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"
```