PROGRAMMING FOR BEGINNERS

LEARN TO CODE BY MAKING LITTLE GAMES

TOM DALLING

Programming for Beginners

Learn to Code by Making Little Games

Tom Dalling

This book is for sale at http://leanpub.com/programming-for-beginners

This version was published on 2020-03-11

Contents

Introduction 1
- What's in This Book 1
- What to Do When You Get Stuck 2

Level 1: Hello, World! 4
- Install Ruby 4
- Install a Text Editor 5
- Write the Code 6
- Download the Code Runner 6
- Run the Code 8
- Optional Further Exercises 8

Level 2: Input and Output 9
- Make a Project Folder 9
- Control Flow 9
- Strings 10
- Variables 11
- Variable Names 13
- Functions 14
- Comments 15
- Boss Project: Name the Ogre 17
- Optional Further Exercises 17

Introduction

So you want to learn how to write code. As of 2020, software developers are in demand, which makes software development quite a lucrative career. I also think it's a lot of fun. You type in some text, then the computer does what you say! And if you already own a computer then it's basically free, other than the time that you invest.

Programming is a creative endeavour. You can create whatever you want, and then interact with your creation. It's an eye-opening experience to make something that asks you questions, and responds to your answers. I hope you will have that experience very soon, as you start working through this book.

However, the learning curve can be very steep and frustrating. The majority of programming books and tutorials are made for people who already know the basics. They are too advanced for true beginners – people who have never written any code before – which makes them difficult to absorb if you are just beginning to learn.

You don't need to know anything about writing code, or making software. You will need some basic computer skills – like downloading, opening, and saving files – but everything else will be explained here, step by step, starting from the very beginning.

This book is designed to be your first step into the world of computer programming. It teaches the fundamentals – the core concepts that programmers have used for over 50 years. With a knowledge of the fundamentals, you will have the ability to learn the more advanced concepts that come next.

What's in This Book

You will learn the fundamentals of computer programming by:

1. looking at, and experimenting with, example code
2. reading explanations of the example code and programming concepts

3. creating your own small, text-based games

We will be using the Ruby programming language, but you will learn the essential concepts that are common to all programming languages – concepts such as variables, values, branching, looping, and functions.

This book is divided into levels. Each level introduces new programming concepts, demonstrated with example code. At the end of each level there is a *boss project* – a description of a small, text-based game that you must create. Each boss project is more complicated than the last, and requires you to apply everything that you have learned up to that point.

At the end of this book, there are resources to further your learning. You will also have access to the code for a small game with 2D graphics and audio, as a demonstration of what is possible with a little more study.

What to Do When You Get Stuck

Even though this book is designed for people who have never written code before, the coding challenges will be difficult and probably frustrating at times. This is totally normal. I have tried to remove as much frustration as possible in order to provide a gentle learning curve, but programming remains a complicated endeavor. If you enjoy complicated puzzles, then you are in for a treat!

Here are some tips that should help you when you get stuck:

- **Use the example code**. Every level contains code examples. These examples demonstrate things that you will need in order to complete the level. Try to guess what each example does, then run it to see if you were correct.
- **Run your code *often***. Make a tiny change, then run your code. Make another tiny change, and run the code again. This way, when you make a mistake, you will know exactly what caused it. If you write too much code without running it, you will find it harder to pinpoint errors.
- **Keep at it!** You might be able to solve the first few levels easily, but later levels will require more effort. There are levels that you won't be able to complete on your first attempt. When you get stuck, think about it, sleep on it, and try again tomorrow.

- **Reread previous levels**. Each boss project requires you to use *everything* that you've learned up to that point. The solution to your problem may have been explained in a previous level.
- **Double check your code *very* closely**. Even the *tiniest* mistake can cause the entire program to crash. In natural languages like English, it's OK if your spelling and grammar aren't perfect because other people will still understand you. Programming languages, however, are unforgiving – the computer expects perfection, and your code will not work correctly if there are any mistakes.
- **If in doubt, `puts` it out**. If you are not sure what the code is doing, try displaying values and variables with the `puts` function. This will reveal things that are otherwise invisible. If the output isn't what you expected then that means you have found a problem, and you can diagnose the problem by analysing the output.
- **Indent your code properly**. There is no debate on this topic – all programmers agree that indenting is necessary. Indenting rules exist to help you write code correctly. If you ignore the indenting rules, I guarantee that you will forget to write `end` somewhere and your program will stop working completely. Follow the indentation in the example code, and you will avoid many mistakes.

Remember that the harder the challenge is, the better you will feel when you conquer it.

Level 1: Hello, World!

In this level, we will set up and install everything necessary to make software using the Ruby programming language. To confirm that everything is working correctly, we will make a very simple program that displays the text "Hello, World!" Making this simple program is an old tradition in software development, and it marks the beginning of a new software project.

Install Ruby

The programming language that we will be using is called Ruby. In order to run code that is written in the Ruby language, we must first install Ruby.

If your computer is a Mac, then you already have Ruby. MacOS comes with Ruby already installed. Skip ahead to *Install A Text Editor*.

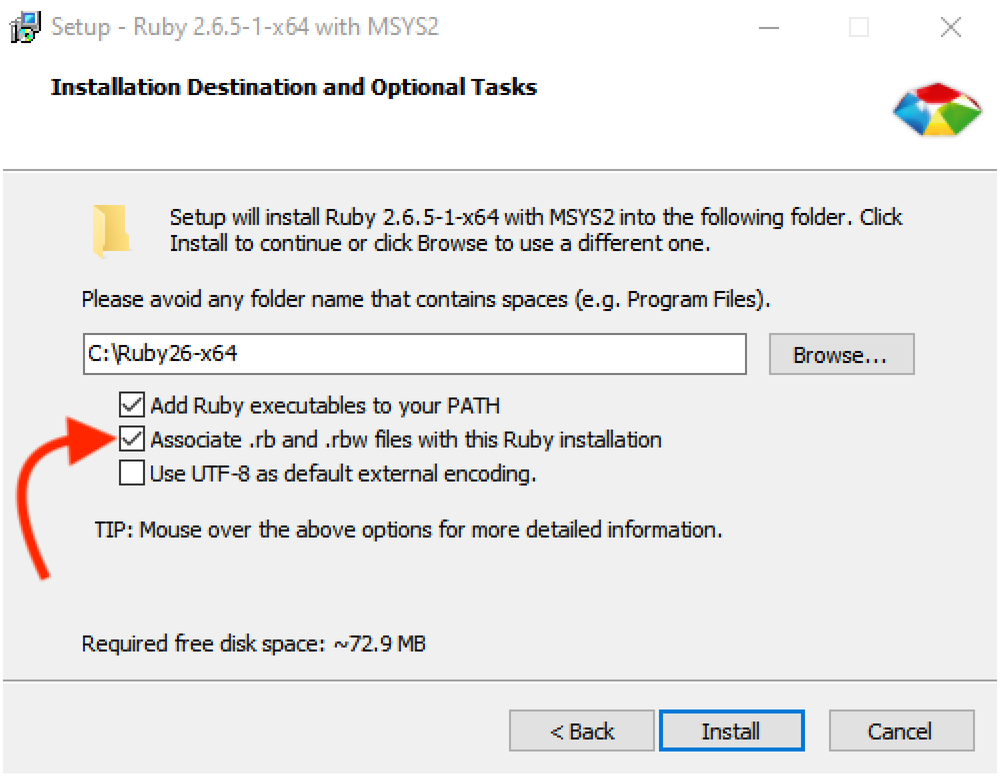
If you have a Windows computer, you will need to download and install Ruby from here: http://rubyinstaller.org/downloads/

As of early 2020, the latest version is `Ruby+Devkit 2.6.5-1 (x64)`. This book is compatible with any version of Ruby 2, so just pick the latest one.

If you have purchased your computer within the last five years you almost certainly want the (x64) variant. If your computer is older, you should check whether you have a 64-bit or 32-bit[1] version of Windows. If it is 64-bit, then download an installer marked as `(x64)`. Otherwise, download the 32-bit installer, which will be marked as `(x86)`.

During the installation, it will ask whether to "*Associate .rb and .rbw files with this Ruby installation.*" Make sure this checkbox is ticked, as it will allow you to run Ruby code files by double-clicking them.

[1]http://windows.microsoft.com/en-us/windows7/find-out-32-or-64-bit


Setup - Ruby 2.6.5-1-x64 with MSYS2

Installation Destination and Optional Tasks

Setup will install Ruby 2.6.5-1-x64 with MSYS2 into the following folder. Click Install to continue or click Browse to use a different one.

Please avoid any folder name that contains spaces (e.g. Program Files).

C:\Ruby26-x64

[✓] Add Ruby executables to your PATH
[✓] Associate .rb and .rbw files with this Ruby installation
[ ] Use UTF-8 as default external encoding.

TIP: Mouse over the above options for more detailed information.

Required free disk space: ~72.9 MB

Tick this option during installation

Install a Text Editor

Code is text, so it is written using text editing software. Text editors are slightly different to word processing software, like Microsoft Word or Apple Pages. Text editors don't have any formatting or styling options – no bold, italics, text alignment, page breaks, headers, footers, etc.

The text editors recommended below are free, and they have features that will assist in writing Ruby code. Please download and install one of the following text editors:

- Notepad++[2] (Windows only)

[2]https://notepad-plus-plus.org/download

- BBEdit[3] (Mac only)

If you are having trouble choosing between 32-bit and 64-bit installers, see the previous section of this chapter, *Install Ruby*.

Write the Code

Create a new folder called `RubyProjects` inside the `Documents` folder on your computer. Inside the `RubyProjects` folder, make another folder called `Level1`. This is where we will save the Ruby code for this level.

Now we can finally write our very first line of Ruby code! Open the text editor that you just installed, and make a new, empty file. Inside the empty file, write the following line of code:

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

Make sure to copy the code *exactly*, because even the tiniest difference can stop the code from working.

This code will be explained in the next few levels, but for now just know that `puts` is part of the Ruby language that tells the computer to display some text, and `"Hello, World!"` is the text to display.

Save this file with the filename "main.rb" inside the `Level1` folder. The filename must also be copied exactly, or else the code will not run.

Download the Code Runner

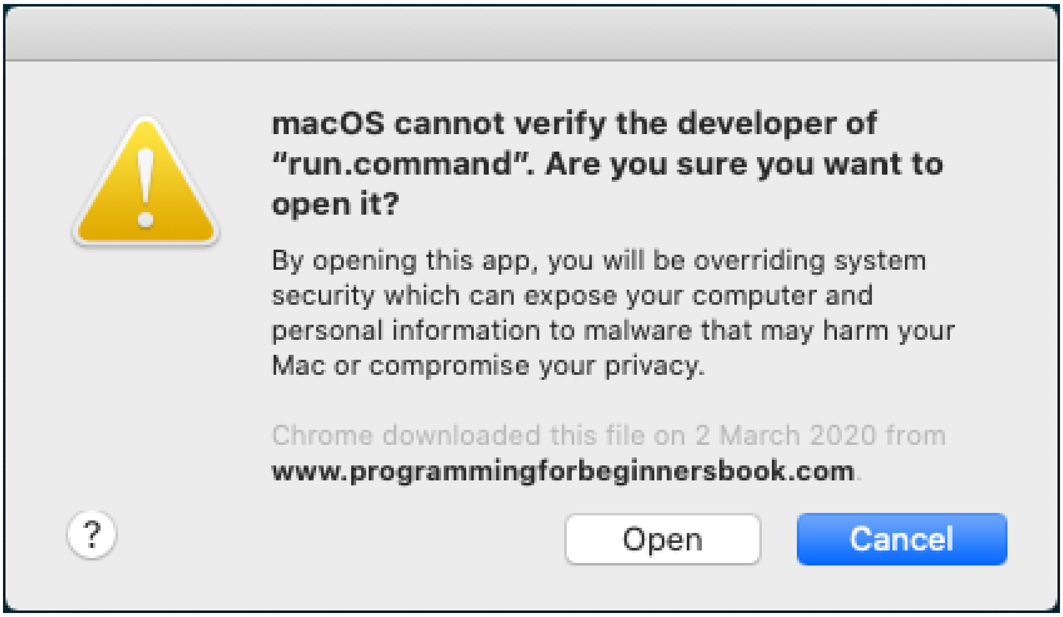
Code is like a set of instructions that a computer can understand. Now we must ask the computer to read the instructions out of the `main.rb` file, and actually perform them. This process of reading and performing has a few different names. It is most commonly referred to as "running" the code, "executing" the code, or sometimes "evaluating" the code.

[3] http://www.barebones.com/products/bbedit/

Running code can be a little complicated at first, so I have created a *code runner* file to simplify the process.

- **For Windows**: Download and unzip windows_code_runner.zip[4] to get `run.rb`.
- **For Mac**: Download and unzip osx_code_runner.zip[5] to get `run.command`.

  There is one extra step for Mac. The first time you use `run.command`, **control-click (or right-click) it** and select the "Open" option. It will ask if you're sure that you want to open the file, then click the "Open" button. This must be a right-click or control-click, otherwise the "Open" button will not appear. After doing this once, you will be able to double-click the code runner like normal.



macOS cannot verify the developer of "run.command". Are you sure you want to open it?

By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

Chrome downloaded this file on 2 March 2020 from **www.programmingforbeginnersbook.com**.

MacOS confirmation dialog

Move this code runner file (`run.rb` on Windows, or `run.command` on Mac) into the `Level1` folder where the `main.rb` file is.

In future, to make a new Ruby project:

1. Create a new folder.

[4]http://www.programmingforbeginnersbook.com/downloads/windows_code_runner.zip
[5]http://www.programmingforbeginnersbook.com/downloads/osx_code_runner.zip

2. Copy the code runner file into the folder.
3. Save the code in a file named `main.rb` or `game.rb`, inside the same folder.
4. Double click the code runner file to run the code.

Run the Code

Now is the moment of truth. Double-click the code runner file, and it should run the code inside `main.rb`. A window should pop up containing the following text:

```
>>>>>>>>>> Running: main.rb >>>>>>>>>>>>>>>
Hello, World!
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

If you see this text, then congratulations! You have written and run your first line of Ruby code. This proves that everything is installed and working correctly on your computer. Now we can start writing some little games.

Optional Further Exercises

- Try changing the `"Hello, World!"` text inside the `main.rb` file, and running the code again. Does the code still run successfully, or does it cause an error?
- Are the quotation marks necessary? What happens if you remove them?

Level 2: Input and Output

In this level, we will make our first bit of interactive software – code that can respond to user input. To achieve this, we will need to learn a little about control flow, variables, strings, and functions.

Make a Project Folder

Just like in *Level 1*, go to your `RubyProjects` folder and make a new folder called `Level2`. Copy the code runner file (`run.rb` on Windows, or `run.command` on Mac) from the previous level into the new folder, and save an empty file called `game.rb` into the new folder using your text editor.

While reading this level, put the code examples into `game.rb` and run them to see what happens. Typing in the code manually will help you to learn the Ruby language, but you can also copy and paste the code if you wish.

Control Flow

Sets of instructions usually have an order. For example:

1. Crack the eggs.
2. Whisk the eggs.
3. Fry the eggs.
4. Eat the eggs.

The same is true of code. As an example, try running this code:

```ruby
puts("first")
puts("second")
puts("third")
```

The output will be:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
first
second
third
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Code is run in a very specific order, and this ordering is called *control flow*. This is the first example of control flow, where each line of code is being run sequentially from top to bottom. Try changing the order of the lines to see how it affects the output.

Strings

In the early levels, we will be working with text a lot. In almost all programming languages, bits of text are called *strings*. Strings get their name from the fact that they represent a sequence of characters strung together. For example, the string `"cat"` is a sequence of the characters “c”, “a”, and “t”.

Strings can be typed directly into code by putting quotation marks around text. `"Cat"` is a string, and so is `"dog"`. There was a string in the previous level: `"Hello, World!"`.

There are lots of different things we can do with strings using code. In the previous level, we saw that we can display strings on the screen by using `puts`:

```ruby
puts("this is a string")
```

In order to complete this level, we will also need to combine strings using `+`:

```
puts("Justin" + "Bieber")
```

When run, the code above displays “JustinBieber” without any space between the strings. When joining strings, a new string is made by copying all the characters from the first string and adding all the characters from the second string. Neither of the strings above contain the space character, so the resulting string has no space character either. If we want a space, we have to include a space inside one of the strings, like so:

```
puts("Justin " + "Bieber")
```

We can join as many strings as we want:

```
puts("R" + "E" + "S" + "P" + "E" + "C" + "T")
```

Variables

A variable is a container with a name. Here is a variable called `greeting` that contains the string `"Good evening"`:

```
greeting = "Good evening"
```

Notice that the variable does not have quotation marks, but the string does.

The = in the code above is not the same as an equals sign in math. In Ruby, the = character is the *assignment operator*, which is used to put a value into a variable. It takes the value on the right (in this case `"Good evening"`) and stores it inside the variable on the left (in this case `greeting`).

Variables act exactly like the value that they contain. For example, instead of using `puts` on a string directly, we could store a string in a variable and then `puts` the variable like this:

```
greeting = "Good evening"
puts(greeting)
```

The output of the code above is this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
Good evening
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Notice how it didn't output the text "greeting". Also notice how there are no quotation marks on the `puts` line. The first line stores a string inside a variable, and the second line displays the string inside the variable.

Strings are data, and data can be displayed, stored, modified, sent over the internet, etc. Variables are *not* data – they are just containers that hold data. That is why we can not `puts` the name of a variable, we can only `puts` the string inside the variable.

Variables are *mutable*, which means that the value inside them can change. For example:

```
greeting = "Good evening"
puts(greeting)
greeting = "Top of the morning to you"
puts(greeting)
```

The code above outputs the following text:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>
Good evening
Top of the morning to you
<<<<<<<<<< Finished successfully <<<<<
```

That is why they are called variables – because the value inside them can vary.

We can even copy the contents of one variable into a different variable:

```
greeting = "Good evening"
other = greeting
puts(other)
puts(greeting)
```

Guess what the code above will output, then run it to see if you were correct.

Variable Names

We can make as many variables as we wish, as long as they all have different names. We get to choose the variable names, but there are some restrictions on what names we can choose. Here are the rules for naming variables in Ruby:

- Variable names must only contain lower-case letters, numbers, and underscores (_).
- Variables must *not* start with a number.
- Variables must *not* be one of the following *keywords*, which are special words in the Ruby programming language: `__ENCODING__`, `__LINE__`, `__FILE__`, `BEGIN`, `END`, `alias`, `and`, `begin`, `break`, `case`, `class`, `def`, `do`, `else`, `elsif`, `end`, `ensure`, `false`, `for`, `if`, `in`, `module`, `next`, `nil`, `not`, `or`, `redo`, `rescue`, `retry`, `return`, `self`, `super`, `then`, `true`, `undef`, `unless`, `until`, `when`, `while`, and `yield`.

Here are some examples of valid variable names:

- `hello`
- `my_name`
- `my_favourite_chocolate_bar`
- `book2`

Here are some invalid variable names:

- `hello!` (exclamation marks are not allowed)
- `my name` (spaces are not allowed)
- `my-favourite-chocolate-bar` (hyphens are not allowed)
- `2nd_book` (must not start with a number)

Functions

At this point you may be wondering if `puts` is a variable. It does look like a variable, but there is a difference: `puts` is immediately followed by brackets. These brackets are called *parenthesis*, and parenthesis indicate that `puts` is not a variable – it is a *function*.

Functions work like this:

1. They *optionally* take in one or more values (like strings).
2. Then they do something.
3. Then they *optionally* return one value (like a string).

When I say that functions "do something," that "something" depends on which function we are talking about. There are literally thousands of different functions that Ruby provides, and they each do something different. In the case of the `puts` function, it takes in a string and then it displays that string.

To complete this level we will need another function called `gets`, which is sort of the opposite of `puts`. While `puts` takes a string to display, `gets` waits for the user to type in a string on the keyboard, and then it *returns* that string so we can use it in our code.

Let's look at an example of how `gets` works:

```
puts("Type in something, and then press enter")
text = gets()
puts("This is what you typed in: " + text)
```

We've already seen the two `puts` lines before. The first line displays a string. The last line displays two strings joined together, one of which is in a variable.

Let's focus on the second line: `text = gets()`. Firstly, there is a variable called `text`, and we are putting something into it. Secondly, notice how the `gets` function has parenthesis, but there is nothing inside them. This indicates that the `gets` function does not take in any values. The `puts` function takes in one string value, but `gets` takes nothing.

Now, run the code example above. It will display something like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>
Type in something, and then press enter
```

The first line of code has run, but the last line has not run yet. This is because the code is frozen on the second line. When we use the `gets` function, it freezes our program until the user presses the enter key.

Now type something into the window and press enter. The output should look like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
Type in something, and then press enter
Baby, baby, baby, oooooooh!
This is what you typed in: Baby, baby, baby, oooooooh!
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Once the user presses enter, the code unfreezes and `gets` *returns* a string. That is, the `gets` function records a string of characters from the keyboard until enter is pressed, then it gives that string back to our code. We are storing this returned string inside of a variable called `text`. Later, on the last line, the string is displayed with `puts`.

Comments

Comments are arbitrary text that we can write throughout our code, that will not be run like code. They are used to record extra information about the code. Here is an example:

```ruby
# this is a comment
puts("This is code")
# this
# is
# another
# comment
puts("More code") # comment 3
```

When Ruby is running our code, it completely ignores comments as if they don't exist. That means they can contain any text we want, and they will not interfere with the running of our code.

Comments start with a `#` character. This character has many different names: "octothorpe", "hash", and "pound", just to name a few. Ruby will ignore this character and everything that follows it on the same line.

We can write comments on their own line:

```ruby
# this displays the text "Hello, World!"
puts("Hello, World!")
```

And we can write comments at the end of a line of code:

```ruby
puts("hi") # this displays "hi"
```

Comments can be written over multiple lines by starting each line with the octothorpe character:

```ruby
# This displays "Hi there"
# by joining two strings together
puts("Hi " + "there")
```

The only place that we can't write a comment is inside a string. For example, the following is not a comment, it is just a string that contains an octothorpe character:

```
puts("Hello # this is not a comment")
```

Comments are normally used to explain and describe code, both for the person who wrote the code, and for other people who will read and use the code later. Write comments in your own code wherever you find them useful.

For the rest of this book, I will be using comments in the code examples to help explain bits of the code.

Boss Project: Name the Ogre

Now comes the time to test your new skills!

Write a program that allows the user to name an angry ogre. The output of your program should look like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>
Me am ogre with no name.
Give me name, or I smash you with club!
Silly Ogreman
Hmmmm. Me called Silly Ogreman
That sound good. Me go now.
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Optional Further Exercises

- Put comments above each line of your code, explaining what the line of code does.
- Does the name of your variable accurately describe the value that it contains? If not, try renaming the variable to something more descriptive, like `ogre_name`.