Ruby Hashes

Ruby hashes are more advanced collections of data and are similar to objects in JavaScript and dictionaries in Python.

A **hash** is a data structure that stores items by **associated keys.** This is contrasted against arrays, which store items by an ordered index. Entries in a hash are often referred to as **key-value pairs.** This creates an associative representation of data.
Most commonly, a hash is created using symbols as *keys* and any data types as *values*.
All key-value pairs in a hash are surrounded by curly braces {} and comma separated.

Keys and values are associated with a special operator called a **hash rocket**: **=>.**

Creating Hashes

```
my_hash = {
  "a random word" => "ahoy",
  "Dorothy's math test score" => 94,
  "an array" => [1, 2, 3],
  "an empty hash within a hash" => {}
}
```

Accessing Values

You can access values in a hash the same way that you access elements in an array. When you call a hash's value by key, the key goes inside a pair of brackets, just like when you're calling an array by index.

```
shoes = {
  "summer" => "sandals",
  "winter" => "boots"
}

shoes["summer"]   #=> "sandals"
```

If you try to access a key that doesn't exist in the hash, it will return `nil`:

```
shoes["hiking"]   #=> nil
```

Sometimes, this behavior can be problematic for you if silently returning a nil value could potentially wreck havoc in your program. Luckily, hashes have a fetch method that will raise an error when you try to access a key that is not in your hash.

```
hoes.fetch("hiking")   #=> KeyError: key not found: "hiking"
```

Alternatively, this method can return a default value instead of raising an error if the given key is not found.

```
shoes.fetch("hiking", "hiking boots") #=> "hiking boots"
```

Adding and Changing Data

You can add a key-value pair to a hash by calling the key and setting the value, just like you would with any other variable.

```
shoes["fall"] = "sneakers"

shoes     #=> {"summer"=>"sandals", "winter"=>"boots",
"fall"=>"sneakers"}
```

You can also use this approach to change the value of an existing key.

```
shoes["summer"] = "flip-flops"
shoes     #=> {"summer"=>"flip-flops", "winter"=>"boots",
"fall"=>"sneakers"}
```

Removing Data

Deleting data from a hash is simple with the hash's `#delete` method, which provides the cool functionality of returning the value of the key-value pair that was deleted from the hash.

```
shoes.delete("summer")    #=> "flip-flops"
shoes
                  #=> {"winter"=>"boots", "fall"=>"sneakers"}
```

Methods

Hashes respond to many of the same methods as arrays do since they both employ Ruby's **Enumerable** module.

A couple of useful methods that are specific to hashes are the **#keys** and **#values methods**, which very unsurprisingly return the keys and values of a hash, respectively. **Note that both of these methods return *arrays***.

```
books = {
  "Infinite Jest" => "David Foster Wallace",
  "Into the Wild" => "Jon Krakauer"
}

books.keys      #=> ["Infinite Jest", "Into the Wild"]
books.values    #=> ["David Foster Wallace", "Jon Krakauer"]
```

Merging Two Hashes

Occasionally, you'll come across a situation where two hashes wish to come together in holy union. Luckily, there's a method for that. (No ordained minister required!)

```
hash1 = { "a" => 100, "b" => 200 }
hash2 = { "b" => 254, "c" => 300 }
hash1.merge(hash2)      #=> { "a" => 100, "b" => 254, "c" => 300 }
```

Notice that the values from the hash getting merged in (in this case, the values in `hash2`) overwrite the values of the hash getting… uh, merged *at* (`hash1` here) when the two hashes have a key that's the same.

Symbols as Hash Keys

In this lesson, we mostly used strings for hash keys, but in the real world, you'll almost always see **symbols (like `:this_guy`)** used as keys. This is predominantly because symbols are far more performant than strings in Ruby, but they also allow for a much cleaner syntax when defining hashes. Behold the beauty:

```
# 'Rocket' syntax
american_cars = {
  :chevrolet => "Corvette",
  :ford => "Mustang",
  :dodge => "Ram"
}
# 'Symbols' syntax
japanese_cars = {
  honda: "Accord",
  toyota: "Corolla",
  nissan: "Altima"
}
```

That last example brings a tear to the eye, doesn't it? Notice that the hash rocket and the colon that represents a symbol have been mashed together. This unfortunately only works for symbols, though, so don't try { 9: "value" } or you'll get a syntax error.

When you use the cleaner 'symbols' syntax to create a hash, you'll still need to use the standard symbol syntax when you're trying to access a value. In other words, regardless of which of the above two syntax options you use when creating a hash, they both create symbol keys that are accessed the same way.

```
american_cars[:ford]    #=> "Mustang"
japanese_cars[:honda]   #=> "Accord"
```

Iterating Over Hashes

Because hashes can have multiple elements in them, there will be times when you'll want to iterate over a hash to do something with each element. Iterating over hashes is similar to iterating over arrays with some small differences. We'll use the each method again and this time we'll create a new file to test this out.

```
# iterating_over_hashes.rb

person = {name: 'bob', height: '6 ft', weight: '160 lbs', hair: 'brown'}

person.each do |key, value|
  puts "Bob's #{key} is #{value}"
end
```

We use the each method like before, but this time we assign a variable to both the key and the value. In this example we are setting the key to the key variable and the value to the value variable. Run this program at the command line with ruby iterating_over_hashes.rb to see the results. The output is:

```
Bob's name is bob
Bob's height is 6 ft
Bob's weight is 160 lbs
Bob's hair is brown
```

Hashes as Optional Parameters

You may recall in chapter three on methods, we talked about the ability to assign default parameters to your methods so that the output is always consistent. You can use a hash to accept optional parameters when you are creating methods as well. This can be helpful when you want to give your methods some more flexibility and expressivity. More options, if you will! Let's create a method that does just that.

```
# optional_parameters.rb

def greeting(name, options = {})
  if options.empty?
    puts "Hi, my name is #{name}"
  else
    puts "Hi, my name is #{name} and I'm #{options[:age]}" +
         " years old and I live in #{options[:city]}."
  end
end

greeting("Bob")
greeting("Bob", {age: 62, city: "New York City"})
```

We used Ruby hash's empty? method to detect whether the options parameter, which is a hash, had anything passed into it. You haven't seen this method yet but you can infer what it does. You could also check out the Ruby Docs to look up the method as well. At the end we called the method twice. Once using no optional parameters, and a second time using a hash to send the optional parameters. You can see how using this feature could make your methods much more expressive and dynamic.

And finally, to add a small twist, you can also pass in arguments to the greeting method like this:

```
greeting("Bob", age: 62, city: "New York City")
```

Notice the curly braces, { }, are not required when a hash is the last argument, and the effect is identical to the previous example. This convention is commonly used by Rails developers. Understanding this concept alone should help you decipher some previously cryptic Rails code!

Hashes vs. Arrays

This chapter and the last covered two very important and widely used data structures: hashes and arrays. It can be a bit overwhelming when you look at all of the different ways there are to represent data with code. Don't feel too daunted. Pick these things up in small parts and apply them. Then add more little parts as you move along. It's impossible to know everything in the beginning so put some effort into learning a few things well and then build from there.

When deciding whether to use a hash or an array, ask yourself a few questions:

•Does this data need to be associated with a specific label? If yes, use a hash. If the data doesn't have a natural label, then typically an array will work fine.

•Does order matter? If yes, then use an array. As of Ruby 1.9, hashes also maintain order, but usually ordered items are stored in an array.

•Do I need a "stack" or a "queue" structure? Arrays are good at mimicking simple "first-in-first-out" queues, or "last-in-first-out" stacks.

As you grow as a developer, your familiarity with these two data structures will naturally affect which one you reach for when looking to solve specific problems. The key is to practice and experiment with each to find out which data structure works best in certain situations.