PROGRAMMING FOR BEGINNERS

Learn to Code by Making Little Games

TOM DALLING

Programming for Beginners

Learn to Code by Making Little Games

Tom Dalling

Contents

Introduction

So you want to learn how to write code. As of 2015, 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.

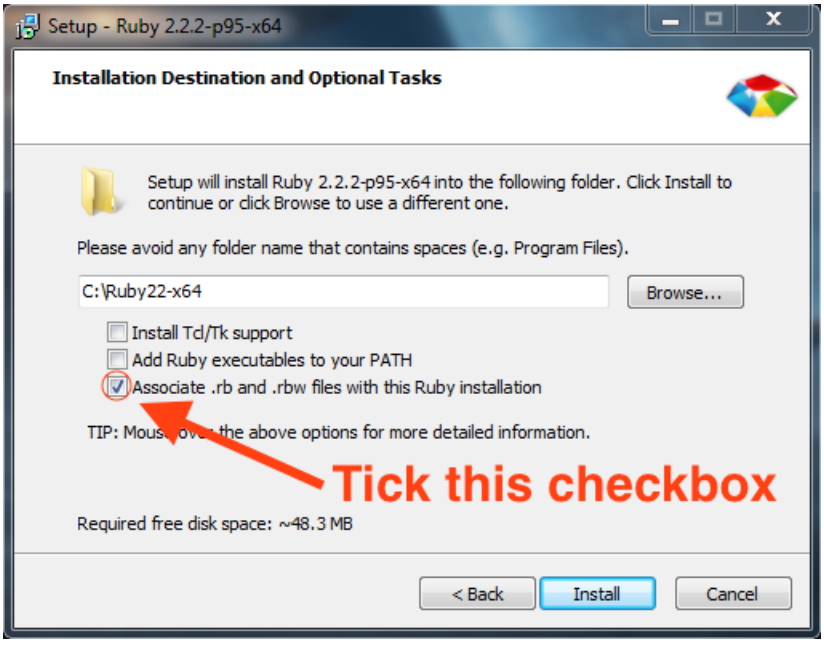
If your computer is a Mac, then you already have Ruby. OS X comes with Ruby already installed. If you have an old version of OS X, you will need to upgrade to at least OS X 10.9 (Mavericks) to get the right version of Ruby. Skip ahead to *Install A Text Editor*.

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

First, determine whether you have a 64-bit or 32-bit[1] version of Windows. If it is 64-bit, then download the installer called `Ruby 2.2.3 (x64)`. Otherwise, download the 32-bit installer called `Ruby 2.2.3`.

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



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)
- Text Wrangler[3] (OS X only)

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

[3] http://www.barebones.com/products/textwrangler/download.html

Write the Code

Create a new folder called `RubyProjects` inside the `Documents` folder on your computer. This is the folder where we will keep all of our different Ruby projects.

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:

```
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.

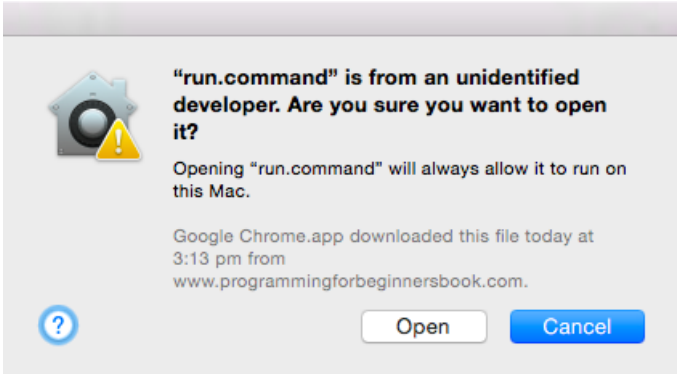
Running code can be a little complicated at first, so I have created a *code runner* file to simplify the process.

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

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

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

After unzipping `run.command`, **control-click (or right-click) it and select the "Open" option**. OS X will ask if you're sure that you want to open the file, then click the "Open" button.



OS X confirmation dialog

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

In future, to make a new Ruby project:

1. Create a new folder.
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 OS X) 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:

```
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`:

```
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:

```
# 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:

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

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

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

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

```
# 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`.

Level 3: Branching

In this level, we will make the first program that could be considered a *game* – something that a player can win or lose. To achieve this, we will learn about:

- some programming terminology
- integers
- booleans
- conditional execution
- converting strings to integers

Make a new project folder, just like in the last level, and let's get started!

Some Terminology

Below are the definitions for a few important terms that I will be using from now on. Until this point, I've been trying to use common words to explain programming language concepts, but these concepts have specific terminology, and we can't talk about the concepts accurately unless we use the correct terms.

Value
: *(noun)* anything that can be stored inside a variable. Strings are the only type of value we have seen so far, but there are other types of values (like numbers) that we will soon encounter.

Assign
: *(verb)* to put a value into a variable. Example: *I assigned the value* `"Tom"` *to the* `name` *variable.*

Argument
: *(noun)* a value that goes into a function. Arguments are also known as *parameters*. Functions are said to "take" arguments. Example: *the `puts` function takes one argument.*

Return value
: *(noun)* a single value that comes out of a function, when that function has finished running.

Return
: *(verb)* to provide a return value. Example: *the `gets` function returns a string.*

Call
: *(verb)* to run a function. Example: *I called the `gets` function.* Calling a function is also known as "running", "executing", or "invoking" a function.

Function call
: *(noun)* code that calls a function. Example: *`puts("hello")` is a function call.*

Function
: *(noun)* a named piece of code that is not run until it is called. It takes zero or more arguments, and may return a single value.

You may be surprised to learn that functions are pieces of code. Every time we call the `gets` or `puts` function, it is running code that someone else has written for us. This will become more clear in later levels when we make our own functions.

Integers

Integers are values that represent whole numbers – that is, numbers that are *not* fractions. Integers include negative numbers, positive numbers, and zero.

As with all values, they can be stored inside variables:

```
x = 5
puts(x)
```

Ruby provides ways to add, subtract, multiply, and divide integers:

```
x = 6
y = 2
puts(x + y) # addition
puts(x - y) # subtraction
puts(x * y) # multiplication
puts(x / y) # division
```

The output for the code above is this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
8
4
12
3
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Booleans

Booleans are another type of value, like integers and strings. Unlike integers (which can be any whole number) and strings (which can be any text), there are only two boolean values: `true` and `false`. Booleans represent yes-no values, like the answer to the question "*is the player's health above zero?*" The answer is either `true` if the player has any health, or `false` if the player has zero health (or negative health).

We can type `true` and `false` directly into the code, like so:

```
puts(true)
puts(false)
```

Booleans are values, which means they can be assigned to variables:

```
alive = true
puts(alive)
```

Most of the time, boolean values are made by testing other values. Ruby provides lots of different tests. There are `>` and `<` to test if one number is greater or less than another:

```
health = 50
puts(health > 0)  # true
puts(health < 20) # false
```

There is ==, which tests whether two values are equal to each other. Note that this equality test has *two* equals signs, and is different to the single equals sign. The single equals (=) is used for variable assignment, but the double equals (==) is an equality test that results in a boolean.

```
puts(5 == 5)        # true
puts(5 == 6)        # false
puts("hi" == "hi")  # true
puts("hi" == "bye") # false
puts("hi" == 5)     # false
puts("5" == 5)      # false (because it's string vs integer)
```

There is also `!=`, commonly called "not equal," which is the exact opposite of ==. It is `true` when both values are different, and `false` when they are equal.

```
# is five not equal to five?
puts(5 != 5) # false

# is five not equal to seven?
puts(5 != 7) # true
```

Below are some more complicated tests. Guess the boolean for each line, then run the code to see if you were correct.

```
x = 5
y = 10
puts(x == y)
puts(y != x)
puts(x > 5)
puts(y > x)
puts(x == y / 2)
puts(y == x * 2)
puts(y - x == x)
puts(x + x != y)
```

By themselves, booleans are pretty useless. But they become extremely useful once we learn how to use them for…

Conditional Execution

We don't want to run all of our code, all of the time. If we're making a game, dead players shouldn't be able to run the code that makes them jump. Choosing which code gets run, and which code doesn't, is called *conditional execution*. Conditional execution is where we only run some code depending on a boolean value.

Here is an example:

```
health = 50
alive = health > 0  # true
if alive
  puts("You are still alive!")
else
  puts("You are dead")
end
```

The code above contains an example of a *conditional*. Let's look at some more examples of conditionals. Try to guess the output of the examples below, then run them to see if you were correct.

```
# Conditional Example 1
if true
  puts("true branch")
else
  puts("false branch")
end
puts("after")


# Conditional Example 2
if false
  puts("true")
  puts("branch")
else
  puts("false")
  puts("branch")
end
puts("after")


# Conditional Example 3
health = 75
if health > 50
  puts("You are pretty healthy, friend!")
else
  puts("You are not very healthy, buddy.")
end
```

```
# Conditional Example 4
age = 45
if age < 5
  puts("You are too young to code")
else
  if age < 200
    puts("You can learn to code!")
  else
    puts("What are you? A vampire?")
  end
end


# Conditional Example 5
puts("Type in your age:")
age_string = gets()
if Integer(age_string) == 27
  puts("You are 27? Me too!")
else
  puts("Cool beans.")
end
```

Conditionals Explained

The general structure of a conditional looks like this:

```
if some_boolean_value
  puts("true branch")
else
  puts("false branch")
end
```

Let's look at each part of the code above.

Firstly, we have the word `if`. This is not a variable, or a function – it is a *keyword*. Keywords are special words that are used to express different features of the programming language. The `else` and `end` words are also keywords. Together, the `if`, `else` and `end` keywords are used to express the conditional execution feature of the Ruby programming language.

Next, there is `some_boolean_value`. This is, as the name suggests, a boolean value. This could be a variable that contains a boolean, or a test that results in a boolean, or any other way of providing a boolean value. Ruby looks at this value, and runs different code depending on whether it is `true` or `false`. This boolean value is called the *condition* of the conditional.

Next comes `puts("true branch")`. This could be any code – even multiple lines of code. We can even put another conditional here. This is the code that will only run if the condition is `true`. This section of code is called the *true branch*, and it includes everything between the `if` line and the `else` line.

We have already covered the `else` keyword above, so next is `puts("false branch")`. This is exactly the same as the true branch, except it's the opposite – it only gets run if the condition is `false`. This is called the *false branch*, and it contains everything between the `else` line and the `end` line.

Conditionals are also called "branches" because they act like a fork in a river. When a boat encounters a fork, it must choose which branch to go down. Branching is another aspect of control flow. Like a boat at a fork in a river, control flows down one of the branches, but not both.

Notice that the code in both branches is indented with spaces. While these indents do not affect how the code gets run, they are very much necessary. I'm going to introduce a law: **all code in the true branch and the false branch must be indented**. The code will still work without indenting, but there is universal agreement among programmers that indenting is necessary to make code readable. For small, simple programs, indenting might not seem important. But for larger, more complicated programs, readability is extremely important.

In summary, `if`, `else`, and `end` are special keywords that are used to make a conditional. The condition is a boolean value that we provide, and it determines whether the true branch or the false branch gets run. The two branches can contain any amount of code, and that code must be indented.

Other Ways of Writing Conditionals

There are a few different ways to write conditionals in Ruby. We've only seen one that contains `if` and `else`:

```
x = 5
if x == 5
  puts("It's five")
else
  puts("It's not five")
end
```

The false branch is actually optional. If we don't have any code in the false branch, we can just put the true branch between `if` and `end` like so:

```
x = 5
if x == 5
  puts("It's five")
end
```

If we have only one line of code in the true branch, we can write a *postfix* `if`:

```
x = 5
puts("It's five") if x == 5
```

This postfix `if` works exactly the same as the previous code example. It's just written in a more compact way that sounds a bit closer to English. Postfix conditionals only get applied to a single line of code, and there is no way to have a false branch.

There is another conditional keyword called `unless`. The `unless` keyword works exactly the same as `if`, except it's the opposite – the true branch and the false branch are swapped:

```
x = 5
unless x == 5
  puts("It's NOT five")
else
  puts("It's five")
end
```

Wherever we can use `if`, we can also use `unless`:

```
x = 5

unless x == 5
  puts("It's NOT five")
end

puts("It's not seven") unless x == 7
```

The `unless` keyword is useful when we have code in the false branch, but nothing in the true branch. For example, instead of writing this:

```
x = 7
if x == 5
else
  puts("It's NOT five")
end
```

We should write this:

```
x = 7
unless x == 5
  puts("It's NOT five")
end
```

Or even better, this:

```ruby
x = 7
puts("It's NOT five") unless x == 5
```

As the examples above demonstrate, there are multiple ways to write the same conditional. Here is the condition "*if x is not equal to 5*" written in six different ways:

```ruby
x = 7

# 1: `if` with a "not equal"
if x != 5
  puts("not five A")
end

# 2: using the false branch of `if`
if x == 5
else
  puts("not five B")
end

# 3: postfix `if` with a "not equal"
puts("not five C") if x != 5

# 4: `unless`
unless x == 5
  puts("not five D")
end

# 5: postfix `unless`
puts("not five E") unless x == 5

# 6: false branch of `unless` with a "not equal"
unless x != 5
else
  puts("not five F")
end
```

You should write your conditionals in whatever way makes your code easier to read and understand. Generally, the less code there is, the easier it is to understand. Try to write your conditionals in the shortest way possible, unless you find a longer way easier to read.

Beware the Difference between = and ==

A very common beginner mistake is to use single equals (=) where it should have been double equals (==). Although they look similar to each other, they are actually very different.

Single equals is the *assignment operator*. It is used to assign values to variables. For example, `x = 5` means "*put the value `5` into the variable `x`*".

Double equals is the *equality operator*. It is used to test whether two values are equal, and returns a boolean. For example, `x == 5` means "*is the value in the `x` variable equal to the value `5`?*" The answer is either `true` if they are equal, or `false` if they are not.

Typically, students will make a mistake like this:

```
# this example contains a mistake
x = 3
if x = 5
  puts("It's five")
else
  puts("It isn't five")
end
```

Reading the code as if it were English, the conditional sounds like "*if x equals five.*" That sounds correct. However, Ruby is not English, and what it actually means is "*if five is assigned to x.*" This condition is *always true*, because the assignment happens every time. It is actually impossible to run the false branch of the code above, no matter what value `x` contains. Every time the condition is run, it assigns `5` to `x`, replacing whatever value `x` contained beforehand.

Thankfully, Ruby displays a warning message if we make this mistake. Running the incorrect code above should produce this output:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
game.rb:2: warning: found = in conditional, should be ==
It's five
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

The `game.rb:2` part of the message says that the warning occurred on line 2 of the `game.rb` file. If you see this warning, go to the line number it says and change the single equals to a double equals, like this:

```
x = 3
if x == 5
  puts("It's five")
else
  puts("It isn't five")
end
```

Getting Integers from the User

We have already seen that the `gets` function returns a string:

```
puts("Type your name:")
name = gets()
puts("Your name is " + name)
```

But what if we wanted the user to type in an integer? Try running this code and see what happens:

```
puts("Type in your age:")
age = gets()
puts("You are half way to:")
puts(2 * age) # crashes here
```

The above code crashes with an error like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>>>>>
Type in your age:
28
You are half way to:
game.rb:4:in `*': String can't be coerced into Fixnum (TypeError)
        from game.rb:4:in `<main>'
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!!!!!!!
```

The error says that on line four of `game.rb`, while doing multiplication with `*`, there was a string when there should have been an integer (`Fixnum` means integer in Ruby). Multiplication doesn't work with strings, so the program crashed. The output indicates that this was a `TypeError`, which means we used the wrong type of value. In this case, the correct type of value is an integer, but we used a string instead.

The `gets` function returned the *string* `"28"`, not the *integer* `28`. It always returns a string, even if the user types in an integer. We stored that string in the `age` variable, and the program crashed when we tried to do multiplication with it.

If we want to work with integers typed in by the user, we have to convert strings into integers. Thankfully, there is a function that does exactly that, and it is called `Integer`. The `Integer` function takes a string argument, converts the string to an integer, and returns the integer. For example:

```
puts("Type in your age:")
age_string = gets()
puts("Next year you will be:")
puts(Integer(age_string) + 1)
```

We could also write the exact same thing like this:

```
puts("Type in your age:")
age = Integer(gets())
puts("Next year you will be:")
puts(age + 1)
```

Both examples above convert the string returned from `gets` into an integer before trying to do the addition. In the first example, the string is still stored in the `age` variable, but it gets converted just before the addition happens. In the second example, the string gets converted immediately – the return value from `gets` is being used as the argument for `Integer`, and the `age` variable is assigned an integer: the return value from `Integer`.

Below is another error that is caused by `gets` returning a string, but this one is sneakier. Run the following code and see if you can spot the problem:

```
# this example contains a mistake
puts("Type the number 4:")
number = gets()
if number == 4
  puts("You did it!")
else
  puts("That wasn't what I asked!")
end
```

This program won't crash, but it will never tell us that `"You did it!"`. Earlier in this level, we saw that when we use == with a string and an integer it always returns `false`. We can see this happening by running the following code:

```
puts(4 == 4)     # true (integer vs integer)
puts("4" == "4") # true (string vs string)
puts("4" == 4)   # false (string vs integer)
```

The first two lines are `true`, but the last line is `false`.

Once again, this problem can be fixed by converting the string into an integer using the `Integer` function:

```
puts("Type the number 4:")
four = gets()
if Integer(four) == 4
  puts("You did it!")
else
  puts("That wasn't what I asked!")
end
```

Another thing to be careful of is that not all strings can be converted into integers. If the string doesn't contain a number, the `Integer` function will crash the program. For example, try running the following code:

```
# this crashes
age = Integer("hello")
```

It will crash with an error that looks like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
game.rb:1:in `Integer': invalid value for Integer(): "hello" (Argume\
ntError)
        from game.rb:1:in `<main>'
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!
```

This crash tells us that on line `1` of `game.rb`, the `Integer` function received an invalid value. It then tells us that the invalid value was `"hello"`. It also tells us that this type of error is an `ArgumentError`, which means that an argument we gave to a function was incorrect.

Boss Project: Guess the Number

Write a program that asks the user to guess a number between 1 and 10. If the user guesses the number 4 then tell them that they win, otherwise tell them that they lose.

The output of your program should look like this when the user wins:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
I am thinking of a number between 1 and 10.
Type in your guess:
4
That is correct! You win!
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

When the user loses, the output should look like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
I am thinking of a number between 1 and 10.
Type in your guess:
6
That is wrong. You lose.
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Optional Further Exercises

- Try to make a very simple calculator. Ask the user to enter two numbers, then add the numbers together and display the result.
- Try to write the number guessing program without using any variables. It is possible!

Level 4: Looping

In this level, we will start to introduce some randomness. When the correct answer is always `4` the game gets boring pretty quickly, and randomness will help to fix that.

We will also look at the next fundamental concept in control flow: looping.

Putting Variables into Strings

We've already seen that strings can be joined together:

```
puts("I'm " + "Tom" + " and I'm " + "28" + " years old")
```

We've also seen that variables act exactly like the values that they contain. That means we can join variables into strings, as long as those variables contain strings:

```
name = "Tom"
age = "28"
puts("I'm " + name + " and I'm " + age + " years old")
```

But we haven't seen an integer joined with a string yet. Try running the following code, where the `age` variable has been changed from a string to an integer:

```
name = "Tom"
age = 28 # <--- is an integer now
puts("I'm " + name + " and I'm " + age + " years old")
```

It crashes with an error that looks like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
game.rb:3:in `+': no implicit conversion of Fixnum into String (Type\
Error)
        from game.rb:3:in `<main>'
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!!
```

This error tells us that on line three of the `game.rb` file, we were joining strings together using `+`, but we tried to join an integer (a.k.a. Fixnum) into the string, and that is not allowed. Ruby says that this error is a `TypeError`, because the integer value is the wrong type of value. It all worked fine when the `age` variable was a string, because strings are the correct type of value.

How do we join an integer into a string? We convert the integer into a string first, *then* we join it with the other strings. We can easily convert an integer into a string using a function called `String`, like so:

```
name = "Tom"
age = 28
puts("I'm " + name + " and I'm " + String(age) + " years old")
#     Converts integer to string here ^
```

The `String` function takes one argument, converts that argument into a string, and then returns that string value. The `String` function can take any type of argument – including integers, booleans, and even strings.

```
puts("Integer: " + String(5))
puts("Boolean: " + String(true))
puts(" String: " + String("hello"))
```

Random Numbers

Number guessing games get boring pretty quickly when the secret number is always four. Let's fix that by introducing some randomness.

There is a function called `rand` that returns a random integer. It takes one integer argument, and returns a random integer that is zero or greater, but less than the given argument. For example, `rand(3)` will return `0`, `1` or `2`.

The `rand` function returns a different value every time it is called. Try running the following code a few times, and notice how the output keeps changing:

```
puts(rand(3))
puts(rand(3))
puts(rand(3))
puts(rand(3))
```

As with `gets()` and all other function calls, the return value from `rand` can be stored inside a variable:

```
x = rand(3)
puts(x)
```

Updating Variables

Often times, while programming a game, we will need to change a variable based on its current value. For example, if a player picks up a health potion, we might want to take their current health and increase it by a certain amount. In the code below, the `health` variable starts at `70`, and then it gets added to, multiplied, and subtracted from:

```
health = 70
puts(health)
health = health + 10  # 70 + 10
puts(health)
health = health * 2   # 80 x 2
puts(health)
health = health - 50  # 160 - 50
puts(health)
```

The output from the code above looks like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
70
80
160
110
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

This is actually another example of control flow. When assigning a value to a variable, the code on the right-hand side of the = is run first, before the variable assignment happens.

Let's look at a simple example:

```
health = 70
health = health + 10
puts(health)
```

First, `70` is put into the `health` variable. Next, the code `health + 10` is run, which returns the value `80`. Lastly, the value `80` is assigned to the `health` variable, replacing the previous value that the variable contained.

The same thing applies to all types of values. Here is an example that joins strings:

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

Looping

Looping allows us to run a section of code *repeatedly*. Like conditional execution, looping is another fundamental control flow concept. The first looping construct we will look as is called a *`while` loop*:

```
i = 5
while i > 0
  puts(i)
  i = i - 1
end
puts("Blast off!")
```

The output for the code above is:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
5
4
3
2
1
Blast off!
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

There are similarities between the code for `while` loops and conditionals. They both start with a keyword, followed by a boolean condition value. They both have indented code inside them. They both end with the `end` keyword. The difference is that conditionals *choose* a section of code, and `while` loops *repeat* a section of code.

Let's look at some more examples of `while` loops. Guess what the output will be, then run the examples to see if you were correct.

```ruby
# Loop Example 1
i = 1
while i < 10
  puts(i)
  i = i + 1
end
puts("done")


# Loop Example 2
while false
  puts("hello?")
end
puts("done")


# Loop Example 3
running = true
while running
  puts("running")
  running = false
end
puts("done")


# Loop Example 4
guess = 4
while guess == 4
  puts("Type in the number 4:")
  guess = Integer(gets())
end
puts("That wasn't a 4!")
```

Notice how all the code inside the `while` loops is indented, like the branches in conditionals. When you are writing code, stick to this standard, and **always indent the code inside `while` loops**.

Beware of Infinite Loops

What happens if the condition of the `while` loop is always true, like below?

```
# an infinite loop
while true
  puts("hello")
end
```

The program will be stuck forever in an *infinite loop*. This is almost always a mistake, and is something we try to avoid.

Try running the following code:

```
# another infinite loop
i = 1
while i > 0
  puts(i)
  i = i + 1
end
```

This is another infinite loop. It displays numbers as fast as possible, and will never stop by itself. To force the program to stop, you will have to close the window.

The `i` variable starts at one, and then increases by one every loop. The `while` loop will keep running as long as `i` is greater than `0`, but `i` will *always* be greater than zero because it is increasing.

If you are running some code and the program appears to freeze, or repeatedly displays text and doesn't stop, then you probably have an infinite loop. In these situations, check your loop conditions and make sure that they are correct.

Displaying Empty Lines

To make the output of our programs look nicer, we can separate parts of it with empty lines. You may have already discovered that you can do this by using `puts` with a string that contains a space:

```
puts(" ")
```

We could achieve the same thing by using `puts` with an *empty string*. An empty string is a string that has *no* characters in it. Empty strings are made by typing two quotes with nothing between them:

```
puts("")
```

There is yet another way to display empty lines – we can call the `puts` function with no arguments:

```
puts()
```

The `puts` function is an example of a function that does different things depending on the number of arguments it is given. When one argument is provided, it displays that argument on a line. When no arguments are provided, it just displays a blank line.

Boss Project: Super Sums

Write a program that asks the user three simple sums. Each sum should ask the user to multiply two random numbers, between one and ten. The program should keep score of how many questions the user got correct, and display the score at the end. Your program *must contain a `while` loop that loops three times.*

The output for your program should look something like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
Welcome to Super Sums!
======================

What is 1 times 6?
6
Correct!

What is 2 times 3?
3
Incorrect.

What is 4 times 7?
28
Correct!

Thank you for playing!
You scored 2 out of 3
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Optional Further Exercises

- Make your program ask five sums instead of three. Did it require many changes to the code? Did the score at the end change to "out of 5"?
- Try write your program so that the number of questions is controlled by a *single* integer value. To change the number of questions, you should be able to change that single value *and nothing else*.

Level 5: Composite Values

In this level we start to see more complicated types of values: composite values. We will use these composite values to make a multiple choice quiz game.

Arrays

Arrays are lists of values. That is, they contain multiple values in a specific order. For example, this is an array of integers:

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

And here is an array of booleans:

```
[true, false, false, true, true]
```

And here is an array of strings:

```
["cat", "dog", "pig"]
```

Values inside arrays are called *elements*. For example: *the array has three elements, and the first element is* `"cat"`. Arrays can have any number of elements, and those elements can be any type of value.

Not only do arrays *contain* values, they *are* values themselves. As such, they can be stored in variables:

```
my_lottery_numbers = [1, 26, 31, 16, 8]
```

As shown in the example code above, making an array follows a specific pattern. To make an array in Ruby, we start with a `[` (left square bracket). Then we write all the values that the array contains, with commas between them. Then, at the end of the array, we type a `]` (right square bracket). This pattern must be followed *exactly*. I can guarantee that you will forget a comma at some point, so double check them if your program is crashing.

We can also use arrays with `puts`:

```
lyrics = ["Heigh-ho", "Heigh-ho", "It's off to work we go"]
puts(lyrics)
```

When `puts` receives an array argument, it displays every element of the array on a separate line:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
Heigh-ho
Heigh-ho
It's off to work we go
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Array Indices

Each element in an array can be accessed using its position in the array. This array position is called an *index*, and it is an integer. The plural of index is *indices*.

Consider the following array:

```
["cat", "dog", "pig"]
```

You may expect that the element indices look something like this:

- Index `1`: `"cat"`

- Index 2: `"dog"`
- Index 3: `"pig"`

However, the indices above are actually incorrect. For historical reasons, the first element in an array is at index `0`, in Ruby and most other programming languages.

These are the correct element indices:

- Index `0`: `"cat"`
- Index `1`: `"dog"`
- Index `2`: `"pig"`

The first index is `0`, and the last index is one less than the number of elements in the array. This is important to keep in mind when working with array indices.

This is how we use an index to get an element out of an array:

```
#               0      1      2
my_animals = ["cat", "dog", "pig"]
puts(my_animals[0]) # cat
puts(my_animals[1]) # dog
puts(my_animals[2]) # pig
```

And here is the output:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
cat
dog
pig
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

To get an element from an array, put an index inside square brackets directly after the array variable.

Guess what the following code examples do, then run them to see if you were correct:

```
# Index Example 1
#           0    1    2    3
letters = ["a", "b", "c", "d"]
puts(letters[3])

# Index Example 2
#          0       1       2       3
names = ["Dane", "Tom", "Rhi", "Nick"]
index = 1
puts(names[index])

# Index Example 3
animals = ["bison", "snake", "frog", "bat"]
index = 0
while index < 4
  puts(animals[index])
  index = index + 1
end
```

If an array is getting a bit long, it can be written over multiple lines like this:

```
lyrics = [
  "I am the very model of a modern major general",
  "I've information vegetable, animal, and mineral",
  "I know the kings of England and can quote the fights historical",
  "From Marathon to Waterloo, in order, categorical"
]
puts(lyrics)
```

This way of writing arrays makes long arrays easier to read, and it still follows the pattern described above: values separated by commas inside square brackets.

Out of Bounds Errors

There is a type of error called an *index out of bounds* error, and it occurs when we use an index that doesn't exist in the array. For example:

```
#           0       1       2     ...     9?
colors = ["green", "blue", "red"]
puts(colors[9])
```

The code above outputs this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>

<<<<<<<<<< Finished successfully <<<<<<<<<<
```

There's nothing but a blank line.

The code tries to get the element at index `9`, but the array only has three elements. This is a programming error, and most programming languages will crash when this runs. Ruby, however, does something worse than crashing: it fails silently.

The code `color[9]` results in a value called `nil`. We will get to `nil` later in this level. For now, just know that accessing an element that doesn't exist is a mistake that we try to avoid.

For Loops

There is another kind of loop called a *`for` loop*:

```
animals = ["cat", "dog", "fish"]
for pet in animals
  puts("I have a " + pet)
end
```

The above code outputs this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
I have a cat
I have a dog
I have a fish
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

The most common kind of loops are loops that do something to every element of an array. It is so common, that the `for` loop is specifically designed to make this kind of loop easier to write in code. If you need to loop over every element in an array, use a `for` loop, otherwise use a `while` loop.

We could use a `while` loop to loop over an array, but it takes more code, and it's not as easy to read:

```
animals = ["cat", "dog", "fish"]
index = 0
while index < 3
  puts("I have a " + animals[index])
  index = index + 1
end
```

The general structure of a `for` loop looks like this:

```
for some_variable in some_array
  put(some_variable)
end
```

It starts with `for`, which is a keyword.

Next is `some_variable`, which is a variable name. Like all other variables, we can choose this name. Unlike other variables, we do not assign a value to it. Values are assigned to this variable *automatically* by the `for` loop. Every time this loop runs, the next element from the array will get assigned to this variable.

Next comes the `in` keyword.

Then comes `some_array`, which can be any array value. We can even type an array straight into this spot:

```
for x in [1, 2, 3]
  puts(x)
end
```

Indented inside the `for` loop is some code. This code gets run once for every element in the array. If the array has five elements, the code inside the `for` loop will run five times. **Always indent the code inside a loop**.

Lastly there is the `end` keyword, which delimits the end of the loop code.

There is no index variable provided by the `for` loop, so if we need the index, we must keep it as a separate variable and update it manually:

```
index = 0
for pet in ["dog", "cat", "fish"]
  puts(String(index) + " " + pet)
  index = index + 1
end
```

Hashes

Hashes are a type of value that contain other values, like arrays do. Unlike arrays, hashes are not lists – they are more like dictionaries. Let's look at one:

```
{
  "bread" => "food made of flour, water, and yeast",
  "toast" => "sliced bread browned on both sides",
  "toaster" => "an electrical device for making toast"
}
```

Hashes even look similar to arrays in the code. They have elements separated by commas inside of brackets – although they are *curly brackets* instead of square brackets. The main difference is that each element is a *pair* of values, separated by `=>` (a *hash rocket*). It is called a hash rocket because it looks a little bit like a sideways

rocket, and it appears inside hashes. On the left side of each hash rocket is the *key*, and on the right side is the associated *value*.

Hashes don't have indices. Instead, the *keys* are used to look up values. This is how we look up a value in a hash:

```
dictionary = {
  "bread" => "food made of flour, water, and yeast",
  "toast" => "sliced bread browned on both sides",
  "toaster" => "an electrical device for making toast"
}
puts(dictionary["toaster"])
```

The above code outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
an electrical device for making toast
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Again, this is very similar to accessing array elements. The difference is that, inside the square brackets, a hash *key* is used instead of an index.

Remember that the key can also be a variable:

```
toaster_key = "toaster"
dictionary = {
  "bread" => "food made of flour, water, and yeast",
  "toast" => "sliced bread browned on both sides",
  toaster_key => "an electrical device for making toast"
}
puts(dictionary[toaster_key])
```

Hash values don't have to be strings. Just as array elements can be any type of value, so can hash values:

```
hash = {
  "boolean" => true,
  "integer" => 99,
  "array of integers" => [11, 12, 13],
  "another hash" => { "a" => 1, "b" => 2 }
}
puts(hash["integer"])
puts(hash["boolean"])
puts(hash["array of integers"])
puts(hash["another hash"])
```

The code above outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
99
true
11
12
13
{"a"=>1, "b"=>2}
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

We have to be careful not to use hash keys that don't exist. Here is a mistake, as an example:

```
# this code contains a mistake
person = { "name" => "Tom", "age" => 28 }
puts(person["namme"])
```

The code above outputs this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>

<<<<<<<<<< Finished successfully <<<<<<<<<<
```

There is a typo in `person["namme"]`. The hash key is `"namme"` when it should be `"name"`. The `"namme"` key does not exist in the hash, so it results in `nil`. We will get to `nil` later in this level, but for now, just know that it is a mistake to use a key that doesn't exist in the hash.

Composite Values

Arrays and hashes are examples of composite values. Composite values are values that contain other values. By putting composite values inside of other composite values, we can get bigger and more complicated values.

A very common example of this is an array of hashes:

```
people = [
  { "name" => "Tom",   "age" => 28 },
  { "name" => "Jane",  "age" => 29 },
  { "name" => "Sarah", "age" => 33 }
]
puts(people)
```

And here is a hash of arrays:

```
things_by_color = {
  "red"   => ["roses", "blood", "wine"],
  "green" => ["grass", "emeralds"],
  "blue"  => ["ocean", "sky", "tardis"]
}
puts(things_by_color)
```

All the brackets and commas are still there, in the correct places. The rules for writing hashes and arrays are still the same, it's just that they are inside of each other.

Representing Data with Values

Here is a composite value that represents a school:

```
school = {
  "subjects" => [
    { "code" => "MATH101", "name" => "Intro to Maths" },
    { "code" => "CHEM301", "name" => "Advanced Chemistry" }
  ],
  "students" => [
    {
      "id" => 12345,
      "name" => "Jimmy Dowl",
      "subjects" => ["MATH101", "CHEM301"]
    },
    {
      "id" => 98765,
      "name" => "Dane Williams",
      "subjects" => ["CHEM301"]
    }
  ]
}
puts(school["students"][1]["name"])
```

Even though the school only has two subjects and two students, it is still the most complicated value we have seen so far.

Hashes are normally used to hold the data for "things." In the example above, the "things" are: subjects, students, and the school. Each of those things is represented by a hash.

- Each student is a hash with an id, a name, and an array of subject codes.
- Each subject is a hash with a subject code, and a name.
- The school is a hash that contains an array of subjects, and an array of students.

This is the way that all software represents data. It starts with simple values like strings and integers, which get collected into composite values like arrays and hashes, which get collected into *other* composite values, and so on.

The final line in the example above uses `puts` to display the name of the second student of the school:

```
puts(school["students"][1]["name"])
```

It runs like this:

1. First, it accesses the `"students"` key of the `school` hash, which returns an array.
2. It accesses index `1` of that array, which returns a hash.
3. It accesses the `"name"` key of that hash, which returns the string `"Dane Williams"`.
4. Finally, it calls the `puts` function with `"Dane Williams"` as the argument.

This one line of code could be rewritten like this:

```
students = school["students"]
second_student = students[1]
student_name = second_student["name"]
puts(student_name)
```

The code above puts each value into a variable before accessing it. The `students` variable holds an array, `second_student` holds a hash, and `student_name` holds the string `"Dane Williams"`.

Nil

In most programming languages, there is a special value that represents the *lack* of a value. In Ruby, this value is called `nil`. The `nil` value kind of means "*nope, there is no proper value here.*"

The `nil` value can be typed directly into code, just like `true` or `false`. Being a value, it can be stored inside a variable:

```
whatever = nil
```

To demonstrate where `nil` is often used, let's consider a hypothetical function called `find_student_name`, which takes a student ID integer argument, and returns a name as a string. For example:

```
# hypothetical code
student_name = find_student_name(123)
puts("Student 123 is named " + student_name)
```

Assuming there is a student named John with the ID 123, then `find_student_name(123)` would return the string `"John"`.

Now consider what would happen if there was no student for the given ID. Assuming that there is no student who has the student ID 99999999, what will the return value of `find_student_name(99999999)` be? There is no correct string value that it could return. In this case, it would probably return `nil` to indicate that no student exists with that ID.

Other than representing the lack of a value, the `nil` value is basically useless. It doesn't really *do* anything. The only thing we can do with `nil` is test whether another value is equal to it:

```
student_name = find_student_name(99999999)
if student_name == nil
  puts("Student doesn't exist")
else
  puts("That student is " + student_name)
end
```

If we try to display `nil` using `puts`, it comes out as a blank line. Try running `puts(nil)` to see this. This is why, earlier in the level, the output was a blank line when accessing an array index that was out of bounds, and a hash key that didn't exist in the hash.

The `nil` value is the cause of a lot of programming mistakes. Consider this code:

```
# this example contains an error
students = ["John", "Rhi", "Nick"]
puts("The 1st student is " + students[0])
puts("The 10th student is " + students[9])
```

The code above crashes with this output:

```
>>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
The 1st student is John
game.rb:4:in `+': no implicit conversion of nil into String (TypeErr\
or)
  from game.rb:4:in `<main>'
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!!
```

This error says that `nil` can not be joined with a string using `+`.

The problem is that `student_name[9]` is out of bounds – the array does not contain an element at index `9` because it only has three elements. Instead of crashing, Ruby just returns `nil` instead. The last line of code was written with the expectation that `student_name[9]` would be a string, and therefor it could be joined to another string using `+`. But it is actually `nil`, which can *not* be joined with a string, so the program will crash on the last line.

Here is another mistake involving `nil`, this time using a hash:

```
students = { 123 => "John", 456 => "Monique" }
puts("Student 123 is " + students[123])
puts("Student 999 is " + students[999])
```

This is the exact same problem as with the array. Key `999` does not exist within the `students` hash, so it results in `nil`. Then, when the last line of code tries to join a string with `nil`, it crashes.

Crashing because of an unexpected `nil` value is a very common kind of mistake, even for professional programmers. Code is often written expecting a specific type of value, and it will crash if it encounters a `nil` value instead.

You are guaranteed to make a mistake involving `nil` at some point. Look out for `puts` displaying empty lines unexpectedly, and also look for "nil" in the crash error messages.

Boss Project: Quiz Whiz

Make a quiz that looks like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
Quiz Whiz
=========

Question: What is the capital of Nigeria?
  0. Paris
  1. Abuja
  2. Berlin
Which option do you choose?
1
That is correct!

Question: In what year was Julius Ceasar killed?
  0. 44 BC
  1. 192 AD
  2. 0 AD
Which option do you choose?
2
That is incorrect.

Question: Which artist wrote the song 'Life on Mars'?
  0. Freddy Mercury
  1. Jimmy Barnes
  2. David Bowie
Which option do you choose?
3
That is incorrect.

Your final score is 1/3
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

- The quiz should ask three questions of your choosing.

- Each question should present three numbered choices.
- The user should indicate their choice by typing in a number.
- Keep track of the user's score, and display it at the end.
- Represent the quiz data as a composite value.
  - The quiz should be represented as an array of questions.
  - Each question should be represented as a hash.
  - There should be an array of choices inside each question hash.
  - If you are having trouble with this, analyse how a school was represented as a composite value, earlier in this level.

You will need to use at least two loops to access the arrays. Remember that you can put a loop inside a loop!

This project may be quite difficult to complete. It could take a few attempts before you get it right, so keep trying! If you're stuck, remember the *What to Do When You Get Stuck* section in the introduction of this book. If you're really *really* stuck, take a sneak peek at the next level.

Optional Further Exercises

- Does your program handle having more or less than three choices per question? If not, write it so that it can handle an array of choices with any number of elements. For example, allow the first question to have two choices, and the second question to have five choices.
- Add another question hash to the array of questions. Does your program handle the new question? If not, rewrite it so that it does.

Level 6: Functions

In this level we will start making our own functions. We will gain a better understanding of how arguments and return values work, and also learn how to break up long, complicated pieces of code into smaller, simpler functions – a process known as factoring.

Functions Refresher

There are some definitions related to functions back in *Level 3*. Those will be important to this level, so here they are again:

Value
: *(noun)* anything that can be stored inside a variable.

Assign
: *(verb)* to put a value into a variable. Example: *I assigned the value `"Tom"` to the `name` variable.*

Argument
: *(noun)* a value that goes into a function. Arguments are also known as *parameters*. Functions are said to "take" arguments. Example: *the `puts` function takes one argument.*

Return value
: *(noun)* a single value that comes out of a function, when that function has finished running.

Return
: *(verb)* to provide a return value. Example: *the `gets` function returns a string.*

Call
: *(verb)* to run a function. Example: *I called the `gets` function.* Calling a function is also known as "running", "executing", or "invoking" a function.

Function call
: *(noun)* code that calls a function. Example: *`puts("hello")` is a function call.*

Function
: *(noun)* a named piece of code that is not run until it is called. It takes zero or more arguments, and may return a single value.

As an example, let's consider the function call `String(3)`. The function has a named `String`. The integer value `3` is going into the function as an argument, and the function will return the string value `"3"`.

After the arguments go in, but before the return value comes out, some code gets run. We've never actually *seen* the code inside the `String` function, but it does exist. The authors of the Ruby programming language wrote this code, and it exists somewhere on your computer, inside your installation of Ruby. That code gets run when we call the `String` function, and it creates the return value.

Writing Functions

Let's dive straight in and make a function called `sword_puts`:

```
# Takes one string argument
def sword_puts(msg)
  puts("o==[::::::::>   " + msg + "   <::::::::]==o")
end

sword_puts("Round 1")
sword_puts("FIGHT!")
```

If you run the code, you will see that the `sword_puts` function is like the normal `puts` function except that it has a cool sword on each side!

Making a new function starts with `def`, which is a Ruby keyword like `if` and `while`. Making a function is also known as *defining* a function, and the `def` keyword is short for something like "define function."

Next comes the name of the function: `sword_puts`. Function names are like variable names – we get to make them up. They have the same restrictions that variable names have, except there are two more valid characters: exclamation marks (!) and question marks (?). For example, `lets_go!` is a valid function name, but an invalid variable name.

Then come the parenthesis with `msg` inside. This is not a function call, although it looks like one. Inside these parenthesis are the arguments, but they are argument *variables* not argument *values*. When the function gets called, the argument values get assigned to these variables, and then the variables can be used by the functions code. Inside the `sword_puts` function, the `msg` variable is being joined with two other strings.

The first call to `sword_puts` is `sword_puts("Round 1")`. The argument value is `"Round 1"`, which gets assigned to the argument variable `msg`, and then the code inside the function is run. The next call to the function is exactly the same, except that the value `"FIGHT!"` gets assigned to the `msg` variable. The argument variables only exist within the function. If we tried to use `msg` outside of the function, it would crash with an error that says "undefined variable."

The `sword_puts` function doesn't return anything, it just calls `puts` to display a string.

Returning from Functions

Let's look at a function that has a return value:

```
# Takes zero arguments, and returns an integer
def integer_gets()
  puts("Enter an integer:")
  i = Integer(gets())
  return i
end

i1 = integer_gets()
i2 = integer_gets()
puts(String(i1) + " times " + String(i2) + " equals " + String(i1 * \
i2))
```

The `integer_gets` function takes no arguments, just like the `gets` function. Every time it is called, it asks the user to enter an integer, then it gets an integer from the user and returns that integer. To specify the return value of a function, we use the `return` keyword followed by whatever value we want to return. It doesn't have to be a variable. We could get rid of the `i` variable and just write `return Integer(gets())`, and it would work exactly the same.

The `return` keyword not only specifies the return value, it also stops the function from running. This becomes apparent when there are more than one `return` keywords in the function, like so:

```
# Takes one string argument, and returns a string
def greeting(country)
  if country == "Australia"
    return "G'day"
  end

  if country == "France"
    return "Bonjour"
  end

  if country == "Japan"
    return "Konichiwa"
  end

  return "Hello"
end

puts(greeting("Australia"))
puts(greeting("Japan"))
puts(greeting("Britain"))
```

The code above outputs this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
G'day
Konichiwa
Hello
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

For the `greeting("Australia")` function call, the condition `country == "Australia"` is `true`, so control flow reaches the `return "G'day"` line. At that line, the function stops running.

For the `greeting("Britain")` function call, all the conditions are `false`, so control flow skips all the `return` lines until it gets down to `return "Hello"` at the bottom.

If a function doesn't have a return value, but we want to stop it from running at a certain point, we can just use `return` by itself.

```
# Takes no arguments, returns nothing
def no_return_value()
  puts("I don't have a return value")
  return
  puts("This will never get displayed")
end

no_return_value()
```

Function Control Flow

When a function is called (e.g. `greeting("Australia")`), control flows *into* the function. That is, the next code to be run will be the code inside the function. The place where the function was called from is frozen – it stops running, and waits for the function to return.

The code inside the function runs from top to bottom, like usual, until it either runs the `return` keyword, or there is no more code left to run. At that point, control flow *returns to the caller*. That is, the function stops running, and the wherever the function call happened unfreezes and continues to run.

As an example, consider this line of code:

```
puts(greeting("Australia"))
x = 5
```

There are two function calls here. The first function call is `greeting("Australia")`, and the second function call is to the `puts` function.

The call to `greeting` happens first, because the return value from this function call will be used as the argument to the `puts` function call. Control flow goes into the `greeting` function. The code inside `greeting` gets run until it gets to `return`. Control flow then returns to the place that the function call happened: the first line of the example code above.

Next, `puts` gets called with the return value from `greeting` as its argument. Control flows into the `puts` function. Some code gets run inside `puts` to display the argument, and then control flow returns to the caller once again.

Once control flow returns from `puts`, there is nothing left to run on that line of code, so control flow continues onto the next line of code: `x = 5`.

This is how the `gets` function freezes our programs. When we call `gets`, control flows into it. Inside the `gets` function, there is a loop that keeps looping until the user presses the enter key. Control flow has left our code and is inside the loop, so our code is frozen. After the user presses the enter key, the loop stops looping and the `gets` function returns a value. Control flow returns to our code, which unfreezes and continues to run.

Arguments

So far, we have only seen functions that take a single argument (e.g. `puts`) or no arguments (e.g. `gets`), but functions can take any number of arguments. Let's look at a function that takes two arguments:

```
# Takes two integers, and returns an integer
def plus(int1, int2)
  return int1 + int2
end

puts(plus(1, 3))
puts(plus(10, 5))
```

In the function *definition*, the argument *variables* are separated by commas. In the function *call*, the argument *values* are also separated by commas.

The argument values get assigned to their variables in order. The first value is assigned to the first variable, the second value to the second variable, and so on.

Variable Scope

Argument variables are only available within their function. Once the function has returned, those variables no longer exist. If you try to access them from outside the function, it will crash. Here is an example:

```
def hello(name)
  puts("Hello, " + name)
end

hello("Tom")
puts(name) # crashes here when trying to use name
```

The above code crashes with this output:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
Hello, Tom
game.rb:6:in `<main>': undefined local variable or method `name' for\
 main:Object (NameError)
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!!
```

This error says that, on line six, we tried to use the `name` variable but that variable didn't exist. This is because `name` is an *argument variable*, which means that it only exists inside its function.

Similarly, variables outside of a function can not be accessed from inside a function. Here is another example that crashes:

```
name = "Tom"

def hello()
  # crashes here when trying to use name
  puts("Hello, " + name)
end

hello()
```

This crashes with almost the same error message:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>
game.rb:4:in `hello': undefined local variable or method `name' for \
main:Object (NameError)
        from game.rb:7:in `<main>'
!!!!!!!!!! Ruby has crashed !!!!!!!!!!!!!!!
```

The error says that, on line four, inside the `hello` function, we tried to use `name` but it didn't exist. That is because `name` was declared outside of the function, so it is not available within the function.

This is a concept called *scope*. Variables have a certain scope, and that scope determines where the variable can be used. If a variable is *in scope*, it means it is accessible. If it's *out of scope*, then it is not accessible.

For variables created outside of any function, their scope is any code that is outside of a function. Any code inside a function is out of scope, for these variables.

Argument variables are scoped to their function. All the code inside their function is within scope, and everything else is out of scope. This also applies to normal variables that are created inside a function – for example, this code also crashes:

```
def function_with_a_variable()
  name = "Tom"
  puts("Hello, " + name)
end

function_with_a_variable()
puts(name) # crashes here when trying to use name
```

The `name` variable is not an argument variable, but it is created inside a function, so it has the same scope as an argument variable. That means that it is not accessible from outside the function.

These scoping rules have some consequences. Firstly, if we have a value outside of a function, but we want to use it inside a function, we must pass it in as an argument. Secondly, if we have some value inside a function, and we want to get it out of the function, it must be the return value. Think of it like this: the only values that go into a function are the arguments, and the only value that comes out is the return value.

Composing Functions

You may have noticed that functions can call other functions. For example:

```
def chicken()
  return "A chicken"
end

def ducken()
  return chicken() + " inside a duck"
end

def turducken()
  return ducken() + " inside a turkey"
end

def display_turducken()
  puts(turducken())
end

display_turducken()
display_turducken()
display_turducken()
```

Every time that `display_turducken` is called:

- `display_turducken` calls `turducken`
- `turducken` calls `ducken`
- `ducken` calls `chicken`

This is called *function composition*, and it is how all complicated software is made – functions calling functions calling functions. Each individual function might be fairly small and simple, but they call other functions, which call other functions, and so on, to do complicated things.

This is a lot like composite values. Simple values like integers and booleans are *not* able to represent complicated data by themselves. But those values can be combined with composite values, like arrays and hashes, to represent *any* kind of complicated data.

In the same way, no individual function can do all the complicated things in a typical program or app. But lots of separate functions can call each other, combining together to make *any* kind of complicated software. **All software is made up of functions that call other functions**. The operating system that you are using (Windows, OS X, Linux, etc.) is made from hundreds of thousands of different functions.

Factoring the Previous Level

In the boss project of the previous level, you may have noticed that the more code you wrote, the harder it was to understand all of it, and the easier it was to make mistakes. To combat this, we take parts of the code and make them into simpler functions – a process called *factoring* or *decomposition*.

Let's look at the process of factoring by applying it to the boss project from the previous level.

For starters, the following `puts_choices` function would have been useful:

```
# Takes one argument: an array of strings
def puts_choices(choices)
  index = 0
  for c in choices
    puts("  " + String(index) + ". " + c)
    index = index + 1
  end
end

puts_choices(["Paris", "Abuja", "Berlin"])
```

This `puts_choices` function takes an array of strings as an argument, and it displays the strings nicely with their index.

The following `get_choice` function would have also been handy:

```
# Returns an integer
def get_choice()
  puts("Which option do you choose?")
  return Integer(gets())
end

choice = get_choice()
puts("You chose " + String(choice))
```

We could then combine `puts_choices` and `get_choice` into a new function that asks a whole question:

```
# Takes one argument: a question hash
# Returns the users score (an integer) for that question
def ask_question(question)
  puts()
  puts("Question: " + question["question"])
  puts_choices(question["choices"])

  choice = get_choice()
  if choice == question["correct_index"]
    puts("That is correct!")
    return 1
  else
    puts("That is incorrect.")
    return 0
  end
end

q = {
  "question" => "What is the capital of Nigeria?",
  "choices" => ["Paris", "Abuja", "Berlin"],
  "correct_index" => 1
}
```

```
score = ask_question(q)
puts("You scored " + String(score))
```

The `ask_question` function takes a hash argument. Using values in the hash, it displays the question, gets the users choice, and returns a score for that question.

The final step is to make a function that asks an array of questions, and keeps score:

```
# Takes one argument: an array of question hashes
def ask_quiz(questions)
  score = 0
  for q in questions
    score = score + ask_question(q)
  end

  puts()
  puts("Your final score is " + String(score) + "/3")
end

quiz = [
  {
    "question" => "What is the capital of Nigeria?",
    "choices" => ["Paris", "Abuja", "Berlin"],
    "correct_index" => 1
  },
  {
    "question" => "In what year was Julius Ceasar killed?",
    "choices" => ["44 BC", "192 AD", "0 AD"],
    "correct_index" => 0
  },
  {
    "question" => "Which artist wrote the song 'Life on Mars'?",
    "choices" => ["Freddy Mercury", "Jimmy Barnes", "David Bowie"],
    "correct_index" => 2
  }
]
```

```
ask_quiz(quiz)
```

After factoring the code into functions, we can run the whole quiz with just one function call: `ask_quiz(quiz)`. Each individual function is fairly simple, but when combined they make a whole quiz.

This isn't the only way to factor the code. The same functionality could be achieved with different functions – functions with different names, that take different arguments, and return different values. I encourage you to try and factor your own code from the previous level, making different functions than I did.

For now, when you factor your own code, just try to pick functions that you feel are simple and understandable. Focus on making your code easier to work with. When a piece of code becomes too difficult to understand, try breaking it down into simpler functions. Run the functions individually and `puts` the return values to test that they are working as expected.

Mutating Arrays and Hashes

We've seen how to access elements in arrays:

```
letters = ["a", "b", "c"]
puts(letters[1])
```

And how to access values in hashes:

```
numbers = { "one" => 1, "two" => 2 }
puts(numbers["two"])
```

In addition to accessing values, we can also *change* those values. Here is an example of changing the second element of an array:

```
letters = ["a", "b", "c"]
letters[1] = "BEE"
puts(letters)
```

The code above outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
a
BEE
c
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Here is an example of changing a value in a hash:

```
numbers = { "one" => 1, "two" => 2 }
numbers["one"] = 3
puts(numbers)
```

Which outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
{"one"=>3, "two"=>2}
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

This is called *mutation*. Arrays and hashes are *mutable* values, because the elements/-values that they contain can be changed. As shown in the example above, hashes and arrays can be mutated by writing square brackets (like when accessing the value) immediately followed by an equals sign and the new value.

It *looks* like a combination of accessing a value and assigning a variable, but it isn't. The following code demonstrates how mutation is different to variable assignment:

```
letters = ["a", "b", "c"]
other_letters = letters
other_letters[1] = "BEE"

puts("letters:")
puts(letters)
puts()
puts("other_letters:")
puts(other_letters)
```

Which outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
letters:
a
BEE
c

other_letters:
a
BEE
c
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

Notice how *both* `letters` and `other_letters` got mutated. This is because there is *only one* array value. The single array has just been stored in two different variables. The variables never actually change – only the array value changes.

The equals sign usually means variable assignment, but when it directly follows square brackets like above, it means mutation.

Here is an example of a function that mutates its argument:

```ruby
def mutate(things)
  things[0] = "TURTLES"
end

animals = ["cat", "dog", "bird"]
mutate(animals)
puts(animals)
```

Which outputs:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>
TURTLES
dog
bird
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

The `animals` variable holds an array. This array gets passed into the `mutate` function as an argument. The function mutates the array by changing the first element to be `"TURTLES"`. Then the array gets displayed with `puts`. All of this happens to the same array – there is only one.

The array is created and assigned to the `animals` variable first. The same array also gets assigned to the `things` argument variable when the `mutate` function gets called. The two variables contain the exact same array, so mutating one affects the other.

Boss Project: Tic Tac Toe

Make a simple tic tac toe game. The output should look like this:

```
>>>>>>>>>> Running: game.rb >>>>>>>>>>>>>>>>

1 | 2 | 3
---------
4 | 5 | 6
---------
7 | 8 | 9

Place an X (or 99 to exit)
1

X | 2 | 3
---------
4 | 5 | 6
---------
7 | 8 | 9

Place an O (or 99 to exit)
2

X | O | 3
---------
4 | 5 | 6
---------
7 | 8 | 9

Place an X (or 99 to exit)
3

X | O | X
---------
4 | 5 | 6
---------
7 | 8 | 9
```

```
Place an O (or 99 to exit)
99
<<<<<<<<<< Finished successfully <<<<<<<<<<
```

To keep this project simple, you *don't* need to detect when the game is over – just let the user enter `99` to exit at any time.

Represent the tic tac toe board as an array. The board has nine positions, so the array will need nine elements. The first three elements are the top row, the second three are the middle row, and the last three are the bottom row. When the user makes a move, mutate the board array.

Use this project to practice factoring your code into functions. You will need to pass the board array to functions as an argument. You choose what functions to make, but if you get stuck, consider these functions:

- `puts_board(board)`
- `do_one_turn(board, player_mark)`
- `opposite_player_mark(player_mark)`

If you are having trouble, reread the *What To Do When You Get Stuck* section of the introduction of this book.

Optional Further Exercises

- Give the user the option to restart the game. You will need to mutate the board array elements to reset them to their original values, or replace the whole board with a new array.
- Can a player overwrite a previous player's turn by entering the same position number? Try to prevent players from doing this. You will need another loop that keeps looping until the user enters a valid position number. Consider putting the loop into a new function, named something like `get_valid_position`.
- Let the user play against the computer. When it's the computers turn, pick a valid position at random.

- Try to detect when the game is finished, and display the winning player or say if the game was a tie. This is more difficult than it sounds! You may need to investigate how use the *logical AND operator (&&)*, which has not been explained yet in this book.

Conclusion

If you have completed all of the levels, then congratulations! Most people don't make it this far. You are now a LVL 6 Code Conjurer.

There is still plenty more to learn, though. Check out the advanced examples below, to demonstrate what you could be making with a little extra learning.

Advanced Example: Top of Reddit

Here is a little program that fetches and displays the current top submission on Reddit:

```
require "json"
require "open-uri"

# load a big composite value for the front page of reddit.com
front_page = JSON.load(open("https://www.reddit.com/.json"))

# get the hash for the top submission
top_submission = front_page["data"]["children"][0]["data"]

puts("The current top submission on reddit is:")
puts()
puts("Title: " + top_submission["title"])
puts("Author: " + top_submission["author"])
puts("Subreddit: /r/" + top_submission["subreddit"])
puts("Score: " + String(top_submission["ups"]))
puts()
puts(">>> Press enter for the full submission hash <<<")
gets()
puts(top_submission)
```

The first few lines will be a bit unfamiliar to you, but you should be able to understand most of the code.

Inside that `front_page` variable is a big composite value that has been fetched from Reddit. It's a hash that contains lots of other hashes and arrays. The `top_submission` variable is a hash, too.

To reiterate *Level 5*, this is how *all software* represents data, and Reddit is no exception. It's all just simple values like integers and strings, collected into composite values like hashes and arrays. Look at the URL in your browser (https://www.reddit.com/.json) to see the big composite value that Reddit provides in JSON format.

Advanced Example: a 2D Game

Making text-based games is a good way to learn the basics of programming, but it's much more fun to output graphics and audio!

You can download an example 2D game project from here: 2d_game.zip[6]

Unzip the folder, drop your code runner file into it, and run it like you do with your other projects. In order to run, it needs to download some game code called Gosu[7], but the code runner file should be able to do this automatically. If you are on Windows, it may ask you whether Ruby is allowed to access the internet.

All the code is in `game.rb`, so open up that file and have a look at it. This code uses advanced features of Ruby that haven't been covered in this book, so don't worry if it doesn't appear familiar to you yet.

You are close to understanding this code! I encourage you to tinker with it. Delete lines and change values just to see what happens. Use `puts` to display variables, and watch the output while you use the program.

Functions for creating a window, drawing images, playing sounds, and responding to the keyboard all come from Gosu. You can read about what Gosu provides in its documentation[8].

[6] http://www.programmingforbeginnersbook.com/downloads/2d_game.zip

[7] https://www.libgosu.org/

[8] https://www.libgosu.org/rdoc/frames.html

The code uses a lot of array functions that aren't explained in this book. The documentation for all Ruby array functions can be found here[9].

I very much wanted every boss project in this book to be a 2D game. Unfortunately, that would be too difficult for readers who have never written code before. There are a lot of programming concepts to learn before you have the skills to make a 2D game, but after completing this book, you should be close! Maybe the next book will be all about 2D games.

If you want to try 2D game programming, despite the difficulty level, you might be interested to watch these videos of a Flappy Bird clone created from scratch in Ruby[10].

What to Learn Next: OOP

This book teaches a style of programming called *procedural programming*. Procedural programming is an old style, but it is fundamental. Procedural concepts – like looping, branching and functions – are important because they are used in almost all programming languages. The way the concepts are written in code is different for every language, but they work the same way.

Currently, the most popular style of programming is called *object-oriented programming*, which is often just abbreviated as *OOP*. The concepts of OOP are built *on top* of procedural programming concepts, so an understanding of OOP first requires an understanding of procedural programming. After completing this book, you will be ready to learn about OOP.

The 2D game project above uses a lot of OOP concepts, and that is why you will not be able to understand parts of the code yet. But all the things explained in this book are present:

- Strings
- Integers
- Booleans

[9] http://ruby-doc.org/core-2.2.3/Array.html

[10] http://www.tomdalling.com/blog/ruby/fruity-bat-flappy-bird-clone-in-ruby/

- Arrays
- Random numbers
- Lots of conditionals
- A few function definitions
- Lots of function calls
- Lots of variables, although some are in capitals (constants), and others start with `@` (instance variables)
- Loops, although types of loops not covered in this book (they require some OOP knowledge)

Ruby is an OOP language, and it is one of the *best* OOP languages, in my opinion. I suggest that you stick with Ruby, and start learning about OOP next.

Other Resources

Here are some online resources for learning Ruby:

- Try Ruby: http://tryruby.org/
- RubyMonk: https://rubymonk.com/
- Codecademy: https://www.codecademy.com/tracks/ruby/

However, the best way to learn is by trying to make things yourself. Make another text-based game, or attempt a 2D game if you want a challenge. This will force you to learn new things as you add features. Programming is not something that can be learned just by reading, or watching videos. Books and videos help, but you must practise, experiment, and create things, in order to learn. Whatever you make, start with something small, then add features to it. Programming is difficult to learn, and it is easy to become overly frustrated if you are too ambitious.

Be careful of places like StackOverflow, and /r/programming on Reddit. They tend to contain a few grumpy old software developers who have forgotten how hard it was to learn their first programming language. The majority of people there are perfectly nice, but some of them will not be very friendly towards beginners. Try forums that are specifically for learners, such as /r/learnruby[11]

[11] http://reddit.com/r/learnruby

Send Me Your Programs!

You have the skill to make software now. If you make a program and you want to share it, send it to me! I am interested to see your creations.

Attach your Ruby code to an email addressed to tom@tomdalling.com. Alternatively tweet to @tom_dalling[12] with a pastebin[13] or GitHub Gist[14] link.

[12] http://twitter.com/tom_dalling
[13] http://pastebin.org
[14] https://gist.github.com/

Acknowledgements

This book would not have been possible without the help of numerous people. Here are some of them:

- Alpha testers: Nick Williams and Danae Williams
- 2D game assets: Kenny "Asset Jesus" Land[15]
- 2D game library: Julian Raschke, for lovingly maintaining the Gosu[16] library
- Cover graphic: designed by Freepik.com[17]
- A kick in the butt: Amy Hoy and Alex Hillman, for running the Ship by September Challenge[18]
- Caffeine: Coffee Dominion in Townsville, for what is possibly the best coffee in Australia

Thank you all for helping me to complete this project.

[15] http://kenney.nl/
[16] https://www.libgosu.org/
[17] http://www.freepik.com/
[18] https://unicornfree.com/2015/ship-by-september