Introduction

In the video, we're going to be looking at how to define our own **methods**.

Create a new file called *methods.rb*:

```
def hello
  "Hello"
end
```

We use `def` keyword to define a method. It is then followed by the name of the method, which in this case is `hello`. The code for the method comes next before finishing with `the` end keyword here. The last line in any method in Ruby, is it's return value, which is the value that will be returned by the method when it's called. So this method will simply return the string "Hello".

Looking at methods.rb in IRB

Launch IRB in your terminal. We need to require that file and to do this, you need to make sure that when you launched IRB, you were in the same directory that the file was saved in.

```
require "./methods.rb"
```

This is very similar to the way that we require the Sinatra gem in our other programs. Now we have access to all of the code inside that file.

To call a method, simply enter its name:

```
hello
```

you can call the same method over and over again.

DRY Principle

This is one of the advantages of using methods: you don't need to write repetitive blocks of code over and over again. Another advantage is that if you want some of the functionality to change, then you only need to update the method in one place. This is known as the **DRY Principle**, which stands for **Don't Repeat Yourself**, and it's a very important principle to keep in mind when programming.

Method say_hello_to

Methods can be made more effective by including **parameters** - values that are provided for the method to use:

```ruby
def say_hello_to(name)
  "Hello #{name}!"
end
```

Parameter `name` comes after method's name inside the parentheses. We can refer to this parameter, as if it was a variable inside the body of the function.

The parentheses are actually optional, but it's always a good idea to use them, because it makes it clearer that this is a parameter.

Default Arguments

We can also provide a default argument for a parameter by putting it equal to the default value in the method definition:

```ruby
def say_hello_to(name="Ruby")
  "Hello #{name}"
end
```

So if I call the method without any arguments, it'll be automatically set to "Ruby".

Method say_hi_to

We can go ahead and add more and more parameters and give some of them default arguments.One thing to keep in mind, it's important that anything that has default argument needs to come last.

```ruby
def say_hi_to(name,job="programming language")
  "Hi #{name}! Your job as a #{job} sounds fun"
end
```

The `job` has a default value. Any parameter that does not have a default value comes first as you can see with `name`. That will have to be entered by the person calling the method, whereas job will be given the default value of programming language if another value is not entered as an argument.

We can use as many parameters as we like, but anything that does not have a default value will have to come first, and we can put default values at the end. One of the problems with this approach, though, is as the number of arguments increases, it can be difficult to remember the order that they appear in.

Keyword Arguments

One way to get round this, is to use what are called **keyword arguments** instead of just using normal default parameters. These acts by using a hash-like syntax for the parameters:

```
def greet(name: "Ruby", greeting: "Hello", job: "programming language")
  "#{greeting} #{name}, Your job as a #{job} sounds fun"
end
```

It looks just like a hash.

When calling this method you can define some arguments while omitting others:

```
greet(name: "Daz")
```

One of the nice things about using keyword arguments is that you don't need to put them in the same order that they're listed in the method definition. You don't even have to use them all as we just saw.

You can see this style of defining method often used in Ruby on Rails framework (it employs a lot of options and settings when methods are called).

Method group_greeting

We can also create methods with an unspecified number of arguments. This is done by placing an asterisk symbol in front of the last parameter in the method definition.

```
def group_greeting(*names)
  names.each { |name| puts "Hello #{name}!" }
end
```

* means that any number of arguments can be used - they will be stored in an array that will have the same name as the parameter (`names` in this case).

We can then iterate over the array using the `each` method just to display a message that uses `puts` to output and say hello to each name in the array. This is useful if we don't know how many arguments we're going to have in the first place.

Block as a Parameter

It's also possible to add a block as a parameter to a method by placing an ampersand symbol before its name:

```
def repeat(number=2,&block)
  number.times { yield }
end
```

The block can then be accessed in the method definition, by merely referring to it. This is useful if you want to run some specific code when a method is called.

Yield Keyword

In this example we use the `yield` keyword - it means “run the block that was given to you”. What will happen is the block will be run as many times as the number states.

You can run it like this:

```
repeat(5) { puts 'hi' }
```

This makes the methods much more flexible, that different pieces of code can be used in the same method. We can make the block of code optional. We do this by using a handy method `block_given?` that allows to check if a block was actually provided as an argument.

Method Roll

```
def roll(sides=6,&block)
  if block_given?
    yield(rand(1..sides))
  else
    rand(1..sides)
  end
end
```

This method demonstrates the rolling of a dice. It takes the number of sides as an argument that defaults to six, and accepts a block as well.

However we check to see if a block is given when the method is called. If the block isn’t given, we go to the `else` clause, and all we do is return a random number from one to the number of sides. If the block is given, then we actually use `yield` to call that piece of code with an argument.

You can pass arguments to the `yield` command, so this block will take an argument that will be a random number from one up to the number of sides of the dice.