Introduction

In this lesson we're going to use IRB to play around with **string objects and methods**.

Working with Strings in IRB

A **string** in Ruby is a collection of characters. We create a string literal by writing a group of characters inside quote marks:

```
"abc"
```

We can also use single quote marks if we prefer, although notice that the output still places the string inside double quotes:

```
'abc'
```

If you want to use quote marks inside a string literal, then you need to use the other type of quote marks:

```
"my string's value"
```

Another option, is to do what's called **escaping** the quotation mark, where you put a backslash before the symbol (apostrophe, for example). It will then appear as an apostrophe inside the string instead of terminating the string:

```
'my string\'s value`
```

In the last lesson, I mentioned that everything in Ruby is an **object**. Objects can perform actions called **methods**. There are a few ways of getting an object to perform a method, but the most common is the **dot notation**, where you write the object followed by a dot and then the name of the method that you want to call.

Let's try calling the `length` method:

```
'abc'.length
```

This will tell us the number of characters in the that string.

Now let's move on to **variables** that are very common in programming languages.

Variables

Variables are a way of storing an object in memory for later use. In Ruby, we can assign a variable to an object very easily, using the equals operator:

```
my_string = 'abc'
```

Here The variable `my_string` refers to the string `abc`. This means I can call methods on the variable than having to type in the string every time. So now if I want to a call a method, all I need to do is type in

```
my_string.length
```

Let's try a few more different methods that can be used on string objects. For example, the `reverse` method should write the string backwards:

```
my_string.reverse
```

Down and Up Case Methods

There's also the `upcase` method, that writes a string in capital letters as well as the `downcase`, that will write it in all lowercase letters:

```
my_string.upcase
my_string.downcase
```

There are loads of other string methods. You can find out what they all are by calling the `methods` method on any string:

```
my_string.methods
```

You can find out more about them all by reading the documentation that can be found in the "Resources" section.

Bang Methods

All the methods that we've seen so far haven't actually changed the object that called the method itself. For example, if I have a look at the `my_string` variable again by just typing

```
my_string
```

and pressing `Enter`, we can see it still contains the string I typed in originally. It hasn't been reversed and it hasn't been changed into uppercase or downcase characters.

However if I create a new variable called `fruit` and store the string "apple" inside that variable we can have a look at some different types of methods that do change the value of the string:

```
fruit = 'apple'
```

The way we do this is by using what's it called **bang methods**. These are methods that end in an exclamation mark or a bang symbol. So we can write

```
fruit.reverse!
```

Now check the `fruit` variable:

```
fruit
```

It has changed for good, it's been reversed and has stayed reversed, so the string has been modified permanently.

Most string methods have a bang equivalent. For example, there's also an `upcase!` method:

```
fruit.upcase!
```

If there is a bang on the end of a Ruby method, it usually means "be careful, this method could be dangerous". In the case of the examples we just looked at, it means that the string will be changed for good, so you need to take extra care whenever using a bang method.

Chaining Methods

You don't have to apply methods one at a time. You can **chain** the methods together one after the other:

```
fruit = 'apple'
fruit.reverse.upcase
```

This will have the effect of reversing and writing the fruit in capital letters all at once.

The two methods are applied in order from **left to right**. Although in this case it doesn't matter, in other cases it may, so you have to be very careful about the order you chain the methods together in.

String Interpolation

String interpolation is a neat way of inserting Ruby code into a string. This is done by placing the code that you want to be evaluated inside curly brackets with a hash symbol at the beginning:

```
puts "The result is #{1 + 1}"
```

```
fruit = 'apple'
puts "The fruit is #{fruit}"
```

One important thing to note is that when you're using string interpretation, the string must be a double quoted string. Single-quoted strings do not form interpolation:

```
'The result is #{1 + 1}'
```

This will just display the actual string that was entered - nothing actually gets interpolated.