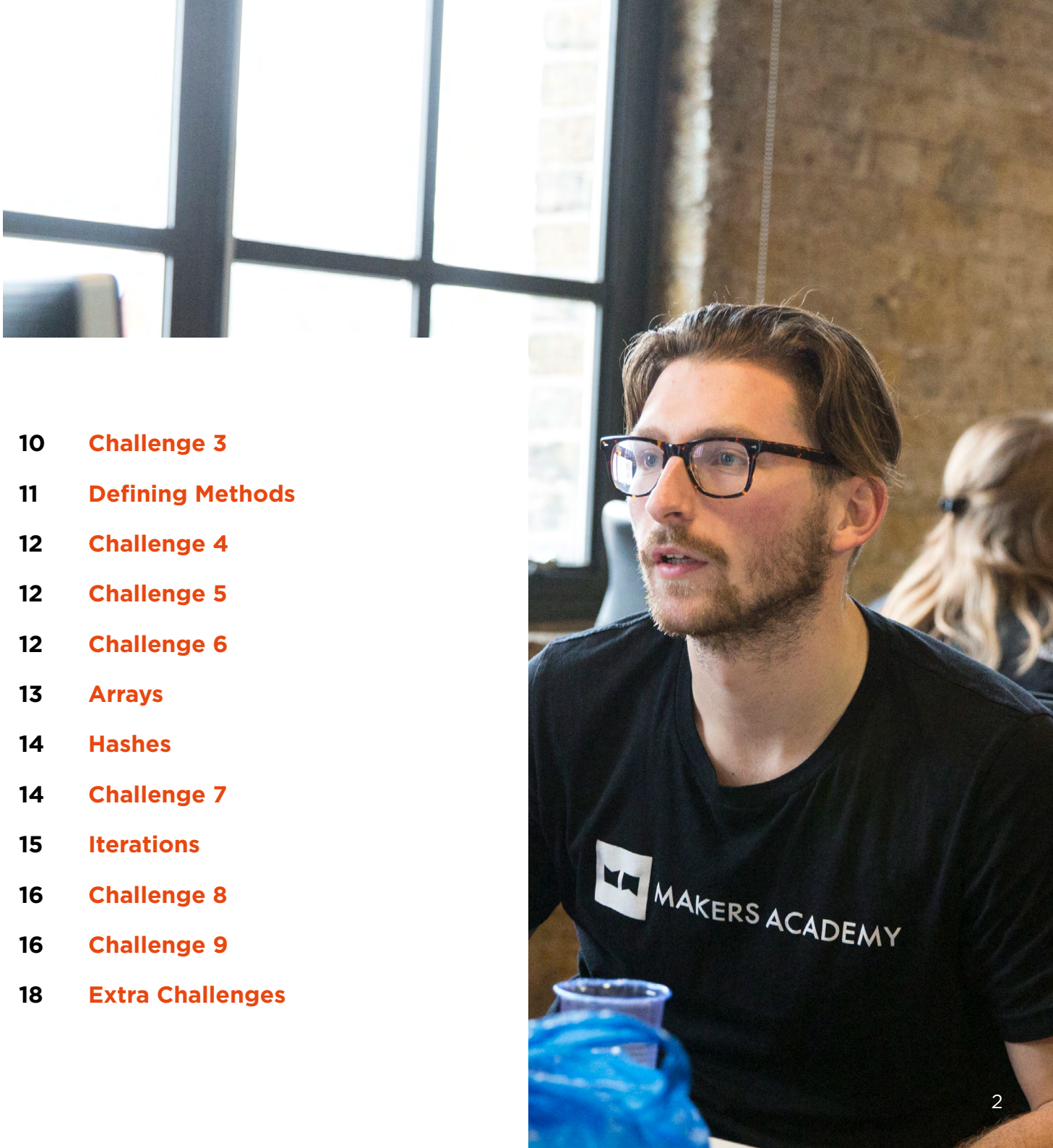
MAKERS

Coding 101: Learn Ruby in 15 minutes

Visit www.makers.tech

Contents



About Us

At Makers, we are creating a new generation of tech talent who are skilled and ready for the changing world of work. We are inspired by the idea of discovering and unlocking potential in people for the benefit of 21st century business and society.

We believe in alternative ways to learn how to code, how to be ready for work and how to be of value to an organisation. At our core, Makers combines tech education with employment possibilities that transform lives. Our intensive four-month program (which includes a month-long PreCourse) sets you up to become a high quality professional software engineer.

Makers is the only coding bootcamp with 5 years experience training software developers remotely. Your virtual experience will be exactly the same as Makers on-site, just delivered differently. If you'd like to learn more, check out **www.makers.tech.**

Installing Ruby

You'll be happy to know that Ruby comes preinstalled on all Apple computers. However we can't simply use the system defaults - just in case we mess something up!

If you've got your laptop set up already you can skip this section.

If you've never written code on your laptop and aren't interested in getting set up properly, you can use this playground area. You'll need to sign up to use it - but it's free.

Bonus

As this guide relies on you using your computer and not an online sandbox, we've made another tutorial on how to setup your computer to code here:

Prepare to code

Checking you've got Ruby

Open the terminal on your computer and then type in

```
ruby -v
```

It will return something like

```
ruby2.1.5 p273
```

or similar. You don't need to do this if you're coding online (they'll have Ruby set up for you).

This will ensure you've got Ruby set up and are ready to code.

Method Calls

In Ruby, what most languages call a 'function' is actually referred to as a 'method'. There are a couple of ways to call a method in Ruby

```
puts("Hello World!")
```

Here you're calling the method

puts passing the string **"Hello World!"**

as the argument, inside parentheses. Parentheses are often optional/implicit in Ruby, so you could just as well type:

```
puts "Hello World!"
```

It's also possible to pass multiple arguments to some methods. In these cases, each argument is separated by a comma:

```
puts("Hello World!", "I am coding.")
```

Although we often won't need parentheses, in some cases they're necessary.

Rather than worrying about when to use them, let's stick to using parentheses so we won't be tripped up by these special cases.

Variables

An ordinary, regular variable is known as a local variable. They are created through assignment using the **"="** sign

```
number = 21
```

You can have multiple words used and the general convention in Ruby is to use snake_case like so:

```
my_lucky_number = 13
```

Slightly more readable right?

Truth and Falsehood

In Ruby, everything is considered to be true, unless it's false or nil. That might seem a little strange at times, and possibly a little confusing but it's really important to remember.

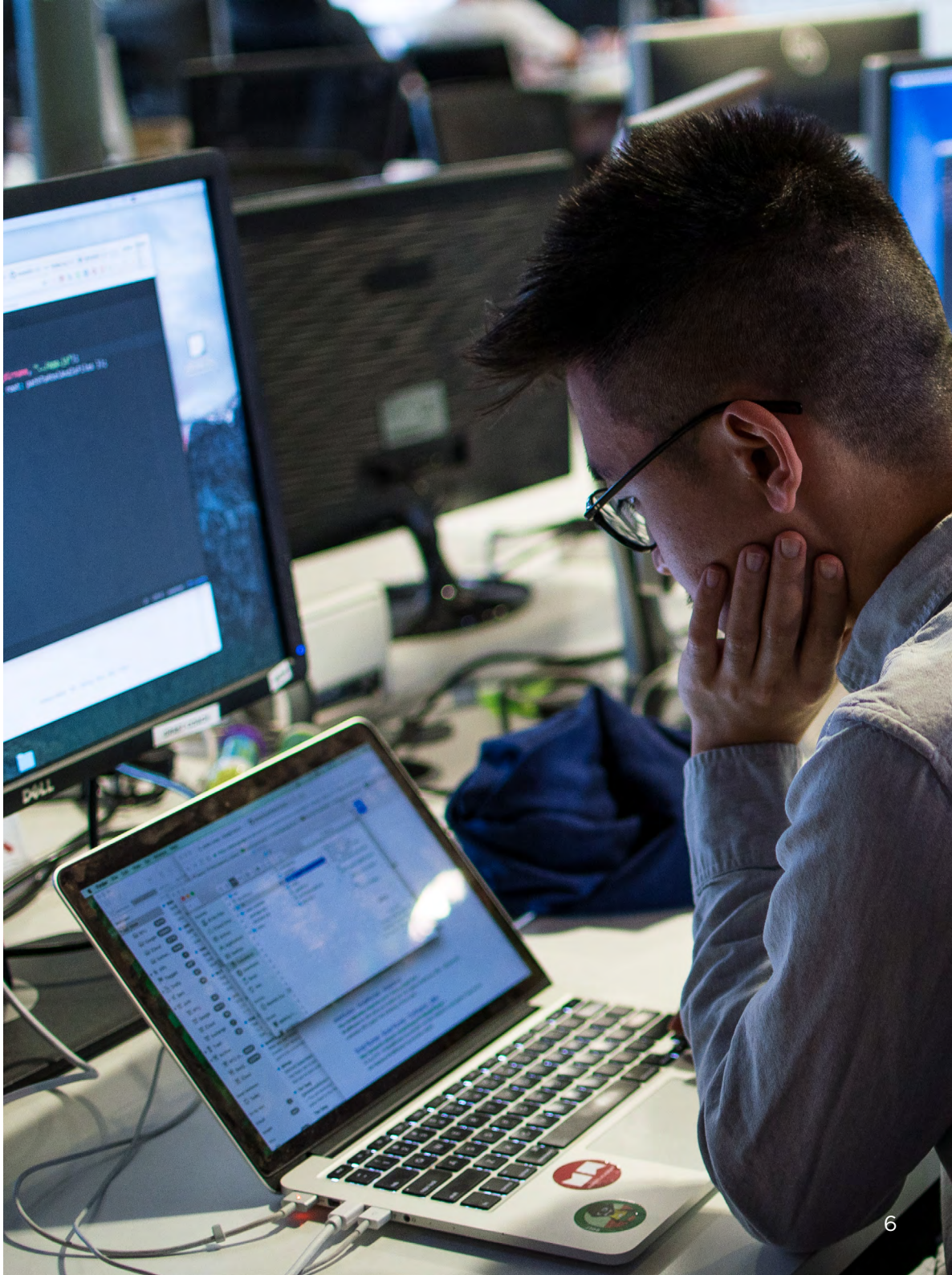
For example, 0 is true, as is String =

```
"Zero"
```

An empty string can be true as can an empty array.
All numbers are true, all strings are true, well everything is true apart from **false** and **nil**.

This will make more sense as we go through the tutorial but for the time being, just remember it.

✓ **Everything in Ruby is true, except false and nil**



Strings, Objects and methods

Strings are simply a sequence of text that are written in quotation marks.

You can use both single and double quotation marks in Ruby so both of the following work:

```
my_name = 'Dana'
my_friends_name = "Ruben"
```

You can use any character in a string, as long as it's between the quotation marks:

```
my_age = "I'm 28 years old :-)"
```

You may also have heard in Ruby, that everything is an OBJECT. Strings are OBJECTS in Ruby and this means they get their very own set of methods. That means you can do fun things like:

```
"makers".upcase
```

Try it and see what it does.

Hint: whenever you see this symbol, try out the code for yourself in ***IRB***

In the example, the object is everything before the full stop. So **"makers"** is the object.

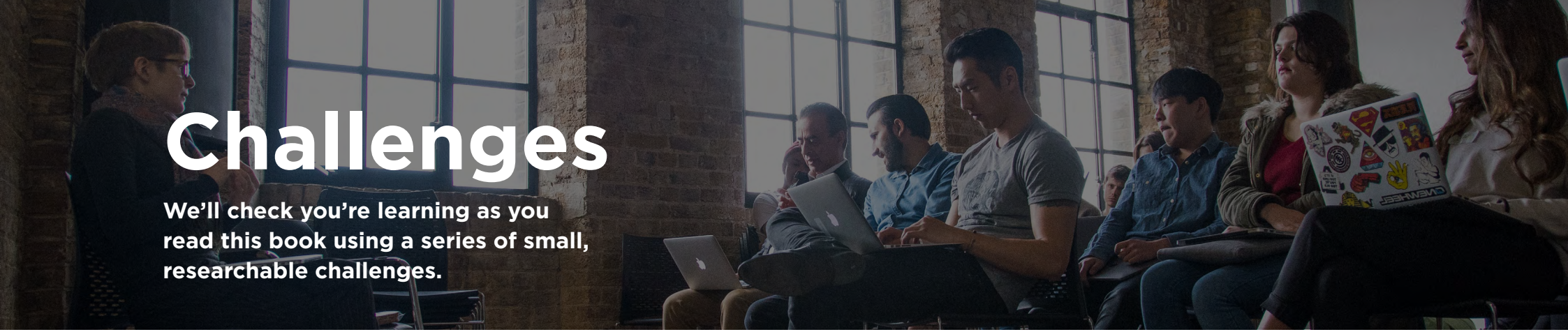
The method we are calling is **upcase** This is the method that will capitalize the object on which it is being called.

You can get a list of methods available for any object by using IRB. Try

```
"my name".methods
```

and see what it returns. There's a lot to go through, have a little play around in IRB to see what some of them do.

How can you sort these methods into alphabetical order?



Challenges

We'll check you're learning as you read this book using a series of small, researchable challenges.

Try them out in IRB or your code editor and explore the power of the Ruby language.

Challenge 1

Try playing with these string methods in IRB and see what they do.

reverse

length

capitalize

For an exhaustive list of all the current methods available you can visit

Ruby Docs

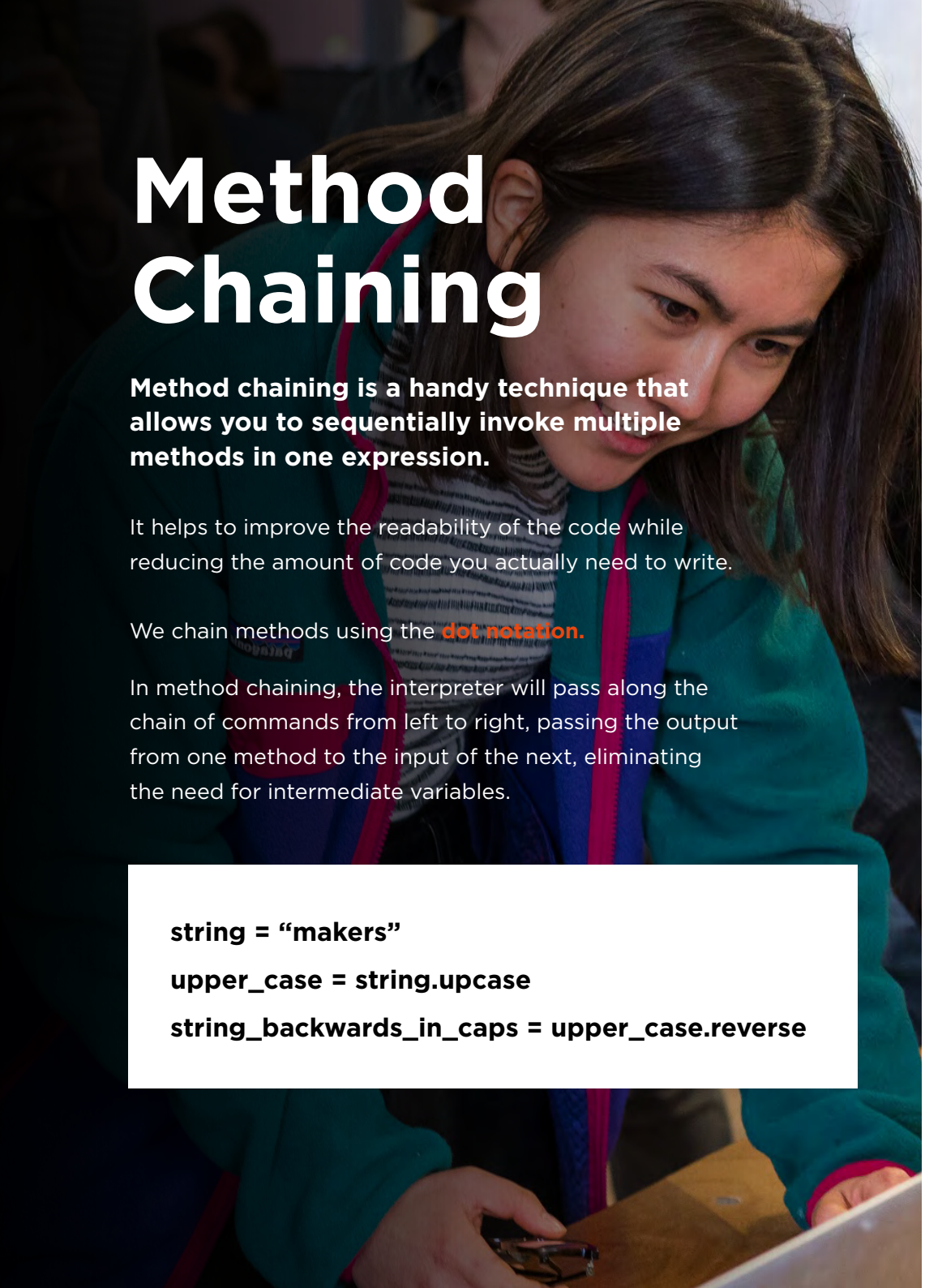
Just like with functions in other languages, Ruby methods can have arguments:

```
"learn to code".include?("code")
```

will return **true** if the **String** object **"learn to code"** includes the string **"code"** which we have *passed* as an *argument* to the **include?** method.

Challenge 2

Is the **include?** method case sensitive? Try different variations of strings to find out.



Method Chaining

Method chaining is a handy technique that allows you to sequentially invoke multiple methods in one expression.

It helps to improve the readability of the code while reducing the amount of code you actually need to write.

We chain methods using the **dot notation.**

In method chaining, the interpreter will pass along the chain of commands from left to right, passing the output from one method to the input of the next, eliminating the need for intermediate variables.

```
string = "makers"
upper_case = string.upcase
string_backwards_in_caps = upper_case.reverse
```

Using Method Chaining

```
string_backwards_in_caps = "makers".
upcase.reverse
```

In both cases, the string **"makers"** is turned into all capitals using the **upcase** method. The letters are then reversed using the **reverse** method. The new **String** in capitals and reversed is set to the **variable** called **string_backwards_in_caps**

Here is a slightly more complicated & different version:

```
"makers".upcase.reverse.next.swapcase
```

Try this in IRB and see what happens to the string.

Conditionals

Conditionals are super simple in Ruby.
They're structured as follows:

```
if some_condition
  return "this code will be executed if some_condition is true"
else
  return "this code will be executed if some_condition is false"
end
```

You'll need to put the if, else and end keywords on separate lines as shown above. Technically you don't have to indent the code, but you definitely should. It's the expected convention (and Ruby loves convention!). Not to mention the fact that it makes the code much easier to read later. But what about if you want more than one condition? You can use the keyword elsif, and you can do something like this:

```
if some_condition
  return "this code will be executed if some_condition is true"
elsif another_condition
  return "this code will be executed if another_condition is true"
elsif a_third_condition
  return "this code will be executed if a_thirdcondition is true"
else
  "this code will be executed if none of the above are true"
end
```

Challenge 3

```
BANK_BALANCE = 100

def dinner_plans
  #insert code here
end
```

Write a conditional statement that will tell me to go out for dinner if my bank balance is > 100.00 and return **"stay at home"** if not.

Defining Methods

A method always starts with the keyword `def` and finishes with the keyword `end`.

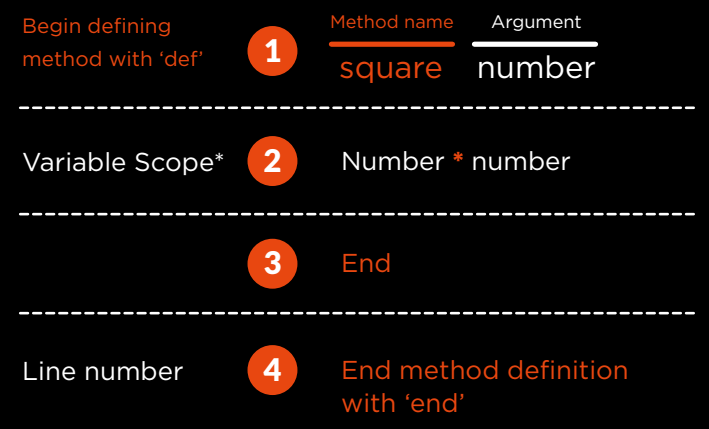
After the `def` goes the method name. This can be anything you like (try to make it relevant to what the method does). Ruby has a list of reserved words that you can't use.

You should try to give your methods clear, descriptive names. Ideally the method should read like a regular English sentence and, if pushed, someone should be able to pretty much guess what the method should do just by the name of the method and the name(s) of the argument(s) that can be passed to it.

If your method takes any arguments, they go in parentheses after the method_name, and are separated with commas.

Some methods take no arguments:

```
def say_happy_birthday
  puts "Happy Birthday!!"
end
```



*means that the variable is only accessible within this boundary. Use @number to break out the boundary!

While others take one argument:

```
def say_happy_birthday_to(name)
  puts "Happy Birthday #{name}"
end
```

And some can take two (or more) arguments:

```
def say_happy_birthday_to(name, repetitions)
  repetitions.times { |i| puts "Happy Birthday #{name}"}
end
```

Bonus

String Interpolation: #{name} is a convention used in Ruby that allows you to embed code in a String that Ruby will use to pass it information.

Challenges

In Ruby, by default, methods will return the last executed statement in the definition. So:

```
def yearly_salary(monthly_salary)
  monthly_salary * 12
end
```

will return the value of monthly_salary multiplied by 12.
You can assign the output of a method to a variable too, so:

```
danas_wishful_yearly_salary = yearly_salary(10,000)
```

Bonus

Pass no more than four parameters into a method

Sandi Metz
School Leaver

Read here

Will set the variable danas_wishful_yearly_salary to the integer 120,000 since we set the method yearly_salary(10,000) to return 10,000 * 12, which is 120,000.

Challenge 4

Can you write out a method that will let you say happy birthday to someone 3 times?

Challenge 5

Write a method that will check whether the number given to it as an argument is positive or negative, and display a message with the answer?

Hint: whenever you see this symbol, try out the code for yourself in IRB

Challenge 6

Could you try to extend your code to accommodate 0, which is neither positive or negative.

Arrays

You can think of an array as a list - a list that's defined in Ruby using the square brackets [ ], and a list which can contain any number of objects, of any type.

You can have an empty array:

```
[ ]
```

Or an array with some numbers in it:

```
[1, 2, 3, 4, 5]
```

You can even have an array with multiple data types. This one has an integer, 2 Strings and a floating point number:

```
[ 1, 'two', 3.0, 'four and five']
```

You can even have what's called a multi-dimensional array... arrays within arrays!

```
[ 1, [2, 3], [4,5] ]
```

You can access each item in an array using its numerical position. Type the following:

```
my_array = [1, 'two', 3]
my_array[1]
```

This will return the item in the array at position 1. What did it return? This is because arrays in Ruby start counting at 0. The first item in the array is accessed by executing my_array[0], the second item with my_array[1] and so on.

You can change an element simply by reassigning it:

```
my_array[1] = 2
```

? **Call my_array now. What do you see?**

```
my_array.length
```

If you're coding along, this will return 3. You can also add new elements to the end of an array:

```
my_array.push('four')
```

? **What will my_array be now? And what will its length be?**

Hashes

A good way to think of a hash is as an array where you access items not by their numerical position but by a unique key.

In an array, the 'key' for accessing a certain object is the number which describes that object's position in that array.

In a hash, you can decide which `key` is used to access a given value. For example, let's say you had an array of capital cities:

```
capital_cities = ["London", "Madrid", "Tokyo"]
```

You would be able to get the capital city of England by entering:

```
capital_cities[0]
```

But this isn't very expressive, is it? Nor is it very clear what we expect capital_cities[0] to return. Rather than the integer 0 being the key that returns the capital city of England, wouldn't it be great if 'England' was the key, and 'London' the return value?

That's what a hash is for.

An example hash can be created as follows:

```
capital_cities = { "England" => "London",
"Spain" => "Madrid", "Japan" => "Tokyo" }
```

This means that we can access the value "London" with the key "England" as follows:

```
capital_cities["England"]
```

See how similar it looks to an array, just with the integer replaced with a more expressive, unique key? We can then add a new capital city with the expression:

```
capital_cities["Hell"] = "Pandaemonium"
```

Challenge 7

Create a Hash with your favourite fruits & their colours. Here's one I've started:

```
fruits = {"banana" => "yellow"}
```

How many more can you add?

Iterations

When you write a script that repeats itself somehow, it is called iteration.

There are many ways to do iteration. Here are a couple of examples:

```
4.times do
  puts “Happy Birthday to you”
end
```

This should print “Happy Birthday to you” to the screen 4 times. The part between the do and end is called a block. If you want, you can replace the do and end with curly braces - they are synonymous:

```
4.times { puts “Happy Birthday to you” }
```

The convention in Ruby is that if the contents of the block fit on one line, use curly braces. If the contents of the block require more than one line, use do and end. But this is a little repetitive, isn’t it? You can do more interesting iterations by creating a local variable inside the block:

```
[1, 2, 3].each { |x| puts “I am on iteration #{x}” }
```

The local variable x is defined inside pipes, which you'll find by pressing shift + backslash on your Mac

On the first iteration, Ruby will take the first item in the array, the integer 1, and assign it to the local variable x. It will then print out **'I am on iteration x'**, which will literally display **"I am on iteration 1"**, since the local variable x has been assigned to the first value in the array on this iteration, which is 1.

On the second iteration, Ruby will take the second item in the array, the integer 2, and assign it to the local variable x. It will then print out **'I am on iteration x'**, which will literally display **"I am on iteration 2"**.

On the third iteration, Ruby will take the third item in the array, the integer 3, and assign it to the local variable x. It will then print out **'I am on iteration x'**, which will literally display **"I am on iteration 3"**.

Make sense?

What will happen when you run the following code:

```
["went to market", "stayed home", "had roast beef"].each do |x|
  puts "this little piggy #{x}"
end
```

Challenge 8

Can you complete the code below to complete the rhyme?

```
default = "clap your hands"
special = "and you really want to show it"
  #insert code here
end
```

Making sense yet?

Challenge 9

Create an array of numbers and see if you can iterate over them, displaying, on each occasion, the number multiplied by 2?

You should know that you can iterate over a hash as well as an array. Let's go back to our old example:

```
capital_cities = { "England" => "London", "Spain" =>
"Madrid", "Japan" =>"Tokyo" }
```

You could print these out in a nice format by doing:

```
capital_cities.each { |country, city| puts "#{country}'s
capital is #{city}"}
```

Have you tried it? What does it do?

On the first iteration, the key "England" is assigned to the local variable "country", and the value "London" is assigned to the local variable "city". (There's nothing clever going on here - the computer doesn't recognise them as countries and cities).

If the local variables were x and y, it would assign **"England"** to x and **"London"** to y - it's simply going through the key/value pairs in order and assigning them to the local variables that you specify, whatever they may be.

Ruby will then print the statement **"#{country}'s capital is #{city}"**, with country replaced by **"England"** and city replaced by **"London"**.

On the second iteration, the key **"Spain"** is assigned to the first local variable, which is **"country"**, and the value **"Madrid"** is assigned to the second local variable which, for the purposes of readability, we have defined as **"city"**.

Ruby will then print the statement **"#{country}'s capital is #{city}"**, with country replaced by **"Spain"** and city replaced by **"Madrid"**. And so on...

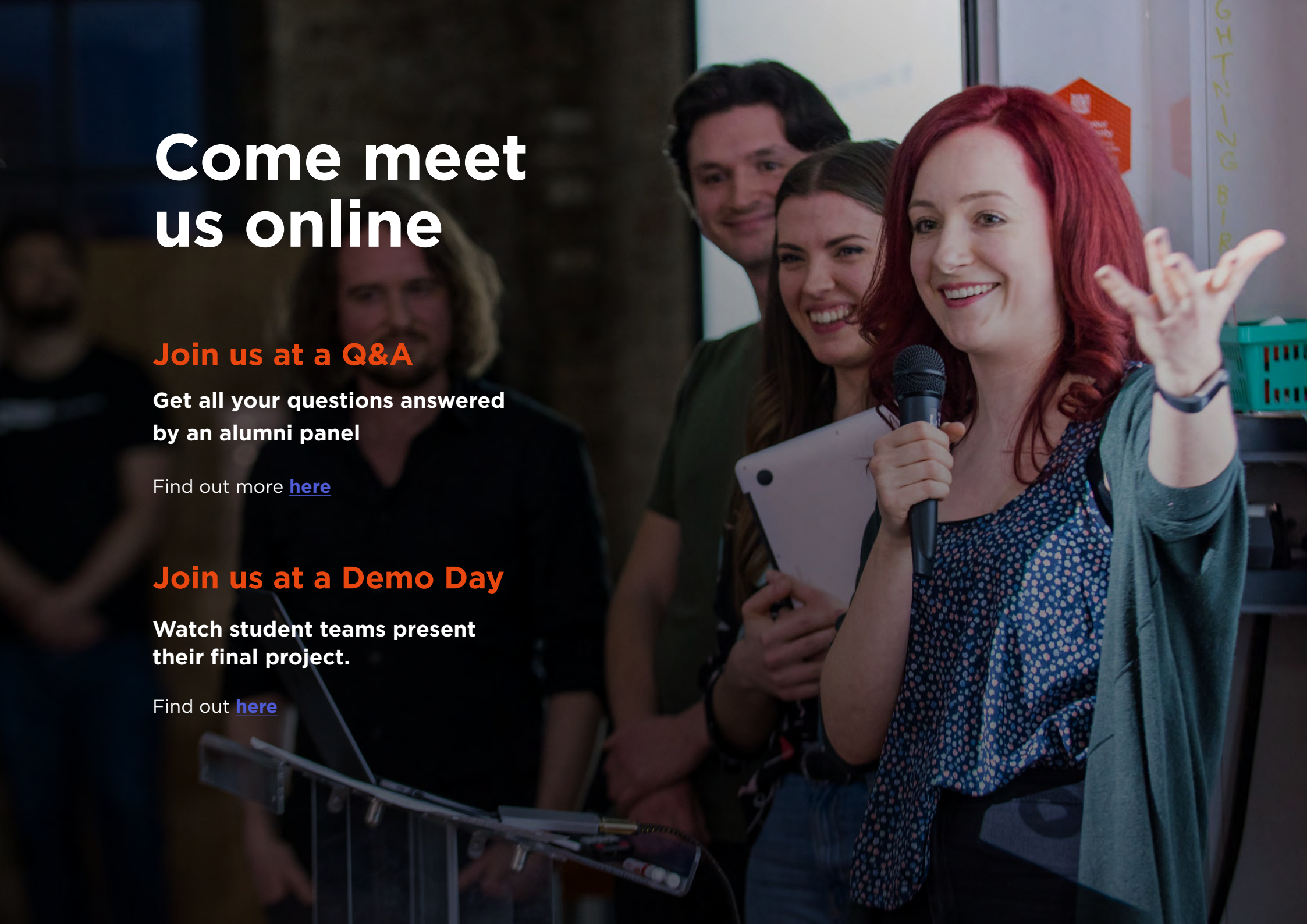
Try this on your computer. Add to the hash and see if you can list 15 countries + capitals in a bit of code?

Extra Challenges

*Bonus

1. Write a method that will tell you if a number is odd or even.
2. Write a method that takes one argument and returns the square of that number.
3. Write a method called ‘shout’ that takes a String as an input and returns that String in capital letters.
4. Write a ‘greeter’ method that takes a name as an input such that I could write “greeter(“Dana”) and it would display “Hello Dana! How are you today?”
5. Iterate over an array of numbers to display the square of each number in the array
6. Iterate over an array of numbers and only display even numbers
7. Create a hash containing your 5 best friends, with each person’s name as the key and their age as the value. Iterate over that array to display 5 examples that look like:

> “Dana is 28 years old”



Come meet us online

Join us at a Q&A

Get all your questions answered by an alumni panel

Find out more here

Join us at a Demo Day

Watch student teams present their final project.

Find out here

Contact

50-52 Commercial St
London E16LT
United Kingdom

contact@makers.tech

Follow us

@MakersAcademy

@MakersAcademy

@MakersAcademy

MAKERS