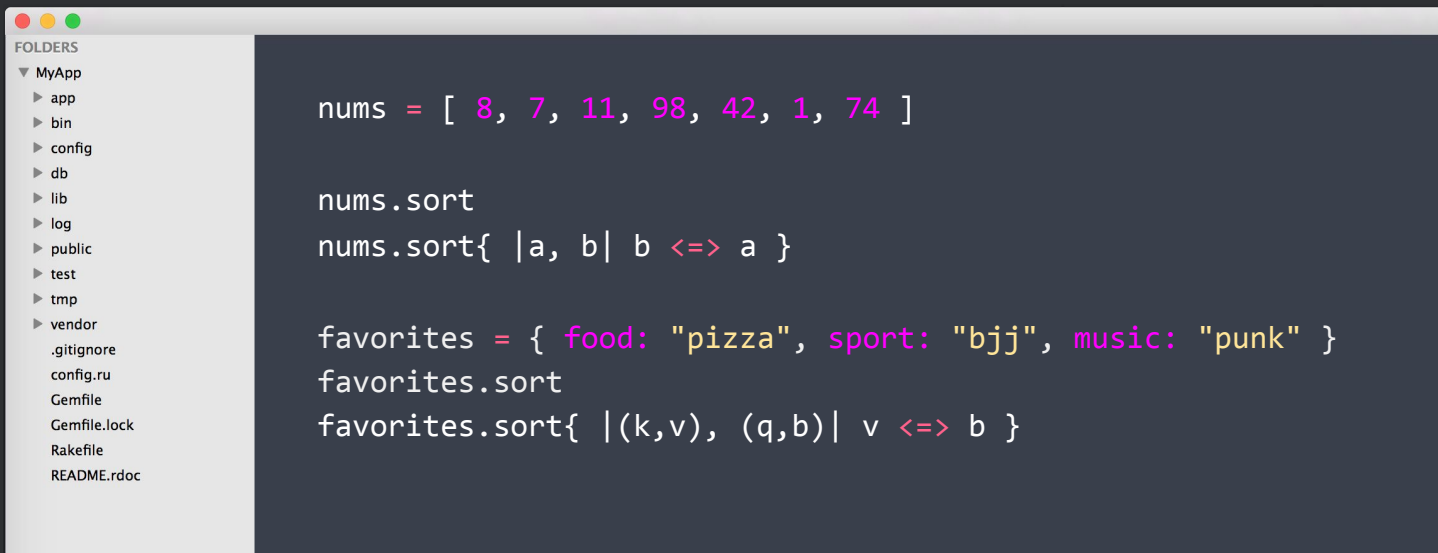
favorites = { food: "pizza", sport: "bjj", music: "punk" }
favorites.each_with_index do |(key, value), index|
  puts "Fave item #{index+1}: My fave #{key.to_s} is
#{value}"
end
```



@xharekx33

More methods: sort

To sort an array you can use `sort`, that will sort items either according to their own `<=>` method, or by using the results of the supplied block.



@xharekx33