Full Stack Notes

Array and Hash Traversals

Arrays and Hashes are well-loved and well-used as data collections by Rubyists.

Table of Contents

Traversing Arrays

In most cases we use looping structures to traverse arrays.

Here's an example of an array traversal **in Java**:

```
String[] ghosts = {"Blinky", "Pinky", "Inky", "Clyde"};

for (int i = 0; i < ghosts.length; i++) {
  System.out.println(ghosts[i]);
}
```

RESOURCES

- Pacman

Each Loops

In Ruby we can iterate through our ghost names using the `each` method of the Array class.

```
ghosts = %w[Blinky Pinky Inky Clyde]

ghosts.each do |ghost_name|
  puts ghost_name
end
```

Note that the `each` method takes a block as an argument. We could also have use a curly-brace style block:

```
ghosts.each { |ghost_name| puts ghost_name }
```

Indexed Each Loops

While traversing an array you may require access to the currently index position:

```
ghosts = %w[Blinky Pinky Inky Clyde]


ghosts.each_index do |i|
  puts "#{i}: #{ghosts[i]}"
end
```

This isn't nearly as elegant as a plain `each` iterator, but gets the job done.

Output:

```
0: Blinky
1: Pinky
2: Inky
3: Clyde
```

Traversing Hashes

Traversing a hash is similar to traversing an Array.

```
ghost_dictionary = { 'Blinky' => 'Shadow',
                     'Pinky'  => 'Speedy',
                     'Inky'   => 'Bashful',
                     'Clyde'  => 'Pokey'    }


ghost_dictionary.each do | nickname, character |
  puts "#{nickname} also know as #{character}."
end
```

Note the use of white-space in the hash definition. This is for human readability.

Output:

```
Blinky also know as Shadow.
Pinky also know as Speedy.
Inky also know as Bashful.
Clyde also know as Pokey.
```

RESOURCES

- Pacman Ghost Characters and Nicknames

A Map is a Conversion Loop

Sometimes we wish to transforms one collection into another. The `map` method makes this simple:

```
secrets = ["eht", "tsohg", "lliw", "ekirts", "ta", "thgindim"]

decoded = secrets.map { |word| word.reverse }

# decoded equals:["the", "ghost", "will", "strike", "at", "midnight"]
```

Map takes a block, passes each array element into that block, and produces a second array based on the block's return value.

```
COMBINED_TAX_RATE = 0.11 # 11%
product_prices = [12.34, 839.00, 90.95, 100]

product_taxes = product_prices.map { |price| price * COMBINED_TAX_RATE }
# product_taxes equals: [1.3574, 92.29, 10.0045, 11.0]
```

We Can Reduce Collections Too

Sometimes we want to reduce a collection down to a single value:

```
product_prices = [12.34, 839.00, 90.95, 100]
total_price = product_prices.sum
max_price   = product_prices.max
min_price   = product_prices.min
```

The `reduce` method lets us write customer reducers. Here's `sum` rewritten as a `reduce`:

```
product_prices = [12.34, 839.00, 90.95, 100]
total_price = product_prices.reduce(0) { |sum, price| sum + price }


# If your reduce block involves a single operator like this it can be refactored to:
total_price = product_prices.reduce(:+)
```

We can also `reduce` hashes, by first grabbing only the values:

```
toys_and_prices = { lego: 120.30, doll: 30.23, catan: 40.55 }
total_price = toys_and_prices.values.reduce(:+)
```

RESOURCES

- Guide to Handy Ruby Array Helper Methods
- Get the most out of Ruby by using the .select .map and .reduce methods together