

# **Ruby**

# **For Beginners**

**Everything you need  
to know to get started**



**RubyGuides**

---

# Table of Contents

Introduction	1.1
Introduction	1.2
Variables	1.3
Strings	1.4
Conditions	1.5
Arrays	1.6
Loops	1.7
Hashes	1.8
Dealing With Errors	1.9
Object-Oriented Programming	1.10
What's Next?	1.11



## Introduction

Hey,

Thanks a lot for picking up this book! You are going to love it!

It will take some time for the concepts in this book to sink in, especially if you are new to this programming thing, but with patience, persistence & commitment you can do anything you desire.

Let's start with an overview of the Ruby ecosystem & then we are going to see how to setup your computer for Ruby development.

## The Ruby Ecosystem

Ruby was created about 20 years ago by Yukihiro Matsumoto, also known as "matz" in the Ruby community.

Ruby is an Object-Oriented language, you will learn exactly what that means in the Object-Oriented Programming chapter of this book.

In Ruby we share code in packages that we call "gems".

You can install a gem in your system using the `gem install` command & instantly have access to a program someone else has created.

Isn't that cool?

You don't need to use any gems to learn the basics of Ruby programming, so it's not something you need to learn more about yet.

Now your first step will be to setup your computer for Ruby development.

## Setting Up Your Environment

If you are using Windows you want to go to this site to download Ruby:

<https://rubyinstaller.org/downloads/>

You want the first file, which at the time of this writing is `Ruby 2.4.2-2 (x64)` .

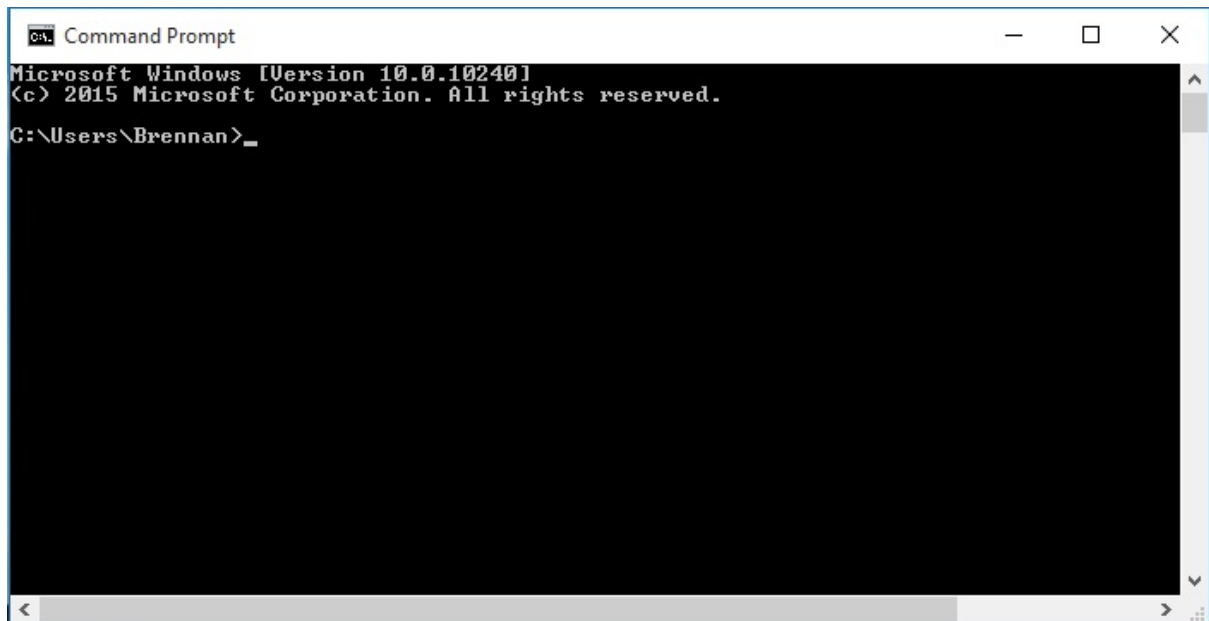
Just download & install it like any other program. If no errors appear then you have Ruby installed on your computer!

Now to start writing your Ruby programs you will need to open a terminal.

To do that open the windows menu & type `cmd.exe` .

Then press enter.

It will look something like this:



```
Command Prompt
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.
C:\Users\Brennan>
```

The next step is to type inside this terminal window.

You want to type this: `irb` .

This will launch a Ruby interpreter.

**It should say something like:**

```
irb(main):001:0>
```

Now you are ready to start typing Ruby code!

## Linux & Mac Setup

If you are on Linux or MacOS then you probably already have Ruby installed. You can confirm this by opening a terminal (search for "terminal" in your menu), then typing `ruby -v` .

This should print your Ruby version.

**Like this:**

```
ruby 2.4.1 (2017-03-22 revision 58053) [i686-linux]
```

If you don't get a Ruby version then [refer to this site](#) for more details on how to install Ruby for your particular Linux version or Mac.

## Your First Ruby Program

Let's start with something super simple. Ruby understands numbers & basic arithmetic operations.

**Try this:**

```
5 * 5
```

**This will result in:**

```
=> 25
```

If this doesn't work make sure to type this inside the black terminal window that you opened before. Also make sure you have `irb` open, it needs to say something like `irb(main):001:0>` .

Now try some other math operations just to get used to this terminal thing!

## Saving Your Programs Into Files

If you would like to save your programs you can use a code editor like [Atom](#).

It's a bit harder to run your program from a file.

You will have to find the full path to your file (available on Atom's bottom-left section), copy it & then from a terminal window run `ruby` + a space + the full path to your Ruby file.

**For example:**

```
ruby C:\Users\Jesus\test.rb
```

Notice that by convention we end Ruby files with the extension `.rb` .

Now you know how to save your program into files & run them. But for learning you are going to spend most of your time under the interactive interpreter `irb` , because it's faster to type in & see the results directly.

Oh, and one more thing!

If you try these math operations from a file you may notice that nothing prints on the screen. That's because you have to add `puts` in front of the operation.

**Like this:**

```
puts 5 * 5
```

The interactive interpreter `irb` , already does this for you, but if you run your program from a file you will need to use `puts` to see the results on the screen.

## Summary

In this chapter you learned about Ruby's history & ecosystem. You also learned how to setup your computer for Ruby development & how to run Ruby programs (from a file or using the `irb` interpreter).

## How do you work with data?

In Ruby, and most programming languages we use something called "variables". Like the name implies variables can change & we use them as name for things.

So if we have a number like `20`, it could have any meaning.

But we can use a variable to give it meaning.

### Example:

```
age = 20
```

Now we know that this `20` represents an age! This also allows us to reference the number `20` by name.

Notice what have done here.

This is an "assignment". We have told Ruby to assign the variable `age` to the value `20`.

We can tell Ruby to print the value of `age` like this:

```
puts age
```

This will print `20` on the screen.

You can treat this `age` just the same as the `20`, so you can do math with it.

### Example:

```
puts age * 2
```

This will print `40`.

### Another example:

```
puts age * age
```

Will print `400`.

Try all of this yourself so you can get familiar with using variables!

## How do you work with text?

A string is a sequence of characters.

Strings are how we work with text in most programming languages.

**Here's what a string looks like:**

```
"hello"
```

This is a string containing the word "hello".

Notice the quotation marks ( " " ) before & after the word. These act as delimiters for when a string is starting & ending.

They are very important so don't forget them!

A string can have anything inside, including numbers, symbols & special characters (like `\n`, the newline character).

Everything inside a string is just part of the string.

Meaning that if you have a string which is just composed of numbers, like this one:

```
"123"
```

It's still a string. In other words, it won't behave like numbers.

So if you try to add up two strings:

```
"123" + "123"
```

You will end up with a new string:

```
"123123"
```

But if you had real numbers (without quotation marks) then they would add up as you would expect:

```
123 + 123
```

Results in:

```
246
```

The quotation marks make all the difference!

## Can I use single quotes?

Yes, you can!

This is also a valid string:

```
'hello'
```



But you should stick with double quotes ( " ) for now.

## Basic String Operations

So what kind of things can you do with strings besides storing data?

Well you can call methods on them, just like any other object. Remember that in Ruby most things you can interact with are objects & **objects have methods you can use to ask an object to do something**.

How would you know what methods are available on `String` objects?

One way to find out is to use the Ruby documentation.

**So try this now, go to this page:**

<http://ruby-doc.org/core-2.4.0/String.html>

And of the left you will find a column title "Methods".

That's the list of methods available to you!

Don't worry if it seems overwhelming, you don't need to know all of them, at least not now :)

I will tell you which methods are important right now.

**You should be familiar with the following methods:**

Method	Description
<code>gsub</code>	Replace part of the string with something else
<code>split</code>	Split the string into an array of smaller strings. Default separator is a space.
<code>upcase</code>	Convert all the letters to UPPERCASE letters
<code>downcase</code>	Convert all the letters to lowercase letters
<code>include?</code>	Returns true if a string contains another other string
<code>start_with?</code>	Returns true if a string starts with another string
<code>chars</code>	Converts a string into an array of its characters
<code>to_i</code>	Converts a string containing numbers into an actual number ( <code>to_s</code> is the reverse operation)

Please notice that none of these methods will actually change your strings.

They return a new, updated string, so keep that in mind.

**You can see this in action:**

```
sentence = "We have many dogs"
sentence.gsub("dogs", "cats")
```

If you didn't know this you would expect that `sentence` now contains the string `"we have many cats"` .

But that's not the case.

So what do you do if you want to "save" the changes?

Well the simplest option is to re-assign the variable.

**Like this:**

```
sentence = sentence.gsub("dogs", "cats")
```

Pay attention here because this is a small change. All I did is add `sentence =` in front of our call to `gsub`.

Now the value of `sentence` has changed & it contains the new value `"We have many cats"` instead of `"We have many dogs"`.

## How do you mix variables & strings?

Sometimes you may have a value saved in a variable & you want to form a message with this value plus some text.

The solution to this is called "string interpolation".

**Here's an example:**

```
name = "Peter"
puts "Hello #{name}"
```

This will print `"Hello Peter"` on the screen. The key here is this funny-looking thing: `#{}`.

That's how you interpolate, you put this symbol `#{}` & the variable name inside that and Ruby will take care of the rest for you.

Ruby will even go as far as converting numbers into strings for you when the variable is a number.

If it didn't do that you would have to use the `to_s` method on your number to convert it into a string.

## How do you work with individual characters?

Another common operation is to work with the individual characters of a string.

One way to do this is to use indexing, just like if you were working with an `Array`.

**Example:**

```
animal = "Cat"
puts animal[0]
```

This will print the letter `"C"`, `animal[1]` would print the letter `"a"`, and `animal[2]` would print the letter `"t"`.

Another way to work with individual characters is to use the `chars` method.

**Example:**

```
name = "David"
characters = name.chars
```

The value for `characters` will be:

```
["D", "a", "v", "i", "d"]
```

So now you have an actual array of the characters & can use `Array` methods like `each`, which was not possible before.

If you want to put this array back into a string you can use the `join` method.

**Example:**

```
characters = ["D", "a", "v", "i", "d"]  
  
characters.join
```

Remember that this doesn't literally convert the `characters` variable into a string. It simply returns a new string.

## What can you do when your data contains quotation characters?

Since quotation marks are used in regular text you will have to deal with this at some point.

**Here's an example:**

```
"John's brother is 30 years old"
```

In this case this will work just fine, because the single quote doesn't get in the way of the double quotes that we are using for our string.

But if you need to handle double quotes inside double quotes...

```
"He was \"interested\", but I think he should be committed."
```

Notice the `\` character, that's how we "escape" the quotation marks. We are telling Ruby this is data & not a string delimiter.

## What happens when you multiply a string?

This is a nice little trick you can use whenever you need the same string repeated many times.

**Example:**

```
puts "A" * 30
```

Will print:

```
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

## Summary

You learned that strings are a sequence of characters, that strings are also objects so you can use methods on them & that you can lookup what methods are available on the Ruby documentation.

You also learned how to mix together variables & strings by using string interpolation.

Something really important to remember, most Ruby methods don't change your object! What this means is that if you call `upcase` **your string will remain the same**, but you will get a **new string** that has changed.

So remember to save the new string in a variable if you want to use it later :)

## How do you make decisions in Ruby?

Very often you will want your program to be able to make decisions.

Decisions based on some data or condition.

### Examples:

- "if the room is too cold turn on the heater"
- "if this customer has been with us for more than 3 years then send him a thank you gift"
- "if we don't have enough stock of this product then send an order to buy more & tell the customer he will have to wait a little longer"

Things like that is what we mean by taking decisions. If something is true (the condition) then do something.

In Ruby the way we do this is to use an `if` statement.

### Like this:

```
if stock < 1
  puts "Sorry we are out of stock!"
end
```

We can also say "if this is NOT true then do this other thing":

```
if stock < 1
  puts "Sorry we are out of stock!"
else
  puts "Thanks for your order!"
  stock -= 1
end
```

Notice the syntax. It's important to get it right.

The `stock < 1` part is what we call a "condition". This is what needs to be true in order for the code inside the condition to work.

In this example we are using the "less than" symbol `<`, but there are other symbols you can use for different meanings.

### Here's a table:

Symbol	Meaning
<code>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>==</code>	Equals
<code>!=</code>	Not equals
<code>&gt;=</code>	Greater OR equal to
<code>&lt;=</code>	Less OR equal to

Notice that we use two equal `==` symbols to mean equality!

One equals sign `=` in Ruby & most programming languages means "assignment" so make sure to use `==` when you want to find out if two things are the same.

## Multiple Conditions

Sometimes it's not just as simple as `if / else`.

And what you need is to check for different conditions at the same times.

For example, you are trying to figure out if a person's age is in a certain range & different ranges mean different things in your application.

The solution, in this case, is to use a `case` statement.

**It looks like this:**

```
age = 31

case age
when 1..18
  puts "You are too young."
when 19..40
  puts "How are you doing today?"
when 41..99
  puts "We have the perfect product for you!"
end
```

In this example we are using ranges (like `19..40`) to check for the value of the `age` variable. If the age doesn't match any of those ranges then nothing will happen.

An alternative is to use the `elsif` statement.

**Example:**

```
if age > 1 && age <= 18
  puts "You are too young to drink."
elsif age >= 19 && age <= 40
  puts "How are you doing today?"
elsif age >= 41 && age <= 99
  puts "We have the perfect product for you!"
end
```

As you can see this a bit harder to read, so for ranges I would always use a `case` statement.

## Logic Operators

In our last example you will also notice this `&&` symbol, which we haven't covered before.

This means that both conditions need to be true for the whole thing to be true.

**So if we have:**

```
if name == "David" && country == "UK"
end
```

Both `name` AND `country` need to match for the whole condition to be true.

We also have the `||` operator.

**Example:**

```
if age == 10 || age == 20
```

```
end
```

Then `age` can be `10` OR `20` for the condition to be true.

Notice how these two operators ( `&&` , `||` ) allow you to combine conditions, but they need to be proper conditions.

In other words, you CANT do this:

```
if age == 10 || 20
end
```

This is not valid.

You need a full condition on each side ( `age == 10 || age == 20` ).

## Things to Watch Out For

Just before we end this lesson & want to mention a few problems you may run into & what to do about them.

The first is about comparing strings.

When comparing two strings they must look exactly the same!

Including the "casing".

So "hello" & "Hello" would be different words according to Ruby (and other languages).

The way to solve this is to make sure you "normalize" both sides in some way.

**Example:**

```
name = "David"
expected_name = "david"

if expected_name == name.downcase
  puts "Name is correct!"
end
```

The key here is the `downcase` method on `name` , this way we can make sure it will always be downcased so we can avoid the problem.

Another problem you may come across with related to arrays is "special symbols". These symbols are for things like new lines

`\n` & the tab key `\t` .

So the problem comes if you try to compare two strings that look the same, but they will not match if one of them has one of these special symbols.

To see them you will need to use the `p` method:

```
name = gets
p name
```

Try this code, type something in, and you will notice that `name` contains the newline character (which is not normally visible with `puts` ).

To remove this character you can use the `chomp` method.

```
name = gets.chomp
```

```
p name
```

Now the newline character ( `\n` ) is gone!

## Summary

In this chapter you learned about conditions. Conditions allow you to take decisions in your code, this is what makes your program "think".

You also learned how to use the `if` statement & the `case` statement to handle different situations where you need to make decisions.

Finally, you learned about a few things to watch out for, like string "casing" & special symbols.

## How do I save multiple things into the same place?

More often than not you will want to work with more than one number, one string, one user account...

...the way programming languages deal with this is with arrays.

Arrays are sequential data structures which can hold any kind of data: numbers, strings & even other arrays!

**Here is an array:**

```
animals = ['tiger', 'gorilla', 'cow']
```

To access any elements contained inside the array you need the array index. The array index is simply the position of the element inside the array.

Arrays are zero-indexed, meaning that the first element starts at index 0.

**Example:**

```
animals[0]
```

**Returns:**

```
'tiger'
```

Because `'tiger'` is the first element of the array its index is 0.

Notice that this is **always** done using the square brackets `[]`, with a number inside. This number represents the index you want to access.

If we wanted to access the second element we would use index 1.

**Example:**

```
animals[1]
```

**Returns:**

```
'gorilla'
```

Remember that the index starts at 0, which can be a bit confusing at first, but you will get used to it!

## What happens if you use the wrong index?

If no value is present at a specific index you will receive a `nil` value back.

**Example:**

```
animals[5]
```

**Returns:**



```
nil
```

This `nil` value means nothing was found at this index.

### Here is the most important thing:

If you only knew this about arrays, then you would be able to use them.

But Ruby gives you a lot of methods to make things easier for you, if you don't know exactly what method to use it's ok to fall-back to fundamentals.

## What else can do you do with an array?

Array are also objects, just like strings, numbers, hashes, ranges, etc. And since they are objects they have methods (commands) you use with them.

Here are the most important `Array` methods:

- `push` / `<<` (they are the same)
- `each`
- `include?`

The `<<` method allows you to add new items to an existing array.

### Like this:

```
the_array = []

the_array << 1
the_array << 2

p the_array
```

### Results in:

```
[1, 2]
```

This is great to build a "results" array. This is **something we do in programming a lot. Build a bigger thing from smaller things.**

**Note:** Even if the bolded part sounds obvious to you, it's very important, so keep it in mind :)

The `each` method allows you go over every element of the array, one at a time, without changing the order.

Every element is feed into your "block", which you can think of like a machine that takes that element (like a word or number) & does something with it (like add it to a total or print it on the screen).

### Example:

```
animals.each { |animal| puts animal }
```

### Output:

```
tiger
gorilla
cow
```

**Important:**

Notice that when you use `each`, you don't need the Array index anymore. That's because Ruby already takes care of this for you!

In this example, `animal` has the value for the current array element.

Then we have the `include?` method. This method will tell you if something is inside the array.

**Example:**

```
animals.include? "tiger"
```

**Returns:**

```
true
```

## What is multi-dimensional arrays & how do you work with them?

Up until now we have worked with arrays with just one dimension. That's what you are going to be doing most of the time.

But sometimes you will come across with arrays with multiple dimensions or layers.

Here's an example of a multi-dimensional array:

```
values = [[1,2,3],[4,5,6],[7,8,9]]
```

Notice that this is just an array, with more arrays inside.

If you want to access the one value inside this kind of array you will need multiple indexes. So it's kind of like a grid with coordinates.

In this case if we want to access the first value of the first array.

**We would do this:**

```
values[0][0]
```

This will give you `1`.

The for the first value from the second sub-array.

**We would do this:**

```
values[1][0]
```

Which results on the number `4`, then `values[1][1]` will give your `5` & `values[1][2]` will give you `6`, etc.

## Summary

If you can master these three methods (`<<`, `each`, `include?`) & understand indexing then arrays shouldn't be too much of a problem for you!

The hardest part is figuring out the logic & strategy for your program. That's where using something like `irb` or `pry` is helpful.

## How do I write a loop?

Loops let you repeat one action many times.

This can be useful for many things, from counting, to going over the elements of an array, to just printing a message on the screen a number of times.

But how do you write a loop in Ruby?

Well there are a few ways to do this...

Let's start with the most fundamental way.

The `while` loop.

## Writing a while loop

The `while` loop is available in most programming languages so it's always useful to know. It's also the kind of loop that you can fall-back to when everything else fails.

And there are some situations when only a `while` loop would make sense. For example, if the stopping condition of the loop is not fixed.

**Here's a code example:**

```
n = 0
while n < 10
  puts n
  n += 1
end
```

What this will do is to print all the numbers from 0 to 9 (10 excluded).

Notice that there are some important components:

- The `n` variable
- The condition ( `n < 10` )
- The `n += 1`

All of these are critical for this to work.

The variable `n` holds the value we are using for counting, the condition tells Ruby when to stop this loop (when the value of `n` is greater or equal to `10`), and the `n += 1` advances the counter to make progress.

## The times loop

This is possibly the easiest loop you can come up with.

**Look at this code:**

```
10.times { puts "hello" }
```

What this does is to print the word `"hello"` 10 times.

As you can see there isn't much to it & it should be easy to remember.

But what if you want the number?

In the last example, with the `while` loop, we had access to this `n` so we could print it.

Well you can also do that with `times`.

**Example:**

```
10.times { |i| puts "hello #{i}" }
```

This will print `hello 1`, `hello 2`, `hello 3`, etc.

The key here is the little `|i|` thing, which by the way, can be anything. It doesn't have to be an `|i|`. It could be `|n|` or `|foo|`.

It's just a name!

If you are familiar with methods, this `|n|` is like a method parameter. In other words, it's just a variable that becomes the current value for each iteration of our `times` loop.

## The each loop

Now let's look at the last kind of loop & probably the one you are going to use the most often.

This loop requires you to have a collection of items, like an array, a range or a hash to be able to use it.

**Example:**

```
numbers = [1,3,5,7]
numbers.each { |n| puts n }
```

This will print all the numbers inside the `numbers` array.

Notice how we have this syntax again with the `|n|`.

In case you are not familiar with this, we call the whole thing after `each` a block.

This -> `{ |n| puts n }`.

A block is just a way to create a method without a name.

So what happens is that `each`, or any other method that takes a block, is able to use our name-less method.

And that's how `puts n` is run multiple times.

In fact, do you want to know something cool?

An `each` loop is just a `while` loop, but Ruby is managing the whole condition, variable & counting for you behind the scenes.

You know, these 3 components that I said were super important for a loop to work.

## Summary

You learned that a loop helps you repeat something a number of times. You also learned 3 different ways to write a loop: the `while` loop, the `times` loop & the `each` loop.

## How do you create a dictionary in Ruby?

A hash table allows you to create a dictionary-like structure in Ruby.

**It looks like this:**

```
hash = {}
```

Yes, this empty pair of "curly" brackets is what creates an empty hash.

**To add one item:**

```
hash["bacon"] = 123
```

What's going on here?

You may recognize the square bracket syntax from Arrays, where we use that syntax to ask for a value at that index.

The idea here is the same.

But instead of numbers we use strings (or symbols, but that's more advanced).

**Now we can use our hash to see what's inside:**

```
p hash
```

The result for this will be the following:

```
{"bacon"=>123}
```

So this tells us we have one **key** inside the hash, with one **value**.

**Note:** the variable `hash` is just an example, it doesn't need to be that way, you could use any name for your hash like `foo`, `potato` or whatever :)

It is also possible to create a hash with values already on it.

**Like this:**

```
foo = {"bacon"=>123, "coconut"=>345}
```

Then you can access one of the values using its key ("bacon" or "coconut" in this case). So the key is like the word in a dictionary & the value is like the definition for that word.

**Example:**

```
puts foo["coconut"]
```

This results in `345` printed on the screen.

Ruby is pretty flexible, so it allows you to use anything as your value, from text (strings), to numbers, arrays, or even another hash!

## How do I get a list of all the keys?

## How do I get a list of all the keys?

If you ever need a list of all the keys, you can use the `keys` method.

```
data = {"bacon"=>123, "coconut"=>345}

data.keys
```

This will result in:

```
["bacon", "coconut"]
```

## How do I loop over all the elements of a hash?

You can do it like an array, using the `each` method.

There is only one difference.

Look at this example:

```
data = {"bacon"=>123, "coconut"=>345}

data.each { |key, value| puts "The key is #{key} & the value is #{value}" }
```

The difference is in the arguments: `|key, value|` .

In an array you just get one thing (the `value` ), but with a hash you have two things, the `key` & the `value` .

## How do I check if a key is available?

Instead of getting the list of keys & check the array using the `include?` method there is a better way.

The `key?` method.

Example:

```
data = {"bacon"=>123, "coconut"=>345}

data.key?("onion")
```

This will output `false` since `"onion"` is not a key inside the hash.

## The Fetch Method

Let me show you about the `fetch` & `merge` methods.

With `fetch` you can access a key in a different way.

Example:

```
hash.fetch("coconut")
```

What's the difference between this & the `[]` way?

The difference is what happens when the key is not in the hash.

With the square brackets `[]` you will get a `nil`. But with `fetch` you will get an error message.

```
hash.fetch("onion")
```

Will result in:

```
KeyError: key not found: "onion"
```

This is good because `nil` values are often a source of errors.

## Should you always use `fetch` to access a hash?

I think you should stick with `[]` for now, but keep `fetch` in mind :)

## Practical Uses for Hashes

So what are some uses for hashes?

The most obvious is to use them as a literal dictionary.

### Example:

```
country_codes = {  
  "ES" => "Spain",  
  "US" => "United States",  
  "FR" => "France"  
}
```

Here we have a mapping of country codes to their country name.

### We can query it like this:

```
puts country_codes["US"]
```

### This will output:

```
"United States"
```

Always use a hash when you have this kind of data mapping.

Another thing you can do with hashes is to use it as a "file cabinet".

For example, you can count letters or words, by making the letter the key & the count the value.

### Here's the code to do that:

```
count = Hash.new(0)  
"hello".each_char { |ch| count[ch] += 1 }
```

Notice something interesting here.

I used `Hash.new(0)`, instead of `{}`. This creates a hash where values will default to `0`, this is needed so we can add to it (the `count[ch] += 1` part).

## Summary

You have learned about hashes, a dictionary-like structure to help you work with data. You have also learned that they are composed of key / values pairs, and that you can access a value using a key & the `[]` syntax.

### You have also learned:

- A non-existent key will return a `nil` value by default.
- The `each` method is available, but you need two arguments ( `|key, value|` ).
- The `fetch` method will allow you to retrieve a value & raise an exception (error message) instead of returning `nil`.



## I got an error message! What should I do?

First of all, don't panic! It's completely normal & even very experienced developers get error messages all the time.

Second, realize that error messages are there to help you.

Most of the time the error message has information to help you fix the problem.

So what you need to do is to read the message & try to understand what it's telling you.

**Here's an example so we can look at it together:**

```
exception.rb:7:in `bar': undefined local variable or method `a' for main:Object (NameError)
  from exception.rb:3:in `foo'
  from exception.rb:10:in `<main>'
```

The most important line is usually the first one.

The first line will tell you what kind of error you are dealing with, where did this error happen & a few more details about the error.

Errors can be categorized into 2 major categories:

- Syntax errors
- Runtime errors

Depending on which category you are dealing with you are going to be looking for different things, so it's important to know what category you are working with.

## Syntax Errors

A syntax error happens when you didn't follow the rules of Ruby & the language interpreter can't understand something you wrote.

It's like a grammar mistake.

Programming languages have strict rules about how things have to be written, so if you (without noticing) break one of the syntax rules then you are going to get a syntax error.

How do you recognize a syntax error?

Because the error will clearly say it's one.

**Example:**

```
test.rb:2: syntax error, unexpected end-of-input
```

When you see one of these **don't try to change your program's logic**, because that's not the problem.

What you want to look for is typically one of the following:

### A missing closing or opening parenthesis.

Parenthesis in all their forms (regular `()`, curly `{}`, squared `[]`) always go in pairs. If a closing pair is missing you will get a syntax error.

## A missing closing or opening quotation symbol.

Like parenthesis, quotation symbols ( " & ' ) used for strings always go in pairs.

## A missing or extra end statement.

When you open a block using `do` or when declaring an `if` statement, method (with `def`) or `class` you need to make sure to have the right amount of `end` statements.

This is another thing that has to go in pairs!

Notice that you can also have **extra** parenthesis, `end` statements, or quotation symbols causing the syntax error that you may need to delete.

## Other syntax errors

These are not all, but they are the most common, so look for them first!

## Runtime Errors

This kind of error will happen because something in your program is wrong while it runs.

Notice that Ruby always checks for syntax first, so if you get a runtime error you don't have to worry about checking for the right amount of parenthesis or quotation symbols.

The most common runtime error is `NoMethodError`.

### Example:

```
undefined method `foo` for nil:NilClass (NoMethodError)
```

It happens because one of two reasons:

- You are trying to call a method on `nil`.
- You are trying to call a method that doesn't exist (probably a typo).

The first one happens when you are working with the result of another method or section of your code that produced a `nil`.

For example, let's say that you want to work with the `:bacon` key in a hash, but for some reason that key is not defined, so when you try to retrieve the value you will get a `nil` back.

### Example:

```
h = {}  
  
h[:bacon]  
  
# nil
```

Then later in your code, you try to use this value thinking it's a valid value, but it isn't so it results in an error.

Once you have identified this error your job is to find why the value is `nil` & not the value you expected...

Did you forget to initialize your hash?

Are you pulling this data from a file, but you are not reading it correctly?

The second reason (wrong method, but valid object) is usually fixed with a visit to the documentation to look for the correct spelling of the method.

Or maybe that method doesn't exist at all because you have the wrong class.

**Note:** Remember that you can use the `class` method on any object to find out what class you are working with.

So pay attention to the details & what the error message is telling you.

## Argument Errors

This is a subcategory of runtime errors & it happens when you have the right object, the right method name, but the wrong number of parameters (also known as arguments).

**It looks like this:**

```
ArgumentError: wrong number of arguments (0 for 1)
```

This means that you are calling the method using `0` arguments, but the method expects `1`.

**There is an easy fix:**

You have to add or remove arguments from your method call to match the expected amount.

## Uninitialized Constant Error

This one usually means a class is missing or that you have mistyped the name of a class. The reason is that class names are constants.

Constants always start with a capital letter, so it's easy to tell you are dealing with a constant.

**Example:**

```
uninitialized constant StringIO (NameError)
```

To solve this error you have to check for typos & make sure you have required the gems or libraries you need.

## Type Error

Another runtime error, this happens when you are trying to combine two incompatible objects, like a number & a string.

**Example:**

```
"abc" + 1
```

**Results in this error:**

```
TypeError: no implicit conversion of Fixnum into String
```

One way to solve this is to make both sides of the equation the same object type.

**Example:**

```
"abc" + 1.to_s
```

If your `TypeError` mentions `NilClass` then you have some `nil` value that you probably didn't expect. Remember that `nil` values can come from things like using the wrong index on an `Array` or using the wrong key on a `Hash` .

## Interpreter Specific Errors

This is a kind of problem that can only happen when working with an interactive interpreter like `irb` or `pry` .

The thing is that since these are interactive, they won't show you syntax errors right away.

So when you make a syntax error in `irb` you might get into a state where it looks like nothing is happening.

And your terminal may say something like:

```
irb(main):017:1>
```

or

```
irb(main):003:0"
```

Notice the `:1` on the first example & the quotation mark `"` at the end in the second example.

Those are indications that the interpreter is waiting for something.

The easiest solution is to press `CTRL + C` to get out of this state & try again!

## Summary

You have learned how to deal with errors, they are there to help you figure out what's wrong.

First you need to identify what kind of error you are dealing with, then look at the details, like the line number & file name.

Once the error is identified then you can narrow down the code that is producing the error & fix it.

Have patience, and don't forget to Google your error message if you can't figure it out!

## How do you create objects, classes & methods?

Object-Oriented Programming is a programming paradigm.

In other words, it's a way to think about how we write code.

In this paradigm we organize code in something we call a class.

A class is a container for:

- Data
- Behavior (actions)

Examples of a class include a `Book`, a `Car` & a `Library`.

But we don't want only one book or one library, we probably want many of them.

So that's why we have objects.

An object is created from a class & it will hold its own data.

In practice what that means is that we can have different books with different values associated with them (author, page count, title, isbn, genre...).

To help you figure out how to create the best classes there are some Object-Oriented Design principles.

**Here's a list:**

- Abstraction
- Encapsulation
- Polymorphism
- Cohesion & Coupling
- Inheritance & Composition
- Command/Query separation
- Design patterns
- Tell, don't ask
- SOLID

I have good news for you:

You don't have to worry about any of these right now! But it's good to start getting familiar with the vocabulary.

## Code Example

Now let's have a look at some example of Object-Oriented code in Ruby.

This is a `Book` class:

```
class Book
end
```

Notice that you can name your class anything you want, but it has to start with a capital letter & usually we want to use a noun instead of a verb.

This class here does nothing yet.

It has not methods & it has no data.

We can add a method (behavior) like this:

```
class Book
  def what_am_i
    puts "I'm a book!"
  end
end
```

Of course this example is a little silly, but it will help you learn how this works.

## Using Instance Methods

A method is just a section of code that we give a name to. It's useful because we can reuse this code multiple times by just using its name.

The name also works as a description of what this code is doing, so it's important to use good names for your methods.

How do we use this method?

**If you try this:**

```
Book.what_am_i
```

You will get an error!

Why?

Because methods by default need to be used from an object, not directly from the class.

**So to create an object & use the method:**

```
book = Book.new

book.what_am_i
# "I'm a book!"
```

The key here is the `new` method. This is what creates a `Book` object.

## Method Parameters

Sometimes you want to send some data into a method so it can work with that data.

Like if you are writing a calculator class, you probably want to have methods like `add` & `multiply`.

These methods will need some numbers to do their work & then return a result.

**Example:**

```
class Calculator
  def add(a, b)
    a + b
  end

  def multiply(a, b)
    a * b
  end
end
```

Notice this: `(a, b)` .

These are your parameters. Parameters can have any valid variable name, so `(foo, bar)` would also be valid.

You can have any number of parameters, but make sure to separate them using a comma!

Inside your method you can treat parameters as normal variables.

**Now, to call this method:**

```
calculator = Calculator.new
puts calculator.add(3, 4)
```

Which results in `7` :)

## How do you create parameters with default values?

You don't always want to force your method to require parameters. You can define your parameters in a way that makes them optional.

**Example:**

```
def greeting(name = "Jesus")
  puts "Hello #{name}, nice to meet you!"
end
```

Notice the `name = "Jesus"` , where my name `"Jesus"` is the default value.

So what this means is that if you use the method & give it a name then it will use that name.

But if you don't give it a name it will use the default name that you defined.

**Note:** If this `#{name}` thing caught you off-guard, this is called "string interpolation", review the Strings chapter for more details :)

## Storing Data

Remember how I said that classes are composed of two things?

Data & behavior.

Let's enhance our `Book` class so that it can store the book title.

```
class Book
  attr_reader :title

  def initialize(title)
    @title = title
  end

  def what_am_i
    puts "I'm a book!"
  end
end
```

Now we can create multiple books, each with a different title:

```
book1 = Book.new("Ruby Deep Dive")
book2 = Book.new("BOBA")
```

And we can read the title like this:

```
book1.title

# "Ruby Deep Dive"
```

Notice that this works because I added the `attr_reader :title` line to our class. This tells Ruby to create a `title` method for us that returns the stored value.

Oh and this `@title` thing?

We call that an "instance variable".

It's a kind of variable that is associated with a particular object.

## Class Methods

We have seen instance methods & instance variables, both work with an instance of a class (which is the same as an object).

But what if you want to define a method that works directly with the class & requires no extra objects to be created?

Well that's possible.

And we call that "class methods" or "singleton methods".

**Here's how to define a class method:**

```
class Book
  def self.method_name
    puts 123
  end
end
```

What makes this a class method?

The "self" keyword before the method name.

When we use "self" we are referring to the current object, which in the context of a class definition refers to the class itself ( `Book` in this example).

**So we are saying:**

```
"Define a method for this class, instead of a method for instances (objects) of this class."
```

Here's how to use a class method:

```
Book.method_name

# 123
```

When to use class methods?

Only when this is some standalone "helper" method. Most of the time you want to use regular instance methods.

If you find yourself using a lot of class methods you might have a class design issue.



## Summary

You learned about Object-Oriented Programming (OOP), what is a class & why we need object. You also learned how to create a class in Ruby that has both data & behavior.

### Important points to remember:

- OOP is a way to think about code & how to organize it using classes & methods.
- A method is the part of a class that does things.
- We store data in instance variables ( `@example` ).
- To access instance variables outside the class own methods you need to define an `attr_reader` .
- If you want to use an instance method you need to create an object of that class first (using `.new` )

### Exercise:

Create a class named `cat` that has a `color` attribute. It also has a "speak" method which prints "meow" on the screen.

Use this class to create 3 `cat` objects with different colors.

## What's next?

Congratulations on finishing the book!

Did you know that most people that buy a book never finish reading it?

Isn't that crazy?

But you did it! So give yourself your favorite healthy reward (like a high % cocoa chocolate bar!).

There is still plenty of work to do, you are just getting started.

First I want you to read this book again a few times until everything sinks in. Repetition is very important when developing new skills.

Then I want you to see what kinds of programs can you come up on your own just for fun. I know you have big projects in mind, but keep it simple for now :)

When you are feel ready to take the next step you will want to start reading on the following topics:

- Modules
- Enumerable methods
- Inheritance & Composition
- Regular expressions
- TDD (Test-Driven Development)
- Sinatra

You will also want to join my ongoing Ruby education program here:

<https://www.rubyguides.com/ruby-guides-pro/>

And grab a copy of the next book in the series:

<https://www.rubyguides.com/ruby-book/>

Thanks for reading!

- Jesus Castello