Table of Contents

learn2love

Current progress:

- Chapter 1 - Programming basics ✓
- Chapter 2 - Introducing LÖVE ✓
- Chapter 3 - Programming in depth (in progress)
- Chapter 4 - LÖVE in depth (to do)

View as a webpage: link

Download in ebook format: pdf - epub

What is this book?

This book teaches programming from the ground up in the context of Lua and LÖVE. It teaches basic computer science and software building skills along the way, but more importantly, teaches you how to teach yourself and find out how to go about solving a problem or building a solution. Tools come and go, so the goal is to teach things of value with less focus on the programming language and other tools used to build the software. I have been programming since 2007, focusing on teaching myself best practices. Along the way I have found a lot of good and bad tutorials on the right and wrong way to build things and I want to help others avoid getting stuck like I did.

Who is this for?

- **Any age group.** Kids too, with a bit of demonstration, help and encouragement!
- **Anybody that wants to learn basic computer science.** This book will touch on several computer science subjects in order to build programs.
- **Anybody that wants to learn to program.** No prior skills or knowledge required.
- **Anybody that wants to learn to make a game.** Making games are fun and require learning many things along the way. We'll build a few through this book.
- **Anybody that wants to learn Lua.** Although we won't dive into the advanced features of the language, we will gain a large understanding on how the language works in order to actually build some things. There are already online guides and references covering some of the more advanced topics. For experienced programmers wanting to learn Lua, the Programming in Lua book may be sufficient.

Author and contributors

- jaythomas: Original author

- JimmyStevens: Edits and suggestions in chapter 1 & 2
- rm-code: Chapter 2 getting started
- ValentinChCloud: Chapter 3 primitives and references
- valuefit: Chapter 2 bug fix [1]

Contributing

- Issues, comments, and suggestions can be made using the GitHub issues page.
- To download, build, and run the book or any code examples use the "Clone or download" button on the main repository page.

For developers and the curious

Feel free to submit a pull request. The documentation is built using NodeJS. If you wish to run the documentation for local development purposes, install nodejs then run these commands from within the `learn2love` directory you downloaded:

```
npm install # Downloads build tools to the a "node_modules" folder inside the d
npm start   # Creates a local web server to where you can visit the link http:/
```

Once the local web server is running, any edits you make to the pages will rebuild the book and reload the page you're viewing.

Chapter 1: Programming Basics

The goal of this chapter is to teach the most necessary building blocks of programming. By the end of the chapter you will be be able to build basic programs which we will apply with exercises in the following chapters.

Interactive coding

What's a REPL?

Programming doesn't take much effort beyond loading up a REPL and just typing. What is a *REPL*? It's an interactive window you can type code into and it spits out the results on screen when you hit enter. It stands for **R**ead-**E**valuate-**P**rint-**L**oop. These are the 4 things the REPL does:

1. Read the code that was just typed
2. Evaluate, or process the code down into a result
3. Print, or spit out the result
4. Loop... do everything again and again until the programmer is done

It's actually simpler than it sounds. Let's go to a website with a REPL and try it out: https://repl.it/languages/Lua

You will see two window panes on the website: a light side on the left and dark side on the right. The right-side is the REPL and is what we're interested in for now. It has a lot of information that isn't necessarily useful to us at the moment. Something similar to this:

```
Lua 5.1  Copyright (C) 1994-2006 Lua.org, PUC-Rio
[GCC 4.2.1 (LLVM, Emscripten 1.5)] on linux2
```

This is just telling you what programming language this REPL is loading, in this case, Lua. If you click inside the window pane and start typing you will see your text appear.

Let's try typing some code for the REPL to **R**ead. You already know some code if you know arithmetic. Type:

```
2 + 2
```

Then hit ENTER and immediately the REPL will **P**rint out:

```
=> 4
```

A lot happened very quickly. After hitting ENTER, the REPL, **R**ead the line `2 + 2`, it **E**valuated the value of that statement to be `4`, it **P**rinted 4 on the screen for you, then **L**ooped back to a new line to await your next command. Try out some more arithmetic. Multiplication:

```
2 * 3
```

Subtraction:

```
2 + 2 - 4
```

Division:

```
6 / 2
```

You can use parenthesis to tell it which order to do the operations:

```
(2 + 2) * (3 + 1)
```

Which gives different results than:

```
2 + 2 * 3 + 1
```

If you give the REPL a single number:

```
12
```

It will give you back `12`, because this can't be simplified down any further.

You can also do exponents using the `^` (caret) symbol:

```
2^4
```

Numbers are a type of data, and `+`, `-`, `/`, `*`, `^`, `%` are operators. Statements such as `2 - 2` and `23 * 19` are all operations.

One last arithmetic operation we'll cover is modulo, which is done with the modulus operator. The modulus operator is represented in most languages as a `%` (percent) symbol:

```
8 % 3
```

Modulus operations aren't seen in grade school classrooms as often as the rest, but are quite common in software and computer sciences. The way it works is you take the 2nd number and subtract it from the bigger number as many times as possible until the 2nd number is bigger than the 1st. The result is what's left of the 1st number. With `8 % 3`, if you keep subtracting `3` from `8` then you end up with `2` left.

A real world example is time elapsing on an analog clock. Imagine the face of a clock with the hour hand on noon. If 25 hours pass then the hour hand goes all the way around twice and ends on 1. That would be equivalent to writing:

```
25 % 12
=> 1
```

The hour hand resets every time it passes 12, so `13 % 12`, `25 % 12`, and `37 % 12` would all equal `1`. Likewise, `10 % 4` results in `2` because 4 goes into 10 twice, and leaves a remainder of 2.

Exercises

- Try typing different modulo operations in and guessing what the answer will be.
- Try using negative numbers (`-3 + -2`).
- Try using a set of parenthesis inside another set of parenthesis. Does it behave as you expect?
- After running through all the exercises press the 'up' key in the REPL. What happens and how can this speed up your work?

Strings

Numbers are one type of data that can be operated on. Let's explore another data type within the REPL. Take a set of quotes and put some text in it and hit ENTER:

```
"hello"
```

The REPL will print `hello` back to you. This is a string. A string is a set of characters (letters and symbols) *stringed* together as one single piece of data. This string is made of 9 characters:

```
"H-E-L-L-O"
```

Like numbers, there are operators to make strings play with each other. The *concatenate* operator (`..`) combines strings together:

```
"hello" .. "world"
```

What's the result? Notice that the resulting string has no space between the two words. If you wanted a space, you would have to put one in the quotes to be part of the operation:

```
"hello " .. "world"
```

You could even make a separate string with the space in it:

```
"hello" .. " " .. "world"
```

Strings can have any characters in them that you want.

```
"abc" .. "123"
```

```
"Япо́нский" .. "ロシア語!!"
```

Exercises

- Try using an arithmetic operator on strings `"hello" / "world"` . What happens?
- Try using the concatenate operator (`..`) on numbers (`1 .. 1`).

Nil and variables

Data, or the lack thereof

Humans have different ways of representing a lack of data. If there are no sheep to count then we have zero sheep. If there are no words on a page then the page is blank. In a computer we may represent the number of sheep as `0` or the missing words on a page as an empty `""`. These are still data though... a number and a string. In software when you want to represent a lack of data we have:

```
nil
```

Sometimes called `null` or `undefined` data in other languages. It's seemingly useless. You can't use operators on `nil`.

```
nil + nil
```

This will print an error like it did when you tried doing arithmetic on strings. Let's take a look at variables and we'll discover the purpose of `nil`.

Variables

Sometimes you want to write out data, but you want that data to be easy to change. Variables let you give data a name to reference. Here's an example to try:

```
name = "Mandy"
"hello my name is " .. name
```

Since you told it what `name` is, it knows what value to add to the string `"hello my name is "`. If you type:

```
name
```

...and hit ENTER, it will print out the value that belongs to this variable to remind you. The `=` (equal) sign tells Lua that you want to assign a value to the given name/variable. You can change the value of a variable and get different results:

```
name = "Jeff"
"hello my name is " .. name
```

Assignment isn't the same as it is in Algebra. You can change the value of a variable multiple times. We can tell `name` that it equals itself with some additional information *concatenated* to it:

```
name = "abc"
name = name .. "def"
name
```

You can assign any type of data to a variable, including numbers:

```
name = "Jeff"
age = 16
"hello my name is " .. name .. " and I am " .. age .. "."
```

You can change numbers after assignment too:

```
age = 16
age = age * 2
"my age doubled is " .. age
```

So, what if you type in a made up variable name?

```
noname
```

You will see it has `nil`, or no data yet. If you try to use `nil` in your string operation you will get an error:

```
"hello my name is " .. nil
```

```
[string "return "hello my name is " .. nil"]:1: attempt to concatenate a nil va
```

```
"hello my name is " .. noname
```

```
[string "return "hello my name is " .. noname"]:1: attempt to concatenate globa
```

Try assigning a value to a variable name:

```
best_color = "purple"
```

then assigning that variable data to another:

```
worst_color = best_color
worst_color
```

You'll see that both variables now have the value `"purple"`.

Variables can have names made up of letters, numbers and underscores (`_`). Variable names cannot begin with a number though, otherwise it will think you're trying to type in number data. Here's some examples of valid variables:

```
my_dog = "Poe"
myDog = "Zia"
DOG3 = "Ember"
```

Exercises

- Try out different variable names. Try a few invalid variables names too just to see what the error message looks like. It's important to see error messages and understand them. They help you understand how a program breaks so you can fix it.

Using functions

Most programming languages come with some variables already defined for us. Lua has many, so let's type one in and hit ENTER to see what the value is:

```
string.reverse
```

```
=> function: 0x2381b60
```

Oh my. So "function" is another data type, but what is `0x2381b60`? It's just telling you where in the computer's memory that function exists, just in case you wanted to know. Functions work very differently than numbers in strings. Essentially functions are pre-defined instructions that tell the program how to do different things. They take data and *return* back different data. Let's see how to give this function data:

```
string.reverse("hello")
```

```
=> olleh
```

At the end of the function's variable name, `string.reverse`, we type a set of parenthesis, `string.reverse()`, and put inside the parenthesis some data we want changed (`string.reverse("hello")`). Making the function run is often called *invoking* the function. Having a function that reverses text in a string for us can be useful, and we can capture the return value (the results) of the function using a variable. Try it out:

```
greeting = "hello, how are you?"
backwards_greeting = string.reverse(greeting)
backwards_greeting
```

```
=> ?uoy era woh ,olleh
```

It should be obvious from the name what that function's purpose is. How about this one?

```
string.upper("hello, how are you?")
```

Now try capturing that value by assigning it to a variable:

```
greeting = "hello, how are you?"
shouting_greeting = string.upper(greeting)
crazy_greeting = string.reverse(shouting_greeting)
```

We can get crazier. How about invoking a function when invoking another function??

```
string.reverse(string.upper("hey"))
```

What's happening here is the string is being uppercased by `string.upper` but then the value from `string.upper` is being reversed by `string.reverse` as soon as it is done. It's just like in arithmetic when you have nested parenthesis. The inner-most parenthesis are resolved before doing the outer-most parenthesis.

Let's try one more function. This function has two parameters, meaning it accepts two pieces of data which it requires to work properly.

```
math.max(7, 10)
```

When giving more than one piece of data to a function, you need to put a comma (`,`) between the parameters

These are great functions, but wouldn't it be great if we could make our own? We'll give it a shot in just a few pages.

Exercises

- See if you can figure out what `math.max` does. Give it different numbers and examine the result.
- There is another function called `math.min` that also takes two numbers. What does it return?

Comments in code

Sometimes we might want to write a comment in our code– an explanation to a friend or our future selves on what the purpose of some code is. Perhaps we want to write a note to ourselves to change something later. Comments work very similarly in different languages so they're pretty easy to read even if you don't understand the programming language or the code itself. Lua denotes a comment as `--` and any text that follows it:

```
1 + 1
-- This is a code comment
1 + 2
-- This is another line of comments
3 + 4
```

These comments will be completely ignored by the computer and are meant for the human to read. Comments can also be on the same line as code. The computer will just ignore the rest of the line when it sees a comment starting.

```
1 + 1 -- This is my comment. This code adds some numbers together in case you
```

You will see comments appear in future example code, so don't let it surprise you!

Scripting and printing

Looking back at the website, (you bookmarked it, right?) we have been using the REPL window pane on the right, but haven't talked about the pane on the left. This window is just a text editor. Instead of running the program with each line you type, it allows you to write multiple lines of code before executing it all. Let's try typing something in it. Once you are done typing all the code you can click the "Run" button.

```
number = 4
number = number + 1
```

But when you click run, nothing happens. So let's provide another *statement* to our program.

```
number = 4
number = number + 1
print(number)
```

Now when you click Run, the text `5` appears in the right-hand pane. When you told it to run, it read and evaluated each line of the code in sequence.

You can print any type of data, not just numbers:

```
print("hello")
```

Remember those other functions we used before? You can write those inside of a print statement.

```
print(string.reverse("hello"))
```

We can even print functions themselves:

```
print(string.reverse)
```

And see a memory location of where that function exists:

```
function: 0x1795320
```

This can serve as a unique indentity for that function, which we'll see more of in a later page.

Lua provides this `print` function to allow us to poke around while our program is running. We can print as many things as we want.

```
print("hello")
print("world")
```

Exercises

- When we pass data into a function, it is called an argument. We passed 1 argument into `print` but it can pass in two, or three, or more. What does it look like when you print multiple arguments?
- When using a text editor along-side the REPL you can run the code without the mouse by pressing 'command + enter' on Mac and 'Ctrl + Enter' on Windows. Does this speed up your learning?

Making functions

Functions are the third data type we've seen. We've accessed some variables where functions were defined for us and had a blast using them (I know I did). Functions are the building blocks of software. You can compose them then snap them together like Danish plastic blocks. It takes time to understand how they work and much longer to master their inner power. So without further ado, let's see what they actually look like:

```
function()
  return 4 + 4
end
```

Type it out in the text editor window and let us break this down line by line and word for word. Whenever we type `function()` we are beginning a new function. The 2nd line is the body of our function where things happen. The body of the function can be many lines long. The body of the function could also be empty (but that's not very useful). On the last line of the function body we write `return` which tells our function to stop running and to "return" data back to the main program. Then on the third line, we're telling the computer we're done writing our function. In order to use this example function, we should probably use a variable to give it a name:

```
add = function()
  return 4 + 4
end
```

The first bit should be understandable. We *declared* a variable called `add`, then we assigned some data to it on the right of the equal sign. In this case, our function. Now it is ready to use.

```
add = function()
  return 4 + 4
end

result = add()
print(result)
```

```
8
```

We've made our very own function with our very own name for it and even invoked it and got back data! If you instead got an error message, double check what you typed that nothing is missing. Error messages give you a *line number* of where to find the error that crashed the program.

Take a look for a minute at how we invoked our function:

```
add()
```

We typed out the variable name that our function is assigned to, followed by some parenthesis. In those parenthesis is the data that we passed into our function... wait a minute the parenthesis are empty. We didn't pass any data into our function. Whenever we called those other functions we passed in data, like when we passed `"hello"` into `string.reverse("hello")`. What if we modify our line where we invoke our function and give it some data?

```
add = function()
  return 4 + 4
end

result = add(16)
print(result)
```

It seems it always returns `8` no matter what arguments we try to pass in. We need to rewind to the first line of our function and take a close look at this bit:

```
add = function()
```

The `()` at the end of `function()` is where we tell our program how many arguments we are accepting. If the parenthesis are empty, then our function is ignoring all arguments and will likely always return the same result. Let's tweak the function slightly and give it one parameter with the name `a`. Let's also tweak the second line while we're at it:

```
add = function(a)
  return a + 4
end

result = add(16)
print(result)
```

```
20
```

Now when we pass in different numbers, we get different results:

```
add = function(a)
  return a + 4
end

print(add(16))
print(add(12))
```

To complete this function, let's give it a second parameter of `b` and modify the return statement in the function body:

```
add = function(a, b)
  return a + b
end

print(add(16))
print(add(12))
```

If we try and run the code now, we'll get another error:

```
[string "add = function(a, b)..."]:2: attempt to perform arithmetic on local 'b
```

Let's read this error carefully. It is saying inside the square brackets that an error occurred when using the function we defined (`add = function(a, b)...`). To the right of the square brackets it is saying line 2 (`:2`) of our text is the particular location of the crash. To the right of the line number is what happened that made it crash. It tried to perform addition with `a + b` but the value of `b` was nil. We stated that our function requires two parameters now, `a` and `b` , and our program will crash if we try and invoke the function with only one parameter. Let's modify the lines where we invoke the function to give it two arguments each time we invoke it:

```
add = function(a, b)
  return a + b
end

print(add(16, 10))
print(add(12, 2))
```

Great, everything is working again! With the experience of our first, fully-functional function, we can now start treading the waters of this great world.

Exercises

- To get used to writing functions, try writing some complimentary functions named `subtract` , `multiply` , `divide` , or `modulate` (modulus).
- Make a `concatenate` function that accepts 2 strings and returns 1 combined string.
- Try making a function that takes 3 or more parameters.

Booleans

Data types are like elements on the periodic table. The more elements you have the more chemicals can create. Luckily there aren't as many data types as there are elements. In fact we've learned almost all of them. There are only two possible booleans:

```
true
```

and

```
false
```

That's right. And you can assign them to variables just like numbers, strings, nil, and functions:

```
myboolean = true
print(myboolean)
```

The cool thing with numbers and strings is you can use them to create statements that can be evaluated as `true` or `false`. Let me give an example by introducing some new operators. Try these out in the REPL:

```
5 > 3
```

```
=> true
```

```
5 < 3
```

```
=> false
```

"5 *is greater than* 3" is a true statement so it returns a `true` boolean. Naturally, "5 *is less than* 3" is a false statement and returns `false`. We can check to see if two numbers are equal in value:

```
number = 5
number == 5
```

```
=> true
```

By using a double equal (`==`) we can compare the *equality* of two numbers. This also works for strings:

```
"hello" == "hello"
```

```
=> true
```

```
"hello" == "HELLO"
```

```
=> false
```

For strings, often time you will see single quotes `' '` (apostrophe) used instead of regular quotes (sometimes called double quotes) wrapper around the text. Lua doesn't care as long as the text inside both strings are identical. We can prove that with an equality check:

```
'hello' == "hello"
```

```
=> true
```

Anyways, you can also do the inverse of an equality check and check for inequality (if two things are *not* equal):

```
5 ~= 3
```

```
=> true
```

```
"HELLO" ~= string.upper("hello")
```

```
=> false
```

Now let's dig in a little deeper with two more operators. First is the `and` operator:

```
3 < 4 and 4 < 5
```

```
=> true
```

This reads out almost as plain English. 3 *is less than* 4 **and** 4 *is less than* 5. This is a logically sound statement so it evaluates to true. Just to be clear on what's actually going on here though, let's break it down. What we said is being grouped into 3 separate operations:

```
(3 < 4) and (4 < 5)
```

The two sets of parenthesis are evaluated first and internally the computer breaks these two operations down to:

```
(true) and (true)
```

True and true are both true. This sounds silly, but it is indeed logically sound. Let's try one more just to get the hang of it:

```
"hello" == "hello" and 6 > 10
```

Finally, let's try one more operator to put a bow on things. Sometimes we don't care that both operations are correct. We only care if one `or` the other is correct.

```
4 == 10 or 4 ~= 10
```

```
=> true
```

```
1 > 100 or 12 == 12 or "hello" == "bananas"
```

```
=> true
```

As long as one of the operations is correct, the entire statement is logically true. With the introduction of `true` and `false` we've brought in a lot of new operators: "greater than" (`>`), "less than" (`<`), "equal" (`==`), "not equal" (`~=`), "and" (`and`), and "or" (`or`).

Trivia

Booleans get their name from George Boole who invented boolean algebra, which we've just seen a little bit of.

Exercises

- Try writing different statements with all the new operators.
- Try using two `and` operators in the same statement and see if you can make it evaluate to `true` .
- Try out these two bonus operators with some numbers: "greater than or equal to" (`>=`), and "less than or equal to" (`<=`).

Flow control

Typically the computer starts at the top of our script and reads each line down in a sequence. We make the programs jump around with functions in the mix Try this out in the text editor:

```
print("I'm called 1st")

add = function(a, b)
  print("I'm called 5th")
  return a + b
end

subtract = function(a, b)
  print("I'm called 3rd")
  return a - b
end

print("I'm called 2nd")

subtract(16, 10)

print("I'm called 4th")

add(12, 2)
```

We have a function that is saved to the variable `add` but it isn't invoked until further down in the code. So in a sense our program has worked its way down the page then jumped back up to the function and worked its way through the body of the function then picked back up where it was before. In a similar fashion, we can make our program take one path or another depending if the data is `true` or `false`.

```
noise = function(animal)
  if (animal == "dog") then return "woof" end
  return ""
end

print(noise("dog"))
print(noise("rabbit"))
```

Let's analyze this function line by line. The function is called noise and takes an animal name (string) as a parameter. On the next line it says if "animal is dog" is true then return something special. We put an `end` at the end of our statement to make it obvious to the computer. If the statement was false, then `"woof"` does not get returned. Instead an empty string (`""`) gets returned. When we invoke the function with the argument "dog" then we get back "woof!". With "rabbit" we get back silence. Maybe the rabbit doesn't want the dog to hear where she is. Let's make our function more versatile by adding more animals:

```
noise = function(animal)
  if animal == "dog" or animal == "wolf" then return "woof" end
  if animal == "cat" then return "meow" end
  return ""
end

print(noise("dog"))
print(noise("cat"))
print(noise("rabbit"))
print(noise("wolf"))
```

We have branching paths happening within our function. If we were to map out these branches it may look something like:

```
 |
 +--> "woof"
 +--> "meow"
 |
 +--> ""
```

There's no requirement that a statement has to be all written out on one line. Sometimes when doing multiple things inside an *if statement* we may want to put it on multiple lines:

```
if my_age > 17 then
  print("You're an adult!")
  print("Get a job!")
end
```

Similar to functions having bodies, everything between `then` and `end` is considered the body of the if statement. Sometimes it is necessary for our branches to have forks within them. Let's say our function takes a language as a second, optional parameter:

```
noise = function(animal, language)
  if animal == "dog" or animal == "wolf" then return "woof" end
  if animal == "cat" then return "meow" end
  if animal == "bird" then
    if language == "spanish" then return "pío" end
    return "tweet"
  end
  return ""
end

print(noise("dog"))
print(noise("rabbit"))
print(noise("bird"))
print(noise("bird", "spanish"))
```

The if statement for checking if the animal is a bird is 4 lines long. Once we find out that the animal is a bird, while still in the body of the if statement, we stop to check and see if the language is set to Spanish. If it is, we end up inside an if statement within an if statement! Otherwise we'll return `"tweet"` if the language isn't Spanish. Maybe mapping out the paths will clear things up:

```
|
+--> "woof"
+--> "meow"
+-----> "pío"
|   |
|   +--> "tweet"
|
+--> ""
```

Our code can get unreadable very quickly if we start nesting if statements inside each other. Fortunately doing so isn't usually necessary.

Let's talk about another aspect of if statements. Suppose I have two branches of code that are opposite of each other:

```
if daytime == true then
  thermostat = 71
end
if daytime == false then
  thermostat = 68
end
```

Rather than writing this out as two if statements and checking the value of daytime twice, I can take advantage of the keyword `else`:

```
if daytime == true then
  thermostat = 71
else
  thermostat = 68
end
```

That way if `daytime` is not `true`, it will default to the second branch. You could read this off almost like a sentence: "If daytime is true then set the thermostat to 71, otherwise set the thermostat to 68." Not having to check things twice when doing computations saves us time and makes our program run more efficiently. Since `daytime` is a boolean in this case, we don't need to check if it is true or false. We can just pass it to the *if statement* to be checked for `true` / `false` and make our operation even simpler.

```
if daytime then
  thermostat = 71
else
  thermostat = 68
end
```

Better. "If daytime then set thermostat to 71, otherwise set thermostat to 68." There's one more feature of *if statements* we should discuss. If there is another condition you need to check, maybe several more, you can use the `elseif` keyword. It looks something like this:

```
color = "green"

if color == "blue" then
  print("That's my favorite color!")
elseif color == "green" then
  print("Very subtle choice. I like it.")
elseif color == "pink" then
  print("Nice, bold choice.")
else
  print("I don't think that color would match your shoes.")
end
```

Try it out!

- The beginning of the *if statement*... `if color == "blue" then` ... is false. This code gets skipped over.
- Then the next part of the *if statement*... `elseif color == "green" then` ... is true so that section of code underneath it... `print("Very subtle choice. I like it.")` is ran.
- The rest of the *if statement* is skipped without checking if its true or not. So `elseif color == "pink" then` / `else` are never processed.

Exercises

- Write out a function that takes 1 parameter named "sides". Make the function return the name of the shape depending on the number of sides (for instance, "triangle"). Try to make the if statement include an `else` at the end to account for everything else that the if doesn't.

While

Another way to check conditions is with the `while` keyword.

```
while 1 + 1 == 2 do
  print("My math is correct!")
end
```

While a condition is true, the body (everything between the `do` and `end`) will be run repeatedly and not stop. So if you tried to run that bit of code, your screen probably went crazy printing over and over in a never-ending loop. We need to make sure the condition can get changed so we're not stuck in a never-ending loop. Let's write a loop we can escape out of.

```
boolean = true

-- This condition will get checked twice. The first time it
-- is checked it will be true and the body of the while-loop
-- will be run. The second time the condition is checked,
-- our boolean will be false and the while-loop won't be run again!
while boolean do
  print("Switching boolean to false.")
  boolean = false
  print("Boolean has been set to false.")
end

print("We made it out of the loop!")
```

Understanding that we can change the *while* condition from inside the body of the loop, we have the power to write programs that end exactly when we want them to. Can you guess what this will do when we run it?

```
countdown = 10

while countdown > 1 do
  print(countdown .. "...")
  -- This line is critical to make our number shrink.
  countdown = countdown - 1
end

print("Blast off!")
```

...And remember to use a `>` and not a `<`, or your loop may never run.

Exercise

- Come up with your own idea for a while loop.

Type checking

Lua doesn't care what type of data a variable has.

```
data = 12
data = "hello"
data = true
```

To this end, we can use the `type` function to check what kind of data a variable is holding.

```
type(data)
```

```
=> boolean
```

We can check the type of function:

```
type(string.reverse)
type(type)
```

We can also use it to check what type of data a function is returning back to us:

```
type(string.reverse("hello"))
```

```
=> string
```

```
type(type(12))
```

```
=> string
```

Converting data types

We've already seen data type conversion previously when we took numbers and an operation in, transforming that into a true or false statement.

```
type(12 > 3)
```

```
=> boolean
```

There are also ways to convert between numbers and strings using `tonumber` and `tostring`.

```
number = tonumber("24")
print(type(number))
string = tostring(number)
print(type(string))
```

```
number
string
```

Interesting but maybe less useful, you can convert other data types to string:

```
print(tostring("already a string"))
print(tostring(true))
print(tostring(nil))
print(tostring(tostring))
```

Exercises

- Which of these strings can be converted to a number successfully? `"001"`, `"7.12000"`, `" 5 "`, `"1,943"`

First game

Let's learn about a few new functions and then we'll be able to write our first game!

Reading input

Not only can our program print out data, but using the function `io.read` it can take data too. This function doesn't need any arguments because it will prompt us in the window on the right for us to type in data.

```
print("Enter your name:")
name = io.read()

print("Your name is " .. name .. ".")
```

After you click "Run", the program will pause when it runs `io.read()`. Type your name and hit ENTER and look, the program prints back out the name you gave it. Notice the last print statement. We combined the name with two other strings to form a sentence. You can prompt the user multiple times if you need to get additional information:

```
print("Enter your name:")
name = io.read()

print("What's your favorite food?")
food = io.read()

print("Your name is " .. name .. " and your favorite food is " .. food .. ".")
```

One limitation with doing this is the data will always come in as a string:

```
print("What's your favorite number?")
data = io.read()

print(type(data))
```

```
string
```

In the last section we talked about converting data between different types. If we wanted to find out whether your favorite number is odd or even, we would need to convert it to an actual number to perform operations on it. Type this in your text editor and run it:

```
print("What's your favorite number?")
data = io.read()
number = tonumber(data)

-- If the user gave us an answer that isn't a
-- number, then the value of "number" is nil.
if number == nil then
  print("Invalid number.")
elseif number % 2 == 0 then
  print("Your number is even.")
else
  print("Your number is odd.")
end
```

Random number

Many languages give us access to a random number generator. Randomness is how we generate secure passwords and keys in the real world. To generate a random number in Lua, we use `math.random`:

```
math.random(100)
```

```
=> 63
```

This generates a random number between 1 and 100. Except, if you run the program repeatedly you may notice that it spits out the same number. That's because nothing in the computer world is random. If we fed in random noises through a speaker or white noise from an old television set then our computer could use this to generate random numbers. Since we don't easily have access to those things, we need to *seed* Lua with some *perceived* randomness.

If we run `os.time` we will get the computer's current time in integer form:

```
os.time()
```

```
=> 1.529098167e+09
```

This number is hard enough to guess that it will work as a seed for our program. Let's take the system time and feed it in using `math.randomseed` then from there, Lua will be able to generate a "random" number in the range we want (1-100).

```
seed_number = os.time()
math.randomseed(seed_number)
print(math.random(100))
```

```
=> 19
```

Success! It is generated different numbers each time we run it, with no pattern.

Putting it all together

I should probably explain what this game is. It's quite simple. We want the computer to make up a number and the user has to guess what the number is. If they're wrong, then we should give them a hint and make them guess again. We can take advantage of the while loop to make them continue guessing while their guess is incorrect.

```
-- The computer's secret number
math.randomseed(os.time())
number = math.random(100)

print("Guess my secret number. It is between 1 and 100.")

guess = tonumber(io.read())

-- While the user's guess is not equal to
-- the number, repeat the body of the loop.
while guess ~= number do
  -- Give them some hints
  if guess > number then
    print("Your guess is too high.")
  end
  if guess < number then
    print("Your guess is too low.")
  end

  -- Make them guess again and again until they get it
  print("Guess again:")
  guess = tonumber(io.read())
end

-- Winning message
print("You guessed correctly! The number was " .. number .. ".")
```

Let's re-factor one bit of this code to make it easier to read. When we talked about if statements, remember the keyword `else`?

```
-- The computer's secret number
math.randomseed(os.time())
number = math.random(100)

print("Guess my secret number. It is between 1 and 100.")

guess = tonumber(io.read())

-- While the user's guess is not equal to
-- the number, repeat the body of the loop.
while guess ~= number do
  -- Give them some hints
  if guess > number then
    print("Your guess is too high.")
  else
    print("Your guess is too low.")
  end

  -- Make them guess again and again until they get it
  print("Guess again:")
  guess = tonumber(io.read())
end

-- Winning message
print("You guessed correctly! The number was " .. number .. ".")
```

Now that things are cleaner, let's add one feature to our program. It would be more fun if the game kept track of how many guesses we made so we could give them a special message. Let's create a variable called `guess_count` that will start at `1` and increment every time the user makes another guess. We'll also go ahead and add some messages to the bottom to praise the user if they did it in a reasonable number of guesses.

```
-- The computer's secret number
math.randomseed(os.time())
number = math.random(100)
-- Our starting number of guesses
guess_counter = 1

print("Guess my secret number. It is between 1 and 100.")

guess = tonumber(io.read())

-- While the user's guess is not equal to
-- the number, repeat the body of the loop.
while guess ~= number do
  -- Increment the guess counter by 1
  guess_counter = guess_counter + 1

  -- Give them some hints
  if guess > number then
    print("Your guess is too high.")
  else
    print("Your guess is too low.")
  end

  -- Make them guess again and again until they get it
  print("Guess again:")
  guess = tonumber(io.read())
end

-- Winning messages
print("You guessed correctly! The number was " .. number .. ".")

if guess_counter <= 5 then
  print("Amazing! It only took you " .. guess_counter .. " tries.")
else
  print("It took you " .. guess_counter .. " tries. Not bad.")
end
```

Exercises

- Try adding more messages for different numbers of guesses. You can modify the if statement where `guess_counter` is checked.
- Make the `while` condition exit if `guess_counter` goes above 10 and tell the user they lost the game (but that should try again).
- Try adding some messages to the if statement with the hints for when the user guesses an invalid number too far out of range. What if they guess a number that is more than 100? How would you do that?

Tables (part 1)

Tables are the last data type we'll discuss in this chapter. Other languages have different names for this data type like "object", "hash", "map" and "dictionary", and the features may vary from one programming language to another. Tables are used to build *composite data types* like lists, trees, or a big green orc running across the screen. Composite data types are higher order data structures created from more primitive data types like numbers and strings. The number of data structures you can create are endless. We need to learn about a few to not only understand how tables work, but to be able to build any modern software.

The basic syntax for tables is to make a curly brace `{` (same key as the square brace on most keyboards) to start the table, write some data in the table, then put a closing curly brace `}` to end the table. So an empty table would look like this:

```
my_cool_table = {}
```

Lists

Lists are usually started by writing the first item, then the second, and so on. If we wanted to make a grocery list in software, it may look like this:

```
groceries = {
  [1] = "beans",
  [2] = "bananas",
  [3] = "buns"
}
```

Ok maybe your typical grocery list looks different. What do we do with this data now that we got it? We can access and modify the data as if they were stored in their own variables.

```
return groceries[1]
```

```
=> beans
```

First we specify the variable name of the table, then in square brackets we put the number we want. You can access them in any order and modify them as needed:

```
print(groceries[3])
groceries[1] = "coffee beans"
print(groceries[1])
```

```
buns
coffee beans
```

The order you define them in doesn't matter:

```
groceries = {
  [3] = "beans",
  [1] = "bananas",
  [2] = "buns"
}
```

The number in square brackets is the *key*. A key that is part of a numeric sequence of keys such as this list is often called an *index*. So `"bananas"` has an index of `1`. The plural of index is *indices*.

Don't forget the commas between each item in your list or you will get quite the error message:

```
[string "groceries = {..."]:3: '}' expected (to close '{' at line 1) near '['
```

When you are missing a comma between items, it thinks it has reached the end of the table but then errors out when it goes to close the table but sees another item instead of the close curly bracket `}`.

Another issue you may run into is if you try to access a key that has no data. There is no 4th item in our table so if we try to access it:

```
print(groceries[4])
```

We get back `nil`, the same way we would if we tried to access a variable name that has no data assigned to it.

Writing out large lists can become a headache when we have to manually number each item in a list:

```
groceries = {
  [1] = "beans",
  [2] = "bananas",
  [3] = "buns",
  [4] = "blueberries",
  [5] = "butter",
  [6] = "broccoli",
  [7] = "basil"
}
```

What if we remove an item or we want to move something to a different position in the list? What a pain to have to re-index everything. Thankfully there is shorthand way of writing lists:

```
groceries = {
  "beans",
  "bananas",
  "buns",
  "blueberries",
  "butter",
  "broccoli",
  "basil"
}
```

This is identical to the code written above, except now the indices are auto-generated for me. `"basil"` has an index of `7` since it is the 7th item in the list, but if I cut and pasted it to the top of my list, it's index would be `1` and everything below it would be renumbered accordingly.

Looping over lists

If we wanted to print our grocery list, we could say something like:

```
print(groceries[1])
print(groceries[2])
print(groceries[3])
print(groceries[4])
-- and so on...
```

But that is quite repetitious and requires updating if the size of our list changes. Luckily we already know about *while loops*.

```
index = 1

while groceries[index] ~= nil do
  print(index, groceries[index])
  -- Go to the next index in the list
  index = index + 1
end
```

See how instead of accessing each item as `groceries[1]`, `groceries[2]` ... we can just use a variable in the square brackets instead of a number. Then inside the loop we bump the number up and access the next item in the list. The loop stops when the index goes beyond the last item in the list and there is nothing there. So when index 8 is read, `groceries[8]` is nil and the while condition is no longer true. While conditions don't even need a boolean expression. It can know whether or not to continue simply if the given item has data or is nil. It can be simplified to read:

```
index = 1

while groceries[index] do
  print(index, groceries[index])
  -- Go to the next index in the list
  index = index + 1
end
```

Again, it knows to exit when it sees `false` or `nil`. The caveat to this would be if you make a special list with `false` in it:

```
groceries = {
  "beans",
  "bananas",
  false,
  "blueberries",
  "butter",
  "broccoli",
  "basil"
}
```

When the while loop gets to the third item in the list and sees `false`, it would stop looping before it reads the rest. It's not typically a good idea to mix and match different data types in a list because of issues like this, however, we could work around this if we needed to. There is a special operator for tables to get the size of the list.

```
print(#groceries)
```

```
7
```

An easy way to remember the `#` operator is to remember that it returns the # of items in a list. Using this operator we could write our while loop in a different way.

```
index = 1

while index <= #groceries do
  print(index, groceries[index])
  -- Go to the next index in the list
  index = index + 1
end
```

You could even tweak this slightly to read the list backwards if you wanted to:

```
index = #groceries

while index > 0 do
  print(index, groceries[index])
  -- Go to the next index in the list
  index = index - 1
end
```

Notice we are subtracting from the index with each loop in order to accomplish this.

Exercises

- Try to modify the while loop to only print every other item in the grocery list. (Hint: instead of incrementing by 1 on each read, you want to increment more.)

- Write a while loop that counts to 10 and populates an empty table with the same item 10 times. (Hint: you assign to indices just like variables, `list[index] = "hi"`.)

Tables (part 2)

In the last section we saw how simple it was to make a list. Working with the list was a little tricky at first but hopefully not too bad. If we rewind back, we can remember that we created a table by assigning some keys values.

```
boxes = {
  [1] = "John Doe",
  [2] = "Amanda Parker",
  [3] = "Tyler Reese"
}
```

Think of it like post office boxes and we label each box with a unique number. Whenever we reference a postal box, we do so by referencing the number within the array (list) of boxes: `boxes[2]`. The label, or key, is ultimately arbitrary though. For making a list, we label things in an incremental order to make them easier to loop over and to give us a sense of linear sequence. Keys don't need to be numbers. They could just as well be strings:

```
coins = {
  ["half"] = "50 cents",
  ["quarter"] = "25 cents",
  ["dime"] = "10 cents",
  ["nickel"] = "5 cents",
  ["penny"] = "1 cent"
}
```

Which would be accessed just the same way:

```
print(coins["nickel"])
```

```
5 cents
```

This can be really useful for doing a lookup if we instead use a variable for the key. Try this one out:

```
coins = {
  ["half"] = "50 cents",
  ["quarter"] = "25 cents",
  ["dime"] = "10 cents",
  ["nickel"] = "5 cents",
  ["penny"] = "1 cent"
}

print("Which coin do you have?")
response = io.read()

print("Your coin is worth " .. coins[response] .. ".")
```

This isn't far off from how certain databases and digital services work. Items are stored in a unique key that can be referenced for getting a definition out of later. That's why this data structure is sometimes called a dictionary. Remember, we can add items to a table after it is defined:

```
coins["silver dollar"] = "1 dollar"
```

Another shortcut Lua gives us is we don't need to use the square braces or quotes when adding keys that are strings.

```
coins.nickel = "5 cents"
```

The limitation with doing this is the keys defined this way can't have spaces or special characters. They must be valid in the same way variable names are valid.

```
coins.silver dollar = "1 dollar"  -- INVALID
coins.silver_dollar = "1 dollar"  -- Valid
coins.100 = "1 dollar" -- INVALID
```

You can use variable names for keys when creating the table too:

```
color = "purple"
description = "the best color"
colors = {
  [color] = description
}

print(colors.purple)
print(colors[color]) -- prints the same thing
```

By convention, strings are typically used for dictionary-like tables while lists are numbers. Don't make the mistake of thinking these are the same:

```
list = {
  1 = "some item",
  ["1"] = "a unique item"
}
```

You could use other data types as keys, but you might find your results to be very unexpected:

```
crazy_list = {
  [true] = "works",
  [false] = "works",
  ["true"] = "not the same",
  ["false"] = "not the same"
}

print(crazy_list[true])
print(crazy_list[false])
print(crazy_list["true"])
print(crazy_list["false"])
```

```
crazy_key = {}
crazy_list = {
  [crazy_key] = "works"
}
print(crazy_list[crazy_key])
```

```
crazy_list = {
  [nil] = "doesn't work!"
}
print(crazy_list[nil])
```

Throws an error:

```
[string "crazy_list = {..."]:2: table index is nil
```

Values in a table can be any type of data, including functions:

```
cat = {
  color = "gray",
  smelly = true,
  make_sound = function()
    print("meyuow!")
  end
}

cat.make_sound()
```

Exercises

- Remember the early function we made that returned the animal sounds? Make a function with a table in it, where each key in the table is an animal name. Give each key a value equal to the sound the animal makes and return the animal sound. Try invoking the function and see if you get back the correct sound.

For loops (part 1)

We saw previously that we could use while loops for many things, but we also saw how easy it was to make a while loop that didn't run properly. The programmer has to make a variable to pass to the condition, make sure the condition is written out correctly, and then make sure the condition can be changed so the loop can eventually end. This many steps each time we want to write a simple loop leaves us prone to errors and wasting our time. With *for loops*, we can tell a loop exactly how many times we want it to run and skip all these steps.

Numeric for loops

```
for number = 1, 10 do
  print(number)
end
```

On the first line we are saying "For [starting number] through [ending number] do the following". The `number =` is a variable you are assigning the starting number to. The variable name can be whatever you want. The second line is the body of the loop and the third line ends the loop. If you run this program, it will print the numbers `1`, `2`, `3` ... through `10` as `number` is being incremented by 1 with each loop. This variable is a bit peculiar though, not only because we defined it in the middle of a statement but because it disappears after we are done with the loop.

```
for number = 1, 10 do
  print(number)
end

return number
```

```
=> nil
```

This is called a *local variable*, because it only exists locally within the for loop.

For loops actually have 3 parameters:

- start number - we assign the variable to it and the variable will increment with each loop
- stop number- the last number to increment to before stopping the loop
- step - how much to increment by with each loop. If we don't specify a step size it will default to 1.

Let's say we wanted to print out only even numbers. We could change the starting number to 2 and set the size of the step (3rd parameter) to 2:

```
for number = 2, 10, 2 do
  print(number)
end
```

If we wanted to *iterate*, or loop over and read each item in a list, it would look similar to a while loop. Let's look at the while loop example again just for comparison:

```
items = {'a', 'b', 'c', 'd'}
index = 1

while index <= #items do
  print(items[index])
  index = index + 1
end
```

```
items = {'a', 'b', 'c', 'd'}

for index = 1, #items do
  print(items[index])
end
```

We could also count down by changing the parameters around and setting the step to a negative 1.

```
items = {'a', 'b', 'c', 'd'}

for index = #items, 1, -1 do
  print(items[index])
end
```

In this case the index starts at the position of the last item and stops when it gets to the stop number, 1.

Exercises

- Modify the previous loop so that it only prints every other item in the list.

For loops (part 2)

We can create a different style of for loop using functions, but in order to do that, we need to understand another aspect of functions we haven't yet covered. Functions can return multiple values.

```
sort_numbers = function(a, b)
  if a > b then
    return a, b
  end
  return b, a
end

bigger, smaller = sort_numbers(12, 18)

print(bigger)
print(smaller)
```

This function takes two numbers, checks to see which is bigger, then returns both the bigger number first then the smaller number second. Notice we did this by putting a comma in the return statement then providing a second value after the comma. Likewise, we were able to capture both values into variables by putting the first variable name, a comma, then the second variable (`bigger, smaller =`). We don't need to capture everything returned from a function. We could have just as easily called the function and only captured the bigger number if that's all we wanted from it.

```
bigger = sort_numbers(12, 18)
```

Generic for loops

Let's take a look at the sibling to the numeric for loop called the generic for loop. It's called generic for loop because it takes a function that makes it behave in different ways for different situations. It doesn't do anything on its own. It relies on the function to tell it how to behave.

ipairs

Here's what generic for loops look like:

```
list = {'dog', 'cat', 'mouse'}
for index in ipairs(list) do
  print(index, list[index])
end
```

`ipairs` takes our for loop and makes it iterate over each item in the list and gives us an `index` variable to work with inside the loop. But wait, there's more! `ipairs` provides us with another variable that holds the value of the item at that index. Try it out yourself:

```
list = {'dog', 'cat', 'mouse'}
for index, value in ipairs(list) do
  print(index, value)
end
```

Ah yes, so convenient! There is one gotcha with doing this. If you wanted to edit the table from inside the loop, you need to access the table directly:

```
list = {'dog', 'cat', 'mouse'}
for index, value in ipairs(list) do
  list[index] = string.upper(value)
end

print(list[1])
```

If you try to just edit the value:

```
list = {'dog', 'cat', 'mouse'}
for index, value in ipairs(list) do
  value = string.upper(value)
end

print(list[1])
```

the list won't be modified, because `value` is just a copy of the data that's actually in the list. You're editing a temporary copy.

pairs

Another function for programming for loops with special functionality is `pairs`. This will iterate over every key in a table:

```
table = {
  cat = 'meow',
  dog = 'bark'
}

for key, value in pairs(table) do
  print(key, value)
end
```

Even indices:

```
table = {
  'a',
  'b',
  'c',
  cat = 'meow',
  dog = 'bark'
}

for key, value in pairs(table) do
  print(key, value)
end
```

No sneaking past `pairs` for any of these keys either:

```
table = {
  [1] = 'a',
  [2] = 'b',
  [3] = 'c',
  cat = 'meow',
  dog = 'bark',
  [true] = false,
  [{}] = 'what?'
}

for key, value in pairs(table) do
  print(key, value)
end
```

An easy way to remember the difference between `ipairs` and `pairs` is the "i" in `ipairs` stands for index. Sure there's a difference when working with weird tables like the one above, but why can't we just use `pairs` for regular list-style tables?

```
table = {
  [2] = 'b',
  [3] = 'c',
  [1] = 'a'
}

for key, value in pairs(table) do
  print(key, value)
end
```

```
3   c
2   b
1   a
```

As you can see, the order of the items isn't guaranteed with `pairs`. `ipairs` is also optimized to handle numeric keys and will generally perform faster, so it's good to know the difference.

Under the generic-for-loop hood

`ipairs` and `pairs` are just regular functions that we invoke. They return a function (yes, a function that returns a function!) and this returned function programs our loop to behave how we want.

```
for key, value in iterator, list, start_number do
  print(index)
end
```

So this is what a generic for loop really looks like without the help of `ipairs` or `pairs`. It requires 3 parameters that `ipairs` / `pairs` provides data back to the key and value variables that we can use inside the loop. `iterator`, `list`, `start_number` are all variables we would otherwise have to define without their help.

- `iterator` would be a function we provide to the loop

- `list` would be what we want to iterate over
- `start_number` would be the starting index in the list

```
list = {'a', 'b', 'c'}

iterator, list, start_number = ipairs(list)

for index, value in iterator, list, start_number do
  print(index)
end
```

`ipairs` gives us an iterator to pass to the for loop, as well as our list we already had, and a starting number. We can print the results of ipairs and see the 3 things it gives us:

```
print(ipairs(list))
```

```
function: 0x156a3f0    table: 0x1572aa0    0
```

So to say it again, generic for loops require 3 things: an iterator function, our list, and a number. In order to not have to write them ourselves, we generated those 3 things by invoking `ipairs` then passing them into the for loop parameters. Don't fret too much if this seems confusing right now because we're not going to need to write custom for loops or custom iterators.

Numeric versus generic: which to use?

Numeric for loops are good for simple counting but perform just as well or maybe even better than generic for loops. Generic for loops are more adaptable. If you have a situation where either would work, just use whichever you want. It really won't make any difference.

Exercises

- Make a list and then write both a numeric for loop and generic for loop that iterate over the list and print each item. Compare the two approaches.
- Make a table with animals for keys and the sounds they make for the key values. Make a for loop that uses `pairs` to iterate over each and change the noises to all capital letters.

Scopes

When defining functions, we define parameters for those functions which work like regular variables. If we try to access a parameter outside a function we will see that it is `nil`.

```
addition = function(a, b)
  print(a, b)
  return a + b
end

addition(1, 2)

print(a, b)
```

The parameters `a` and `b` are *local variables*. We've seen local variables with for loops, where the variable `counting_number` couldn't be accessed outside the for loop:

```
for counting_number = 1, 4 do
  print(counting_number)
end

print(counting_number)
```

Functions, for loops, and while loops create a *scope* each time they are ran. Things created in the scope, including local variables, are destroyed when that loop or function invoke is done. This is how the program tidies up after itself and keeps the computer from running out of memory. The process of removing unused data from memory and releasing control of that memory is called garbage collection. Lua does this for us so we don't have to think about it. Variables we create normally don't follow the same rules. They will continue to exist after the scope they were created in has been destroyed.

```
addition = function(a, b)
  text = "I'm not going away."
  return a + b
end

addition(1, 2)

print(text)
```

Eventually all these variables we make will fill up memory unnecessarily. This can also be problematic if we accidentally make two variables but use the same name.

```
x = 2

addition = function(a, b)
  -- This modifies the x at the top!
  x = 9
  return a + b
end

print(x)
result = addition(x, y)
print(x)
```

When you write a large program, you'll inevitably make two variables with the same name, so this could be a huge issue. The solution is to make our variables local variables by putting the keyword `local` before all our variables when we create them.

```
addition = function(a, b)
  local text = "I'm only accessible inside the function."
  return a + b
end

addition(1, 2)

print(text)
```

Now `text` is only in the scope of the function and not getting into other people's business. If you don't write `local` before a variable, then what you are creating is a *global variable*. It's a shame that variables are global unless we explicitly tell them not to be. There is never a reason to create global variables if you have enough knowledge to know not to. So as a best practice, all code examples going forward, only local variables will be created. Let's see a few more examples:

```
local number = 12

-- This function has no parameters
local print_numbers = function()
  -- This works. You can see variables outside the function
  print('number:', number)
  -- This doesn't work. The variable didn't exist
  -- at the time "print_numbers" was created.
  print('number2:', number2)
end

local number2 = 18

print_numbers()

-- We already "declared" number. We don't write "local" again.
number = 13

print_numbers()
```

Notice when it printed that it knew `number` was updated to 13 but couldn't track `number2`. As long as a variable was created before the scope (function's scope in this case) was created then it will always track the latest value.

As a reminder, we already saw with the for loop and while loop that you can modify variables in the outer, or parent, scope:

```
local number = 1

while number < 10 do
  number = number + 1
end
```

This also works with functions:

```
local number = 1

local mutate_number = function()
  number = 7
end

print(number)
mutate_number()
print(number)
```

What if you make two variables with the same name in two different scopes? Try running this one:

```
local number = 18

local shadowing = function()
  local number = 6
  print(number)
end

print(number)
shadowing()
```

The inner `number` does not affect the outer `number` in any way. The outer `number` is not accessible inside the function as long as the inner `number` exists. If a variable has the same name as another variable in a parent scope then the parent scope variable becomes inaccessible: this is called *shadowing*. Typically you would want to avoid shadowing if at least for the reason that using the same variable name twice in the same file can make the code harder to read and more prone to errors being introduced.

One more interesting things with scopes. Normally a function cannot see itself:

```
local self_reference = function()
  print(self_reference) -- This will be nil!
end

self_reference()
```

It doesn't see itself because the variable is still being created when the function is being created. But remember that if a variable exists before the function does, it can see the latest up-to-date content of that variable. So here's the trick to make that work:

```
local self_reference = nil
self_reference = function()
  print(self_reference)
end

self_reference()
```

The variable is declared, even though we assigned `nil` to it. Assigning nil to get a variable declared is pretty common, so Lua includes a shorthand way of declaring empty variables:

```
local self_reference
self_reference = function()
  print(self_reference)
end

self_reference()
```

This may seem silly that a function would need to access itself, but there are some very powerful applications for this that we will see later on.

Exercises

- Declare a global variable inside a function, `x = 5` (no local keyword) then try to print the variable from outside the function. Can it be printed? How?

Chapter review

Terminology

- **Operator** and **Operation** - Operators are symbols that cause an operation, or interaction to happen between two pieces of data. An example operation would be `5 + 5`. `(5 + 5) * 2` would be two operations.
- **Modulo** and **Modulus** - Modulo is a special type of arithmetic operation between two numbers using a modulus operator. The modulus is represented by a `%` (percent symbol). Example: `24 % 2 == 0`
- **Variable** and **Value** - Variables are names that reference a certain piece of data. The value is what is stored inside the variable: `variable = "value"`
- **Statement** - This is when you do something, like an operation (or group of operations), declare a variable, or invoke a function. For instance, this is a print statement: `print("hello")`
- **Invoke** - Run/call a function.
- **Parameter** and **Argument** - Functions tell you what and how many parameters they have. Arguments are the data that gets passed into those parameters.
- **Boolean** - `true` or `false`
- **Equality** - Whether or not two things are equal. This is usually done with an equal (`==`) comparison.
- **Loop** - Code that repeats.
- **Key** and **Index** - Key is the named reference in a table where data can be found. It is similar to a variable. Index is a key that comes in an ordered sequence, such as numbered keys in a list. The plural of index is indices.
- **Iterate** - Loop over a list and do something with it.
- **Scope** - An area where variables can be created that aren't accessible from the outside. Scopes are created by functions and loops.
- **Local** and **Global** - Local describes things accessible only in the current scope, such as local variables. Global things are accessible from anywhere in the program.
- **Shadowing** - When a local variable has the same name as a variable in a parent scope and prevents you from accessing the parent scope variable.

Chapter 2: Introducing LÖVE

The goal of this chapter is to apply all the building blocks we learned in the first chapter and make them concrete through practice. By the end of the chapter you will learn real-world skills such as how to interact with other people's software and basic principles for designing and building your own programs.

Up and running

Learning by making is fun and effective. Learning to interface with other people's software is part of being a programmer and is a necessary skill to have as one. LÖVE is a framework for making games. A framework is just a set of tools or functionality combined together to serve a larger purpose. In the case of LÖVE this includes, but is not limited to:

- Functions for loading images, audio, and text
- Functions for creating and moving objects on screen
- Parameters for making the objects interact

Installing your development environment

The LÖVE website has links to install the software on your system. If you have LÖVE installed already, make sure that you at least version 11 (mysterious mysteries) as some functionality we'll cover here doesn't exist in older versions. For mobile devices you can find a copy in the app store.

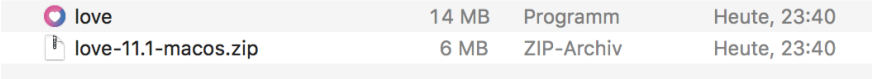
Along with installing LÖVE, you will need a text editor for creating Lua files on your system. I'm not going to make any recommendations here, because in the end it all comes down to personal preference, but you can check this list by the LÖVE community if you need a starting point. It features different editors (and recommended plugins) for LÖVE and Lua development. Simply pick one.

Test that LÖVE runs

When you launch LÖVE, (see instructions below on how to do that) you will be greeted with a friendly graphic and the text "NO GAME", meaning you are running the engine but didn't give it a game to load.

macOS

Once you have downloaded the LÖVE binary for macOS (64-bit zipped), proceed to the "Downloads" folder and unzip the archive. You should now see an application called "love".



macOS might show you a warning modal, because you are trying to open an application by an unverified developer. If so, right click on the application and choose "open" and "open" again in the following dialog. You should now be greeted by the no-game screen.

Addendum: Homebrew

If you are familiar with development on a macOS machine, you might have heard of Homebrew. It is a package manager which allows you to install a lot of programs, libraries and so on directly through your Terminal.

I only recommend this approach for advanced developers who know what they are doing. For completeness sake here are the steps to install LÖVE via Homebrew.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/insta
brew tap caskroom/cask
brew cask install love
```

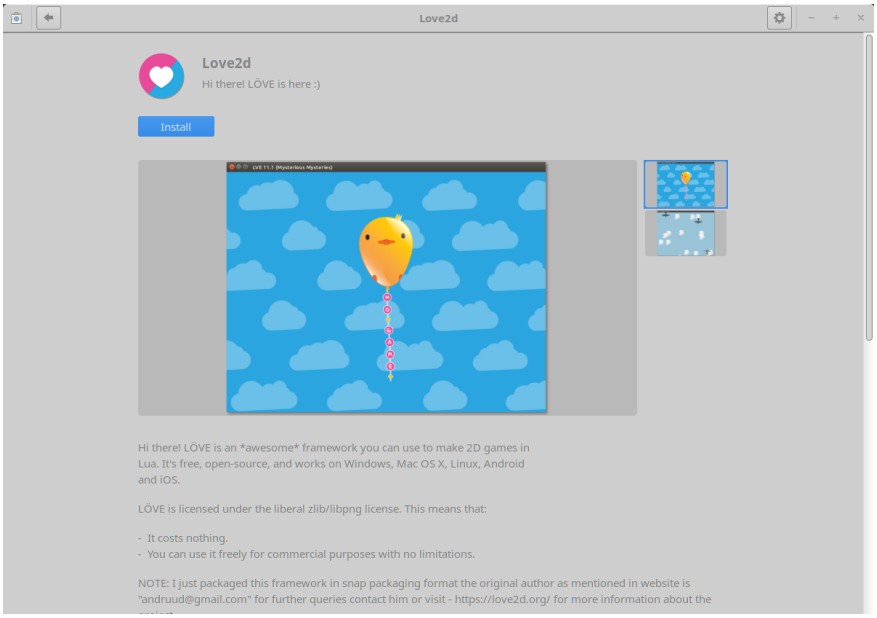
One of the benefits of this approach is, that you don't have to set up your own terminal alias, because Homebrew also takes care of that.

Windows

If a shortcut for LÖVE didn't appear in the start menu, you should be able to type "love" in the search and see it.

Ubuntu

Open the "Ubuntu Software" application and search "love2d". Click on the top result and you should see a familiar application description:



Click the "Install" button to install it. Once installed, you can search for the "terminal" application. Once that is open, type `love` to launch the application.

Other GNU/Linux operating systems

Most distros have LÖVE in their respective repositories:

- Arch linux-based systems - `sudo pacman -Sy love`
- Fedora-based systems - `yum install love`

Once installed from your package manager, open a terminal and type `love` to test that it runs.

If your distro doesn't have LÖVE in the package manager an alternative way to get it is to downloaded the AppImage version from the homepage (https://love2d.org/). AppImage files are like a universal executable that works across Linux systems similar to the way an "exe" file works on Windows. Once downloaded, open the terminal, change to the directory where you downloaded the AppImage and type the commands:

```
chmod a+x love-11.1-linux-x86_64.AppImage # Marks the file as a safe executable
./love-11.1-linux-x86_64.AppImage
```

`love-11.1-linux-x86_64.AppImage` should be changed to match the name of the downloaded AppImage file.

Create a project folder

Find a safe place to create a folder and give it the name "hello". Within the folder, create a new text file named "main.lua". This will be where our game's code goes.

Note for Windows: In order to create a file with the name "main.lua", you may need to first create a new "Text Document", right-click on it, click "Properties" then from the properties menu change the file extension from reading "main.lua.txt" to just "main.lua". To avoid having to do this for every Lua file you create in the future you can tell Windows to always show the full name of files, including their extension. To enable this, type "Control Panel" in the program search and open the "Control Panel" result. Within the control panel, select "File Explorer Options". Click the "View" tab. Remove the check mark from the "Hide extensions for known file types" and press Apply/OK.

Create a test game

Within "main.lua", write out the following function:

```
love.draw = function()
  love.graphics.print('Hello World!', 400, 300)
end
```

Now let's figure out how to run it and see what it does.

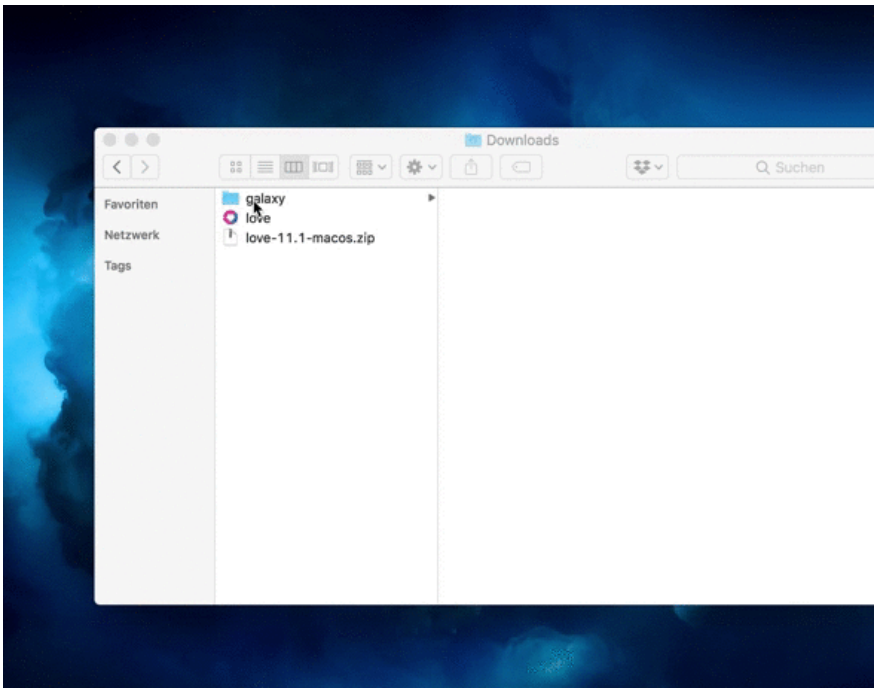
Run the game

This will be different for different operating systems.

macOS

Starting your game

The simplest way to start a LÖVE game is to drag the whole folder containing the game's source files (not just the main.lua file!) onto the application file.



This also works with .love files.

Using the terminal

If you are familiar with the Terminal, you can use it as a more convenient method of starting games.

Assuming the downloaded "love" application is still in your "Downloads" folder, open a new Terminal and type the following lines (you need to press return after each line):

```
# Switches to the Downloads folder
cd ~/Downloads/

# Start the LÖVE app
open love.app
```

This obviously starts LÖVE with a no-game screen since we didn't specify which folder to load. Let's fix this by typing the following command:

```
# In my case the full command to start the galaxy demo would be:
# open -a love.app ~/Downloads/galaxy
open -a love.app <path-to-your-game>
```

Using a terminal alias

We can still improve on the previous method by using an alias. Before we do this, we move the "love" application bundle to the Application folder.

```
# Move the app from Downloads to Applications.
~ mv ~/Downloads/love.app/ ~/Applications/love.app
```

Now try the following command:

```
# Start LÖVE by using the script inside of the application bundle.
~/Applications/love.app/Contents/MacOS/love <path-to-your-game>
```

As you can see we now can run LÖVE without using the `open` command, which also has the added benefit of showing the game's console output directly in our terminal.

Of course it would be rather inconvenient if we had to specify the full path each time we want to run our game, so we'll now set up an alias in your .bash_profile (which basically acts as a configuration file for your bash sessions).

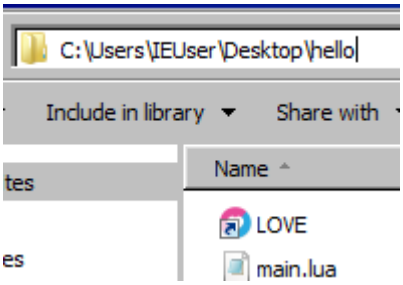
Since it is a hidden file you might not be able to spot it in your finder, but we can simply edit it through our Terminal.

```
# Appends the alias definition to an existing .bash_profile
# or creates a new one.
echo "alias love='~/Applications/love.app/Contents/MacOS/love'" >> ~/.bash_prof

# Use the updated .bash_profile for the current session.
source ~/.bash_profile

# Start your game through the alias.
love <path-to-your-game>
```

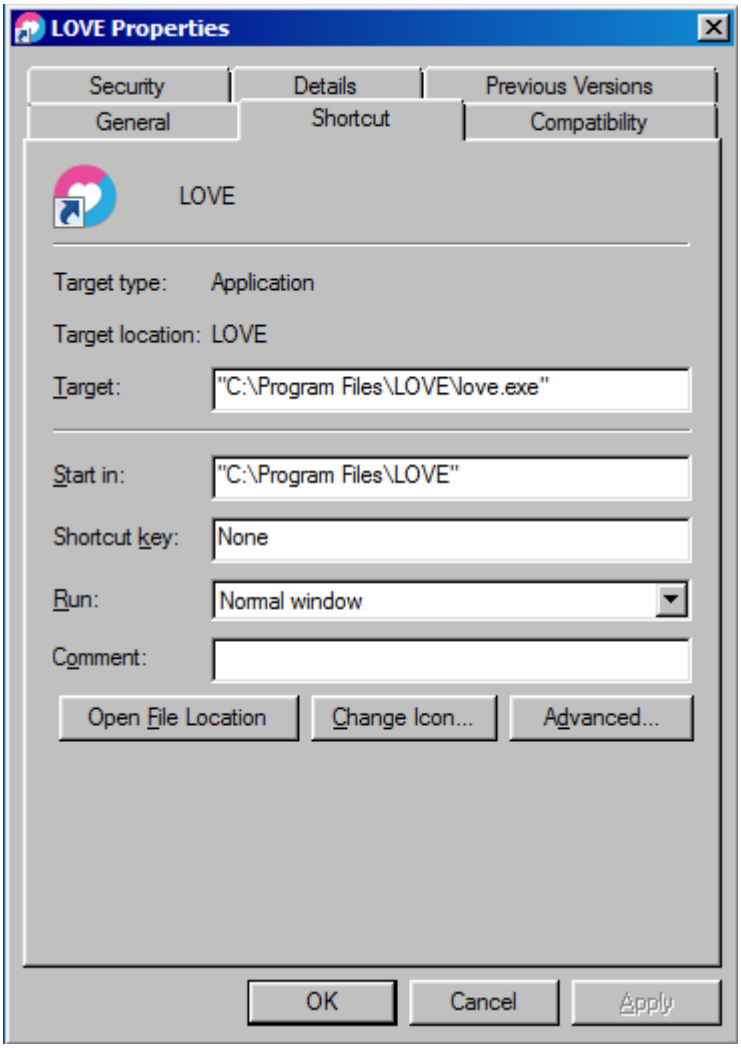
And that's it: You can now quickly run your games with the `love` alias. This is especially handy if you are inside of the game's directory, because all it takes now is a quick `love .` to start the game.

Windows

Find the shortcut to LÖVE and drag and drop it in the game folder like so:



Then right-click on the LÖVE shortcut and you will see a "Properties" dialog window similar to this:



The "Target" field may be the same or slightly different depending on your system version. Without deleting the text string currently in the "Target" field, append the path to your game folder in quotes to the end. You can copy and paste this path from the folder's address bar. For instance the target path in the picture should go from reading:

```
"C:\Program Files\LOVE\love.exe"
```

to

```
"C:\Program Files\LOVE\love.exe" "C:\Users\IEUser\Desktop\hello"
```

Now press "OK" to close the Properties dialog and clicking the shortcut will launch the game. If the game ran successfully, you will see a black window with the text "Hello world!" in small print.

GNU/Linux

If you know the location of your folder, you can open a terminal and type the command:

```
love <path-to-your-game>
```

Where `<path-to-your-game>` has been changed to the actual folder path where your game resides.

If you are already navigated into the game folder, you can run a terminal command within that directory:

```
love .
```

The "." simply means "this folder that I am currently in".

Congratulations!

You've set up your development environment for writing a game in Lua. If you had issues getting through this, reach out to me either through a GitHub issue or my contact information and I will update this guide to including any additional troubleshooting steps for future users.

Now that our development environment is set up and our first game is running, try modifying the code so the string "Hello World!" reads something different. It's pretty apparent that running this function prints to the screen whatever string we give it. But what are the 2nd and 3rd parameters for?

```
love.graphics.print('Hello World!', 400, 300)
```

Try modifying those numbers and see what it does to the text.

LÖVE structure

Open up "main.lua" and take a look at our first line. We defined a function called `love.draw`, which implies there is a table called `love` and we created a key in it called `draw`. Indeed this is the case, but somehow the function was invoked without using having to write `love.draw()` and invoke it ourselves. This requires a high-level explanation of what the engine is doing with our file.

When LÖVE is run, before our main.lua file is ran, a table called `love` is defined as a global variable. We can assign functions to this table (`love.draw`) and access functions already defined in it (`love.graphics.print`).
`love.graphics.print` has two dots in it, so that means the love table probably looks something like:

```
love = {
  draw = nil,
  graphics = {
    print = function() ... end
  }
}
```

The `love` table has a plenty of other tables nested in it, and it puts similar functions in tables together. So all the functions relating to graphics are inside the `love.graphics` table.

Once "main.lua" is done running, we've accessed and modified the `love` table and added some new functionality to it, telling it how to draw to the screen by defining our `love.draw` function. If we define a function with this name, the game engine will see it and invoke it. In fact, it continuously invokes `love.draw` many times a second. To prove my point, let's modify main.lua and make it print a number.

```
local number = 0

love.draw = function()
  number = number + 1
  love.graphics.print(number, 400, 300)
end
```

Each time we go to print the number, we increase it by 1. Run this program and see how quickly the number climbs.

The `love` table is a seemingly complex structure of tables inside tables and functions inside those, but we'll gradually learn the structure and purpose of each thing over the course of this chapter. In the next section, let's take a look at the 2nd and 3rd parameters in `love.graphics.print` and see how they work.

Geometry

If you modified the numbers `400` and `300` in main.lua you will have seen that they move the text. Realizing that they're some kind of coordinates, let's talk about graphs.

When learning about graphs in geometry class, we learned about an x-axis and y-axis and labeled plotted points along the graph. If you wanted to mark (-2, -4) then you would find where -2 on the x-axis intersects with -4 on the y-axis. Knowing that X is horizontal and Y is vertical, if we had (-2, -4) and (1, 2) we could draw it out like this:

These two points could even be connected to form a 2-dimensional line:

Before we get too far on drawing points and lines, let's look back at our function:

```
love.graphics.print("Hello World!", 400, 300)
```

The `400` is the X position and **increasing** it will move the text further to the **right**. **Decreasing** the X position will move the text further to the **left**. The `300` is the Y position, one difference between computers and geometry class is data is calculated from top to bottom, so **increasing** the Y position moves the text **down** and **decreasing** it moves the text **up**. Let's take a look at what our game's graph looks like with the point (5, 3) highlighted:

Notice that the top-left corner of our screen is (0, 0), so if you tried to draw any points with negative numbers they would be drawn off screen where we can't see them. Another thing to note is in the game, the coordinates represent how many **pixels** down and to the right we want to draw. Since computer screens are made of so many pixels, you need to use large numbers to make a noticeable difference.

If we wanted to draw a *polygon* (shape) such as a triangle on this graph, we would have to give three points:

In the same way, we can plot out points in our code and tell it to draw a line to connect the dots. Let's use larger numbers though. Rewrite main.lua to look like this:

```
love.draw = function()
  love.graphics.polygon('line', 50, 0, 0, 100, 100, 100)
end
```

The numbers in this code can be read off in pairs to identify the coordinates: (50, 0), (0, 100), (100, 100) LÖVE's physics engine takes the coordinates, starting at the first point and connects them with a line sequentially. Once it reaches the last point, it draws a line from the last point back to the first to close the shape.

Let's try a rectangle to get some more practice in:

```
love.draw = function()
  local rectangle = {100, 100, 100, 200, 200, 200, 200, 100}
  love.graphics.polygon('line', rectangle)
end
```

Notice this time instead of passing the numbers directly into `love.graphics.polygon`, we put them into a list and passed the list in. Passing in coordinates both ways has the same effect.

Another important thing to think about is if you draw a polygon with 4 or more sides, you need to make sure the points are listed in the correct order. Consider the following example:

If we fed the points into the function in a clockwise or counter-clockwise/anti-clockwise order, the rectangle would be drawn the same either way. If we fed the points in from cross directions, we may accidentally draw a bow tie:

Creating shapes with unclosed sides don't play well with LÖVE's physics engine as such shapes are not physically possible. If you try to do this, you may not see the shape you expect, and perhaps nothing will be drawn.

Exercises

- Take a look at the documentation for the function love.graphics.polygon. The example shows the argument `'line'` is a string and represents the "drawMode". Try changing the "drawMode" from `'line'` to one of the other available options (see the examples or click the drawMode link on the wiki page). What other option is there? How does it work? Try it out and see how it works!
- Try making a polygon with 5 sides/points. Hint: use the square example above as a reference.

Game loop

Another aspect common with game engines is that there is a loop (like a while loop) that continuously runs and keeps the game going. The order that things happen in varies, but the contents more of less look like:

1. Game is started. Load game files.
2. Begin loop.
3. Check for input from keyboard, joystick, or other peripherals.
4. Tick time in game.
5. Redraw the screen.
6. Go back to step 2.

During steps in the game loop, LÖVE invokes certain functions inside the `love` table. For instance, every time the screen needs to be re-drawn, the game loop invokes `love.draw()` if you defined it. In the step where LÖVE checks for user input, if there is user input, it invokes `love.keypressed(PRESSED_KEY)` if we defined it. The `PRESSED_KEY` that is passes in of course depends on what key the user pressed. When defining `love.keypressed`, it may look something like this:

```
love.keypressed = function(pressed_key)
  print('key was pressed:', pressed_key)
end
```

Let's modify main.lua to have a contrived example:

```
local current_color = {1, 1, 1, 1}

love.draw = function()
  local square = {100, 100, 100, 200, 200, 200, 200, 100}

  -- Initialize the square with the default color (white)
  love.graphics.setColor(current_color)
  -- Draw the square
  love.graphics.polygon('fill', square)
end


love.keypressed = function(pressed_key)
  if pressed_key == 'b' then
    -- Blue
    current_color = {0, 0, 1, 1}
  elseif pressed_key == 'g' then
    -- Green
    current_color = {0, 1, 0, 1}
  elseif pressed_key == 'r' then
    -- Red
    current_color = {1, 0, 0, 1}
  elseif pressed_key == 'w' then
    -- White
    current_color = {1, 1, 1, 1}
  end
end
```

When you press any of the keys, "b", "g", "r", or "w", our function `love.keypressed` will be invoked and the variable `pressed_key` will be a string matching one of our letters. This changes `current_color`, which is changing the color being drawn in `love.draw`.

In the next section, let's see how LÖVE handles the "4. Tick time in game." step of the game loop.

Exercises

- Try adding a few more colors to the program. To understand how `love.graphics.setColor` works, see the documentation.
- Make is so that if the escape key is pressed, the function `love.event.quit` is invoked and the game exits. The string to use for the escape key can be found on the wiki's KeyConstant page. Spoilers: the solution can be seen in the next section.

Delta time

Here's what we've learned about the game loop so far:

1. Game is started. Load game files.
    - main.lua is loaded and the `love` table is updated with our modifications.
2. Begin loop.
3. Check for input from keyboard, joystick, or other peripherals.
    - If there was keyboard input and we defined `love.keypressed`, invoke it, passing it information about the pressed key.
4. Tick time in game.
    - ???
5. Redraw the screen.
    - Invoke `love.draw` if we defined it.
6. Go back to step 2.

Let's take a look at the function `love.update`:

```
love.update = function(dt)
  print(dt)
end
```

Note for Windows: Unless you are running LÖVE from the console, you won't see anything printed out. Put this file in the game folder next to main.lua under the name "conf.lua":

```
-- LÖVE configuration file

love.conf = function(t)
  t.console = true
end
```

This configuration file will let us set special parameters for our game. Don't worry too much about what all the options are, but if you're curious then you can find the documentation here. Essentially this will open a console window on Windows to see printed values.

Now run the game and if you weren't seeing the `print(dt)` message display anything you should now see it being invoked many times a second, printing out a decimal number. `dt` stands for *delta time* and it represents the amount of seconds that has passed since the last game loop. If the game loops 4 times a second, that means `love.update` and `love.draw` get invoked 4 times each second as well. The delta time in this case would be roughly `0.25` as roughly 1/4 a second has passed between each time `love.update` was called. Some computers are faster than others so the number of game loops per second will be different. You are likely seeing numbers around `0.01` or smaller, meaning the game is running roughly 100 frames a second. Let's add a counter to the screen like before, but now using delta time.

```
local current_color = {1, 1, 1, 1}
local seconds = 0

love.draw = function()
  local square = {100, 100, 100, 200, 200, 200, 200, 100}

  -- Print a counter clock
  local clock_display = 'Seconds: ' .. seconds
  love.graphics.print(clock_display, 0, 0, 0, 2, 2)

  -- Initialize the square with the default color (white)
  love.graphics.setColor(current_color)
  love.graphics.polygon('fill', square)
end


love.keypressed = function(pressed_key)
  if pressed_key == 'b' then
    -- Blue
    current_color = {0, 0, 1, 1}
  elseif pressed_key == 'g' then
    -- Green
    current_color = {0, 1, 0, 1}
  elseif pressed_key == 'r' then
    -- Red
    current_color = {1, 0, 0, 1}
  elseif pressed_key == 'w' then
    -- White
    current_color = {1, 1, 1, 1}
  end
  if pressed_key == 'escape' then
    love.event.quit()
  end
end

love.update = function(dt)
  -- Add up all the delta time as we get it
  seconds = seconds + dt
end
```

Imagine if we wanted to move a character across the screen but we didn't use delta time. The character would run faster on some computers and slower on others. Computers would keep getting faster and the game would run so fast it would no longer be playable. Delta time solves this issue and we'll be taking advantage of it for everything time-based in our game.

Exercises

- Change the line `local clock_display = 'Seconds: ' .. seconds` so that `seconds` is formatted to display whole numbers. Hint: you will need to use Lua's built-in `math.floor` function to format `seconds`.
- Change the x position of the left side of the square from `100` to `(seconds * 10)` and watch what the square does.

Mapping

Let's sidetrack from LÖVE for a minute to learn about a concept called maps. Not to be confused with overhead maps a player would walk around on in a game, but data maps. We actually did mapping back in chapter 1 when we learned about tables.

```
coins = {
  ["half"] = "50 cents",
  ["quarter"] = "25 cents",
  ["dime"] = "10 cents",
  ["nickel"] = "5 cents",
  ["penny"] = "1 cent"
}

print("Which coin do you have?")
response = io.read()

print("Your coin is worth " .. coins[response] .. ".")
```

Whenever the user typed in a coin, we mapped the coin name to a value by looking up the coin name in the table, or dictionary. So what's the difference between tables, dictionaries, and maps?

- tables are just a data type in Lua that can be used to build data structures like lists and dictionaries
- dictionaries are key-value storages used to centralize similar-purpose data in one place and make it easier to look the data up
- maps are data structures used to translate one type of information to another, and a dictionary is one type of map

Dictionaries are the only types of map we'll be concerned about here, but know that maps generally refer to instances of data structures that do mapping. There are often discrepancies in terminology between mathematics and the various fields in computer science. Don't be surprised if you see dictionaries and maps being used synonymously in other contexts later in life.

Let's do some mapping on our code we previously wrote to get a better feel for them. Remember all those if/elseif statements in main.lua?

```
  if pressed_key == 'b' then
    -- Blue
    current_color = {0, 0, 1, 1}
  elseif pressed_key == 'g' then
    -- Green
    current_color = {0, 1, 0, 1}
  elseif pressed_key == 'r' then
    -- Red
    current_color = {1, 0, 0, 1}
  elseif pressed_key == 'w' then
    -- White
    current_color = {1, 1, 1, 1}
  end
  if pressed_key == 'escape' then
    love.event.quit()
  end
```

We can put all that functionality in a map like this:

```
local key_map = {
  b = function()
    current_color = {0, 0, 1, 1} -- Blue
  end,
  g = function()
    current_color = {0, 1, 0, 1} -- Green
  end,
  r = function()
    current_color = {1, 0, 0, 1} -- Red
  end,
  w = function()
    current_color = {1, 1, 1, 1} -- White
  end,
  -- Close the game
  escape = function()
    love.event.quit()
  end
}
```

This doesn't look any more concise than our previous code, but our goal is to keep the `love.keypressed` function clean in this case. When a key is pressed it will be mapped to a key function we define in `key_map`. Another important thing is these functions could be modular and moved anywhere we need them to be, and even re-used. Let's not go too crazy right now though. We'll keep the key map somewhere near the top.

```
local current_color = {1, 1, 1, 1}
local seconds = 0

local key_map = {
  b = function()
    current_color = {0, 0, 1, 1} -- Blue
  end,
  g = function()
    current_color = {0, 1, 0, 1} -- Green
  end,
  r = function()
    current_color = {1, 0, 0, 1} -- Red
  end,
  w = function()
    current_color = {1, 1, 1, 1} -- White
  end,
  escape = function()
    love.event.quit()
  end
}

love.draw = function()
  local square = {100, 100, 100, 200, 200, 200, 200, 100}

  -- Print a counter clock
  local clock_display = 'Seconds: ' .. math.floor(seconds)
  love.graphics.print(clock_display, 0, 0, 0, 2, 2)

  -- Initialize the square with the default color (white)
  love.graphics.setColor(current_color)
  love.graphics.polygon('fill', square)
end


love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  -- Add up all the delta time as we get it
  seconds = seconds + dt
end
```

If you press a key that isn't part of the map then the new if statement (`if key_map[pressed_key] ...`) will see that key doesn't exist in the map and not do anything. `key_map[pressed_key]()` is the same as saying `key_map['b']()`, `key_map['escape']()` or whatever the value of `pressed_key` was at the time.

The world

A world is a physical space where objects can be created (spawned) and interact. Shapes and other things drawn on the screen are not implicitly part of a world and won't interact with each other unless they are. Multiple worlds can co-exist, but the objects in each world won't interact. Going forward I will refer to these objects as *entities*.

Entities

Entities are made up of different components that allow them to interact. These are the 3 fundamental physical components:

- **shape** - some sort of polygon to give our entity a physical shape that determines the boundaries of the entity
- **body** - holds physical properties such as mass
- **fixture** - attaches a shape to a body

Let's write a new main.lua from scratch and see how these are all wired up. First, a world needs to be defined:

```
local world = love.physics.newWorld(0, 100)
```

`love.physics.newWorld` returns a table, an instance of a world. The table holds functions that allow us to apply attributes to the world. It also holds all the entities in our world, which is currently none on initialization. According to the documentation on `love.physics.newWorld`, our 1st and 2nd parameters set the X and Y gravity on our world. We don't want any sideways gravity, but we'll go ahead and set an arbitrary number for the vertical gravity.

While focusing on the world, we should allow the world to know whenever we get an update to the delta time. A world without time would be frozen; By letting the world know about the passage of time, it can know whether it needs to make an entity fall another meter or two meters…

```
love.update = function(dt)
  world.update(world, dt)
end
```

Actually, let's do one trick here. When calling a function in Lua and the first parameter of the function is the table the function is stored in, you can use a shortcut notation:

```
love.update = function(dt)
  world:update(dt)
end
```

Aside from being easier to write, you'll see this way of invoking functions used all over the place in the LÖVE documentation.

Finally, we'll add an entity to the game in 4 steps:

1. Create a table to store all the pieces of our entity together. Not entirely necessary but we'll learn later why this step makes things easier.
2. Create a body. This will be added to the entity table and the world.
3. Create the shape we want the entity to have.
4. Create a fixture to attach the body and shape together.

```
-- Triangle is the name of our first entity
local triangle = {}
triangle.body = love.physics.newBody(world, 200, 200, 'dynamic')
-- Give the triangle some weight
triangle.body:setMass(32)
triangle.shape = love.physics.newPolygonShape(100, 100, 200, 100, 200, 200)
triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)

love.draw = function()
  love.graphics.polygon('line', triangle.body:getWorldPoints(triangle.shape:get
end
```

After creating the `body` table inside `triangle`, we called `triangle.body.setMass` to set a weight property on our triangle so it can fall. Notice we wrote `triangle.body:setMass(32)`, which is the same as saying `triangle.body.setMass(triangle.body, 32)` but shorter and more conventional to the way the LÖVE documentation writes.

What's going on inside `love.draw` looks pretty crazy so let's break the long line up.

```
  love.graphics.polygon(
    'line',
    triangle.body:getWorldPoints(triangle.shape:getPoints())
  )
```

We've used `love.graphics.polygon` previously so its purpose should already be familiar. The first parameter `'line'` is telling it that we want an outline of a shape drawn. The second parameter is a table containing the points that need to be outlined. To get the triangle's points we call `triangle.shape:getPoints()`, but this only returns the shape of the triangle and the relative position of the points. By then calling `triangle.body:getWorldPoints(triangle.shape:getPoints())` we convert those relative points to their absolute position as that's what the polygon drawing function needs to know so it can draw the triangle exactly where it is supposed to be on the screen.

Let's put it all together and add one more entity into the mix so the two can interact:

```
local world = love.physics.newWorld(0, 100)

-- Triangle is the name of our first entity
local triangle = {}
triangle.body = love.physics.newBody(world, 200, 200, 'dynamic')
-- Give the triangle some weight
triangle.body.setMass(triangle.body, 32)
triangle.shape = love.physics.newPolygonShape(100, 100, 200, 100, 200, 200)
triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)

-- Another entity
local bar = {}
bar.body = love.physics.newBody(world, 200, 450, 'static')
bar.shape = love.physics.newPolygonShape(0, 0, 0, 20, 400, 20, 400, 0)
bar.fixture = love.physics.newFixture(bar.body, bar.shape)


local key_map = {
  escape = function()
    love.event.quit()
  end
}

love.draw = function()
  love.graphics.polygon('line', triangle.body:getWorldPoints(triangle.shape:get
  love.graphics.polygon('line', bar.body:getWorldPoints(bar.shape:getPoints()))
end


love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  world:update(dt)
end
```

This is a lot to digest so don't hesitate to re-read through this code several times if necessary. There were a lot of new functions introduced in this section, so in the next section we'll take a deeper look at the documentation and read more about them and their components.

Exercises

- Try changing the mass (`triangle.body:setMass`) and gravity (`love.physics.newWorld`) and see what happens.

Reading documentation

You typically run into two types of documentation for software: guides and API documentation. Guides would be information on getting started, tutorials, and books. An API (application programming interface) is a portion of software that a programmer writes for his/her program to allow fellow programmers to interact with their application. As for LÖVE's programming interface, most of your interactions with the framework are done through the `love` global variable that the framework purposely exposes. API documentation is the most fundamental form of software documentation because without it, you would not know what all the functions in the program do unless you were resourceful enough to go in and study all the source code and figure each function out on your own.

The documentation for LÖVE (bookmark this!) is written in a wiki style where each table and function has an article that describes how to use it. From here you should see many *modules* listed. Click on `love.physics`. Again, `love.physics` is just a table with functions in it. So within the article we see each of the functions stored in it including the functions `love.physics.newBody` which we used to create our entities' bodies, `love.physics.newPolygonShape` which we used to create their shapes, and `love.physics.newFixture` which we used to create their fixtures. We also see `love.physics.newWorld` which created our world table. Let's look at the first function's article.

Clicking on the article for `love.physics.newBody` we get a synopsis showing how the function might be used, along with a description of its parameters and what the function returns. Over the course of different versions of the framework, functions may be modified so in the case of this function you can see examples of how to use it differently in different versions of LÖVE. Since the function lists that it returns a body, click the link to go over to the article on the body table's documentation.

There's a lot of functions here that set properties on the body and get properties from the body. One of them is the `body:setMass` function we used to give our entity weight. We can see that it takes one parameter and that the parameter is meant to simulate how many kilograms of mass the body has. We originally told it that our triangle in the last section weighed 32 kilograms, which if you think the object fell too fast or too slow then you may need to adjust your world's gravity to match your expectation.

Now let's go back to `love.physics` for a moment and take a look at one of the other components we added to our previous code, the fixture. We previously created a fixture table by calling `triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)`. However, we haven't seen any of these functions in the fixture table that could come in handy. For instance, we could give our triangle bounciness by invoking `fixture:setRestitution`. Our triangle fixture is named `triangle.fixture` though, not `fixture`. If 0 is no bounciness (default) and 1 is 100%, try modifying the game code and adding a restitution of 75%:

```
triangle.fixture:setRestitution(0.75)
```

Try running the game and see how that works. If you set the restitution to `1` or higher then the triangle won't stop bouncing and will bounce itself right off the screen.

Callbacks

Let's backtrack now to the main article about the love table. If you scroll down a little on the page, you'll see a section titled "Callbacks" that contains some functions we've become familiar with such as `love.draw` and `love.update`. This is a list of all the functions in the game loop that we have and haven't talked about yet. A "callback" is a function you create and give to an API (the `love` table in this case) that will later get invoked as needed. Creating functions with these names allow you to tap into specific portions of the game loop and trigger your own events.

Let's take a look at the `love.keypressed` callback for instance. In the synopsis you see that it has 3 parameters (the documentation mistakenly calls parameters "arguments"). We used the first parameter `key` to see what key was pressed. If you ever need to know what keys are available, you can click on the link provided next to the parameter name, `KeyConstant` to see a well-defined list of all the available key strings passed in to this parameter. The second parameter `scancode` we didn't talk about, but it has a well-defined `Scancode` article explaining what it is. If you are not familiar with scancodes, take a minute to read it and perhaps you'll learn about a feature you may want to use in your game.

One more callback we'll look at while we're here is `love.focus`. Take a moment to stop here and read what it does and what parameters it takes before continuing. Now it would be really cool if we were making a game and when the user switched to another application window, the game automatically paused for the user. So first let's start by implementing a pause feature in our earlier game code:

```
local world = love.physics.newWorld(0, 9.81 * 128)

-- Triangle is the name of our first entity
local triangle = {}
triangle.body = love.physics.newBody(world, 200, 200, 'dynamic')
-- Give the triangle some weight
triangle.body.setMass(triangle.body, 32)
triangle.shape = love.physics.newPolygonShape(100, 100, 200, 100, 200, 200)
triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)
triangle.fixture:setRestitution(0.75)

-- Another entity
local bar = {}
bar.body = love.physics.newBody(world, 200, 450, 'static')
bar.shape = love.physics.newPolygonShape(0, 0, 0, 20, 400, 20, 400, 0)
bar.fixture = love.physics.newFixture(bar.body, bar.shape)

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  love.graphics.polygon('line', triangle.body:getWorldPoints(triangle.shape:get
  love.graphics.polygon('line', bar.body:getWorldPoints(bar.shape:getPoints()))
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  if not paused then
    world:update(dt)
  end
end
```

Notice the 3 changes:

- We added a boolean called `paused` and set it to false
- We added a new function to `key_map` so that when `"space"` is pressed, the value of `paused` is set to `not paused`. `not` is an operator for booleans we previously didn't discuss. It simply says "the opposite of this boolean". So if `paused` is `true`, then setting `paused` to `not paused` will set it to `false`.
- Lastly, inside `love.update` to told the world to update only if we are `not paused`. So the passage of time in the game world will cease when pressing the space key.

Exercises

- Now with the documentation in hand, define `love.focus` and make it so the game pauses when the user clicks away from the game window.
- Bonus: make the game print a text saying that the game is paused when `paused` is `true`. Go find the documentation for `love.graphics.print` to see an example on displaying text.

Modules and organization

Eventually when you start writing real programs, you realize if you keep all the code in one file that things can get a bit messy. Putting your code in separate files helps you not only keep your different pieces of code separated from each other, but it helps you visualize the structure of your program.

Let's start with a single *main.lua* file and we'll then split it into different files:

```
local my_cool_functon = function()
  love.graphics.print('This function came from function-module.lua', 100, 100,
end

local my_cool_table = {}
my_cool_table.print_stuff = function()
  love.graphics.print('This function came from table-module.lua', 100, 200, 0,
end

print('my_cool_function is', my_cool_function)
print('my_cool_table is', my_cool_table)
```

When we go to run the code, we get a blank window because we're not drawing anything. We do see our print statements output our function and table to the console though:

```
my_cool_function is     function: 0x41f2abc0
my_cool_table is        table: 0x41f2aa08
```

Modules and `require`

Think of your Lua files as giant functions that get invoked whenever you load the file. Just like a function, you can *return* values from your files. If you load one Lua file from another, you will get whatever value is returned. Let's modify our previous code. First define these two new files in the game folder:

function-module.lua

```
return function()
  love.graphics.print('This function came from function-module.lua', 100, 100,
end
```

table-module.lua

```
local my_cool_table = {}
my_cool_table.print_stuff = function()
  love.graphics.print('This function came from table-module.lua', 100, 200, 0,
end

return my_cool_table
```

Then update **main.lua**:

```
local function_module = require('function-module')
local table_module = require('table-module')

print('function_module is', function_module)
print('table_module is', table_module)
```

Let's start from the top. In *function-module.lua* we write a return statement that returns a function with no name. We don't invoke the function, we just return it as a value the same way a function may return a number or string. Likewise in *table-module.lua* we defined a table (with a function in it) and returned the table on the last line of the file. The function name and local variable name `my_cool_table` is inconsequential and can't be seen outside the *table-module.lua* file as modules have their own scope.

In *main.lua* we are *requiring* function-module using a built-in Lua function, `require`. `require` takes one argument, a string that equals the name of your file and it then invokes that file and returns back a function which we assign to a new variable `function_module`. We then do the same thing for *table-module.lua*. We require it, which invokes it and returns back whatever that file returns. In this case is a table. Notice that when we pass in the filenames as arguments we just give the first part of the filename without the extension ".lua" at the end. This function expects that any file it is requiring is a Lua file.

After we required the files, we printed the values of those variables, so you should see the results of the print statements appear in the console like before:

```
function_module is      function: 0x40479548
table_module is table: 0x40479bc8
```

We pulled in the return values from the other two files into our *main.lua* file and printed the values, but since we didn't invoke the functions from those two files then we got a blank game window when running the program. Let's define a `love.draw` in *main.lua* like before and invoke the functions we got back from both modules:

```
local function_module = require('function-module')
local table_module = require('table-module')

print('function_module is', function_module)
print('table_module is', table_module)

love.draw = function()
  function_module()
  table_module.print_stuff()
end
```

We were able to invoke the functions and use the returned data as if it were all in the same file.

Organizing modules

It may help to see a real example of using modules in a game, so let's take our previous game code from 2.8 - Reading documentation and see how we can separate out functionality. The first thing we did in our game was define a world, so let's start by putting our world-related code in a dedicated file named **world.lua**:

```
-- world.lua

local world = love.physics.newWorld(0, 9.81 * 128)

return world
```

Remember that you need the return statement at the end of your files or else the code will return `nil` when you go to require it and this could cause all kinds of unexepcted errors when you run it. Next let's create a folder named **entities** that we can keep all our game entities in. We plan on creating more entities so it will help to keep them all together. In the entities folder, create a file and name it **triangle.lua**. We'll cut all the code from the original main.lua that related to our triangle entity and put it here:

```
-- entities/triangle.lua

local world = require('world')

local triangle = {}
triangle.body = love.physics.newBody(world, 200, 200, 'dynamic')
triangle.body.setMass(triangle.body, 32)
triangle.shape = love.physics.newPolygonShape(100, 100, 200, 100, 200, 200)
triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)
triangle.fixture:setRestitution(0.75)

return triangle
```

Notice how we are requiring the `world` table from world.lua, because we need to access that table in this entity's file so we can add the entity to the world. We also need to do the same thing as above with the **bar** entity:

```
-- entities/bar.lua

local world = require('world')

local bar = {}
bar.body = love.physics.newBody(world, 200, 450, 'static')
bar.shape = love.physics.newPolygonShape(0, 0, 0, 20, 400, 20, 400, 0)
bar.fixture = love.physics.newFixture(bar.body, bar.shape)

return bar
```

Now our **main.lua** should only contain our key map and `love` functions:

```
-- main.lua

local bar = require('entities/bar')
local triangle = require('entities/triangle')
local world = require('world')

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  love.graphics.polygon('line', triangle.body:getWorldPoints(triangle.shape:ge
  love.graphics.polygon('line', bar.body:getWorldPoints(bar.shape:getPoints())
end

love.focus = function(focused)
  if not focused then
    paused = true
  end
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end


love.update = function(dt)
  world:update(dt)
end
```

Our two entities and world get pulled into main.lua and everything should run exactly as before. One thing to note is that even though we require world.lua 3 times in our code, it is the same world and not 3 copies. This is because Lua knows to only run a module the first time you require it and not invoke it again.

Once it runs the first time, the returned results are stored in memory for the next time you try to require it. We can prove this by adding a print statement to **world.lua**:

```
-- world.lua

print("This is the world")

local world = love.physics.newWorld(0, 9.81 * 128)

return world
```

How many times does `"This is the world"` get printed to the console?

Exercises

- Try creating two new modules; One that returns a string and one that returns a number.

Collision callbacks

When writing a game such as a platformer you may want something special to happen when two objects collide. If it's a powerup, for instance, you may want the powerup to *despawn* (be removed from the world) if a player touches it and then give the player a special ability (think Mario and mushrooms). If a player and an enemy bump into each other, you may want the player's health to decrement. The *world* table has a method that allows you to program in functionality like this for when two entities collide. It does this by allowing you to create callbacks as we learned before, but these callbacks are triggered before, during, or after collision. Take a look at World:setCallbacks.

If you look at the parameters for `World:setCallbacks`, you see it can take four functions. The description of these parameters helps explain when the functions will be called. `beginContact` and `endContact` should be self explanatory; They happen at the point where contact begins and ends in a collision, but `preSolve` and `postSolve` may not be as obvious. Nonetheless, let's edit the previously-created **world.lua** file and write some collision callbacks to test this functionality.

```
-- world.lua

local begin_contact_counter = 0
local end_contact_counter = 0
local pre_solve_counter = 0
local post_solve_counter = 0

local begin_contact_callback = function()
  begin_contact_counter = begin_contact_counter + 1
  print('beginContact called ' .. begin_contact_counter .. ' times')
end

local end_contact_callback = function()
  end_contact_counter = end_contact_counter + 1
  print('endContact called ' .. end_contact_counter .. ' times')
end

local pre_solve_callback = function()
  pre_solve_counter = pre_solve_counter + 1
  print('preSolve called ' .. pre_solve_counter .. ' times')
end

local post_solve_callback = function()
  post_solve_counter = post_solve_counter + 1
  print('postSolve called ' .. post_solve_counter .. ' times')
end

local world = love.physics.newWorld(0, 9.81 * 128)

world:setCallbacks(
  begin_contact_callback,
  end_contact_callback,
  pre_solve_callback,
  post_solve_callback
)

return world
```

Try it out. Every time one of the callbacks is invoked, it will increment its own number by 1 then print a message to the console telling you how many times it has been invoked. It's clear right away that `pre_solve_callback` and `post_solve_callback` get invoked many more times than `begin_contact_callback` and `end_contact_callback` in this situation.

Unless you've edited the behavior of the triangle entity, it will bounce a bit (because of the triangle's restitution). Once it bounces and neither corner or side is touching the floor underneath, the contact ends. This process is repeated every time it bounces. Once the triangle settles down it will slide a bit, maybe even a lot... like an air hockey puck. This is because our triangle and bar have no friction between them to prevent that. Anyways, this is good because it allows us to see that while the triangle is sliding it is still making contact. While the triangle is sliding and still making contact, the `pre_solve_callback` and `post_solve_callback` will continue to get called with every frame of movement.

Pretend our triangle was a futuristic race car moving across a neon strip of road that recharged the vehicle. You could start increasing the race car's power meter inside `begin_contact_callback` as the car makes contact with that section of road and then stop increasing power when `end_contact_callback` is invoked. This could work pretty well, but then the player may try parking for a moment on the power strip and continue to gain health as long as they want. So another approach could be to only increase the power meter as the player continues to move *and* make contact with the road, increasing health by 1 point every time the `post_solve_callback` function is invoked.

You don't necessarily need to use all of these callbacks, so you could just pass in an empty function or `nil` to `World:setCallbacks` for the arguments you don't need.

Without knowing what entities are colliding, the collision callbacks aren't very useful. Luckily, our callbacks have parameters of their own that we can access. Let's modify the code again and check out those parameters:

```
-- world.lua

-- Called at the beginning of any contact in the world. Parameters:
-- {fixture} fixture_a - first fixture object in the collision.
-- {fixture} fixture_b - second fixture object in the collision.
-- {contact} contact - world object created on and at the point of
--   contact. When sliding along an object, there may be several.
--   See further: https://love2d.org/wiki/Contact
local begin_contact_callback = function(fixture_a, fixture_b, contact)
  print(fixture_a, fixture_b, contact, 'beginning contact')
end

local end_contact_callback = function(fixture_a, fixture_b, contact)
  print(fixture_a, fixture_b, contact, 'ending contact')
end

local pre_solve_callback = function(fixture_a, fixture_b, contact)
  print(fixture_a, fixture_b, contact, 'about to resolve a contact')
end

local post_solve_callback = function(fixture_a, fixture_b, contact)
  print(fixture_a, fixture_b, contact, 'just resolved a contact')
end

local world = love.physics.newWorld(0, 9.81 * 128)

world:setCallbacks(
  begin_contact_callback,
  end_contact_callback,
  pre_solve_callback,
  post_solve_callback
)

return world
```

This should print out some information in the console similar to:

```
Fixture: 0x561020bf8570 Fixture: 0x561020bf7350 Contact: 0x561020bf7480 beginn
Fixture: 0x561020bf8570 Fixture: 0x561020bf7350 Contact: 0x561020bf7480 about t
Fixture: 0x561020bf8570 Fixture: 0x561020bf7350 Contact: 0x561020bf7480 just re
Fixture: 0x561020bf8570 Fixture: 0x561020bf7350 Contact: 0x561020bf7480 ending
```

`Fixture: 0x561020bf8570` is a text representation of our first entity's fixture. The `0x56...` is the memory address of the fixture to help identify it, although this information still doesn't tell us which entity this fixture belongs to. We also printed out a contact table, which contains a set of functions just like the entities. This instance of a contact provides information such as where the contact happened and how much velocity was involved.

Let's work on modifying the print statements so we can collect more useful information on these collisions. There is a pair of functions on every fixture that let's you set any arbitrary data you want on that fixture and another function to get that data back out the fixture. These functions are called `Fixture:setUserData` and `Fixture:getUserData`. These functions can be used to set a name or ID on the fixture to help us identify what entity it belongs to. We can accomplish this by first modifying our entity files and passing some strings to `Fixture:setUserData`:

```
-- entities/bar.lua

local world = require('world')

local bar = {}
bar.body = love.physics.newBody(world, 200, 450, 'static')
bar.shape = love.physics.newPolygonShape(0, 0, 0, 20, 400, 20, 400, 0)
bar.fixture = love.physics.newFixture(bar.body, bar.shape)
bar.fixture:setUserData('bar')

return bar
```

```
-- entities/triangle.lua

local world = require('world')

local triangle = {}
triangle.body = love.physics.newBody(world, 200, 200, 'dynamic')
triangle.body.setMass(triangle.body, 32)
triangle.shape = love.physics.newPolygonShape(100, 100, 200, 100, 200, 200)
triangle.fixture = love.physics.newFixture(triangle.body, triangle.shape)
triangle.fixture:setRestitution(0.75)
triangle.fixture:setUserData('triangle')

return triangle
```

Now go back to the world's collision callbacks and you can easily extract this information back out of the fixtures:

```
-- world.lua

-- Called at the beginning of any contact in the world. Parameters:
-- {fixture} fixture_a - first fixture object in the collision.
-- {fixture} fixture_b - second fixture object in the collision.
-- {contact} contact - world object created on and at the point of
--   contact. When sliding along an object, there may be several.
--   See further: https://love2d.org/wiki/Contact
local begin_contact_callback = function(fixture_a, fixture_b, contact)
  local name_a = fixture_a:getUserData()
  local name_b = fixture_b:getUserData()

  print(name_a, name_b, contact, 'beginning contact')
end

local end_contact_callback = function(fixture_a, fixture_b, contact)
  local name_a = fixture_a:getUserData()
  local name_b = fixture_b:getUserData()

  print(name_a, name_b, contact, 'ending contact')
end

local pre_solve_callback = function(fixture_a, fixture_b, contact)
  local name_a = fixture_a:getUserData()
  local name_b = fixture_b:getUserData()

  print(name_a, name_b, contact, 'about to resolve a contact')
end

local post_solve_callback = function(fixture_a, fixture_b, contact)
  local name_a = fixture_a:getUserData()
  local name_b = fixture_b:getUserData()

  print(name_a, name_b, contact, 'just resolved a contact')
end

local world = love.physics.newWorld(0, 9.81 * 128)

world:setCallbacks(
  begin_contact_callback,
  end_contact_callback,
  pre_solve_callback,
  post_solve_callback
)

return world
```

Ah, now we can see which fixture is colliding which!

```
bar     triangle        Contact: 0x55bf29c07590 beginning contact
bar     triangle        Contact: 0x55bf29c07590 about to resolve a contact
bar     triangle        Contact: 0x55bf29c07590 just resolved a contact
bar     triangle        Contact: 0x55bf29c07590 ending contact
```

Exercises

- Modify the print statements in each collision callback to print out the coordinates where the entities' fixtures are making contact. You can find the information you need to do this in the documentation for the contact table mentioned above.

Breakout (part 1): more entity practice

Let's bring all these concepts together by making another game, a breakout clone. The requirements are pretty simple:

- The objective of the game is to destroy all the bricks on the screen
- The player controls a "paddle" entity that hits a ball
- The ball destroys the bricks
- The ball needs to stay within the boundaries of the screen
- If the ball touches the bottom of the screen (below the paddle), the game ends

If you still have the code from the previous sections, feel free to copy the folder naming the new one "breakout" or whatever you want your breakout clone to be called. At the end of this section there will be a link to all the source code to use as a reference in case you get stuck. This may be time consuming, but I encourage you to type out each section and stop to understand what it is you are typing. If you copy, paste, and don't read then it will be easy to get lost in this chunk of the chapter as things will move fast.

The first modification we'll make is to set a specific window size so no matter which version of LÖVE you're on we're working with the same window proportions and entity dimensions. To do this, open of **conf.lua** or create it if you don't have it and put in the following code:

```
-- conf.lua
-- LÖVE configuration file

love.conf = function(t)
  t.console = true         -- Enable the debug console for Windows.
  t.window.width = 800     -- Game's screen width (number of pixels)
  t.window.height = 600    -- Game's screen height (number of pixels)
end
```

The *conf*, or *configuration* file lets you define a callback in the `love` table that modifies the game engine's configuration on load. You can read more about all the interesting things you can do with it here but most of its features won't be necessary for our simple game.

The next modification we'll make is deleting the entities from the last section. Let's create new entities to represent the ball and paddle:

```
-- entities/ball.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 200, 200, 'dynamic')
entity.body:setMass(32)
entity.body:setLinearVelocity(300, 300)
entity.shape = love.physics.newCircleShape(0, 0, 10)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setRestitution(1)
entity.fixture:setUserData(entity)

return entity
```

```
-- entities/paddle.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 200, 560, 'static')
entity.shape = love.physics.newRectangleShape(180, 20)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setUserData(entity)

return entity
```

Before we try and run anything, take a look at a few things we've done differently in defining these entities than we've previously done.

- In **ball.lua** we are defining a circle shape instead of a polygon. This means we have no sides or corner points we can reference when spawning or tracking the position of this object. Circles have to be tracked from their center point and their boundaries by their radius.
- In this file we're using `Body:setLinearVelocity` to apply movement on the ball in a specific direction when the entity spawns.
- In **paddle.lua** we are defining a polygon shape, but instead of specifying each point we are using the `love.physics.newRectangleShape` function to define the shape. This will still generate a polygon as before, but instead of specifying each point in the shape we are giving a height and width and allowing it to figure out the shape we want based on those two parameters. Think of it as a shortcut version of the `love.physics.newPolygonShape` function.
- The paddle has a static body while the ball is dynamic. What this entails is the ball will be affected by the paddle but the paddle won't be affected by the ball. Even though the paddle is static, it can be manually repositioned as we'll do later with buttons.
- In both entity files, we are passing the full entity table as the fixture user data instead of just a string name like before. This will allow us to easily access the entire entity inside the collision callback as we'll see later. You'll want to go back and compare that code from the Collision Callbacks section to these entities, but don't worry if it doesn't make complete sense yet.

Now we need to modify **main.lua** to load up our new entities:

```
-- main.lua

local paddle = require('entities/paddle')
local ball = require('entities/ball')
local world = require('world')

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  local ball_x, ball_y = ball.body:getWorldCenter()
  love.graphics.circle('fill', ball_x, ball_y, ball.shape:getRadius())
  love.graphics.polygon(
    'line',
    paddle.body:getWorldPoints(paddle.shape:getPoints())
  )
end

love.focus = function(focused)
  if not focused then
    paused = true
  end
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  if not paused then
    world:update(dt)
  end
end
```

Take note of a few things we're doing here:

- For drawing the circle, we need to invoke `love.graphics.circle`.
- For drawing the paddle, we still invoke `love.graphics.polygon` as the rectangle is still a polygon shape.

Now let's remove any print statements in **world.lua** just to clean things up. We'll leave the callbacks there since we may use them later but we'll leave them empty for now. We'll also set the gravity to `0` because we want the ball to bounce freely like in the real Breakout game and not lose any momentum.

```
-- world.lua

-- Called at the beginning of any contact in the world. Parameters:
-- {fixture} fixture_a - first fixture object in the collision.
-- {fixture} fixture_b - second fixture object in the collision.
-- {contact} contact - world object created on and at the point of
--   contact. When sliding along an object, there may be several.
--   See further: https://love2d.org/wiki/Contact
local begin_contact_callback = function(fixture_a, fixture_b, contact)
end

local end_contact_callback = function(fixture_a, fixture_b, contact)
end

local pre_solve_callback = function(fixture_a, fixture_b, contact)
end

local post_solve_callback = function(fixture_a, fixture_b, contact)
end

local world = love.physics.newWorld(0, 0)

world:setCallbacks(
  begin_contact_callback,
  end_contact_callback,
  pre_solve_callback,
  post_solve_callback
)

return world
```

What happens if you run the game now? The ball flies right off the screen without consequence. There are a couple different ways of preventing the ball from moving off screen. Possibly the most simple approach is to put up some walls. Can you guess what the code to those walls may look like? Yup, they will be entities similar to the paddle except that they just sit at the edges of the screen. Let's create some entities for that purpose:

```
-- entities/boundary-top.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 400, 5, 'static')
entity.shape = love.physics.newRectangleShape(800, 10)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setUserData(entity)

return entity
```

Take a look at these numbers for a minute. For the location of the body we specified 400 pixels. So starting from the top left corner and moving right along the x-axis we've specified the very center of an 800-pixel-wide window. The reason we've done this is because we want the top and bottom wall boundaries to stretch 800 pixels wide, the entire length of the window, and when calling `newBody` and spawning an entity's body it will spawn the center point of the entity shape at that location. Not all entity shapes are square, or even polygonal, so it is simplest for the game engine to center the shape on the body's spawn point

rather than using another point of reference on the shape, like the top left corner of the shape (not all shapes have corners). In fact, the ball and paddle spawned centered on the location we gave for their bodies.

So we made the walls 800 pixels wide and just to give it a little visibility we made them 10 pixels tall. You would think we'd spawn the wall at the very top of the screen (0 pixels on the y-axis,) but since our walls will be centered to the spawn points we should move down half the height of the wall if we want it all to appear on screen.

Now the boundary on the bottom will have the same dimensions, but it will be spawned at the bottom of the screen (600 pixels) minus half the height of the wall (5 pixels):

```
-- entities/boundary-bottom.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 400, 595, 'static')
entity.shape = love.physics.newRectangleShape(800, 10)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setUserData(entity)

return entity
```

The left and right boundaries will follow the same pattern except they will be the height of the screen instead of the width of the screen:

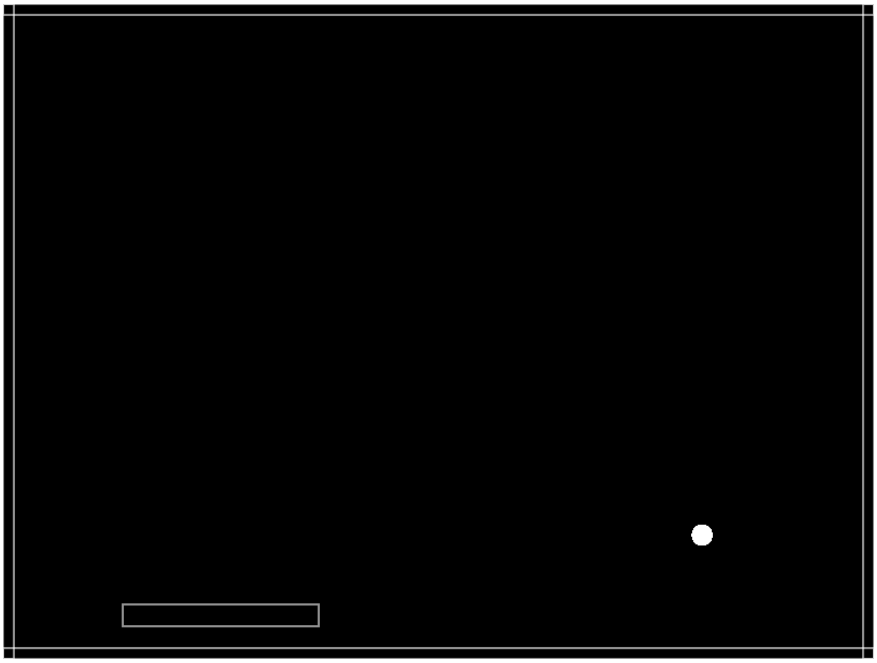
```
-- entities/boundary-left.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 5, 300, 'static')
entity.shape = love.physics.newRectangleShape(10, 600)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setUserData(entity)

return entity
```

```
-- entities/boundary-right.lua

local world = require('world')

local entity = {}
entity.body = love.physics.newBody(world, 795, 300, 'static')
entity.shape = love.physics.newRectangleShape(10, 600)
entity.fixture = love.physics.newFixture(entity.body, entity.shape)
entity.fixture:setUserData(entity)

return entity
```

We won't see these entities until we *require* them and draw them on the screen. So modify **main.lua** to require and draw them the same way we do the ball and paddle:

```
-- main.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_left = require('entities/boundary-left')
local boundary_right = require('entities/boundary-right')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local ball = require('entities/ball')
local world = require('world')

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  love.graphics.polygon('line', boundary_bottom.body:getWorldPoints(boundary_bo
  love.graphics.polygon('line', boundary_left.body:getWorldPoints(boundary_left
  love.graphics.polygon('line', boundary_right.body:getWorldPoints(boundary_rig
  love.graphics.polygon('line', boundary_top.body:getWorldPoints(boundary_top.s
  local ball_x, ball_y = ball.body:getWorldCenter()
  love.graphics.circle('fill', ball_x, ball_y, ball.shape:getRadius())
  love.graphics.polygon('line', paddle.body:getWorldPoints(paddle.shape:getPoin
end

love.focus = function(focused)
  if not focused then
    paused = true
  end
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  if not paused then
    world:update(dt)
  end
end
```

When you run the game, you should see pretty much the same thing as this:



If you missed anything or are having issues, here's a copy of the completed source code for this section:
https://github.com/RVAGameJams/learn2love/tree/master/code/breakout-1

Looking back at our list of minimal requirements we've already completed one thing on our list:

> The ball needs to stay within the boundaries of the screen

There's still quite a bit more work to complete this list so let's continue in the next section.

Exercises

- Maybe it would be better if the boundary lines were even with the screen so we couldn't see them. Modify the boundary positions so it looks like the ball is bouncing off the edge of the screen.
- What happens if we `require` the boundaries but don't draw them in `love.draw`? Does the game still work?

Breakout (part 2): entity management

Review

In the previous section we made a checklist of requirements and accomplished one of them:

- The objective of the game is to destroy all the bricks on the screen
- The player controls a "paddle" entity that hits a ball
- The ball destroys the bricks
- ✓ The ball needs to stay within the boundaries of the screen
- If the ball touches the bottom of the screen, the game ends

In the previous exercise, the goal was to move the boundaries so they were just off screen. This gives the effect that the ball is bouncing off the edges of the game window.

```
-- entities/boundary-bottom.lua
entity.body = love.physics.newBody(world, 400, 606, 'static')
```

```
-- entities/boundary-left.lua
entity.body = love.physics.newBody(world, -6, 300, 'static')
```

```
-- entities/boundary-right.lua
entity.body = love.physics.newBody(world, 806, 300, 'static')
```

```
-- entities/boundary-top.lua
entity.body = love.physics.newBody(world, 400, -6, 'static')
```

Here they have been moved 6 pixels off screen just to use even numbers and make calculation easier. Previously we also raised the question of whether or not the boundaries would work if we still `require` 'd them in **main.lua** but didn't draw them in `love.draw`. The answer is they still work but we don't see them. Since they are off screen, that doesn't matter anyway and we can save our program from doing extra work:

```
-- main.lua
love.draw = function()
  local ball_x, ball_y = ball.body:getWorldCenter()
  love.graphics.circle('fill', ball_x, ball_y, ball.shape:getRadius())
  love.graphics.polygon('line', paddle.body:getWorldPoints(paddle.shape:getPoi
end
```

Entity list

Let's think about the problem of brick entities for a minute. We could create an entity file for each brick, but they are more or less the same except that they spawn in different spots. Imagine making 50 different entity files and then inside `love.draw` making 50 lines to draw each brick and so on. What we can instead do is make an entity file for 1 brick then make a list with 50 copies of it (or however many brick copies we end up fitting on the screen). We can then loop over this list to draw the bricks.

Let's first create the brick entity file:

```
-- entities/brick.lua

local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'static')
  entity.shape = love.physics.newRectangleShape(50, 20)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  return entity
end
```

Instead of returning an entity in this file, we returned a function that takes an x-position and y-position as parameters. When the function gets invoked wherever it is required, it will generate a new entity with those coordinates for its spawn point. Here's how we can use it:

```
-- main.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_left = require('entities/boundary-left')
local boundary_right = require('entities/boundary-right')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local ball = require('entities/ball')
local brick = require('entities/brick')

local entities = {
  brick(100, 100),
  brick(200, 100),
  brick(300, 100)
}


local world = require('world')

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  local ball_x, ball_y = ball.body:getWorldCenter()
  love.graphics.circle('fill', ball_x, ball_y, ball.shape:getRadius())
  love.graphics.polygon('line', paddle.body:getWorldPoints(paddle.shape:getPoi

  for _, entity in ipairs(entities) do
    love.graphics.polygon('fill', entity.body:getWorldPoints(entity.shape:getPo
  end
end

love.focus = function(focused)
  if not focused then
    paused = true
  end
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  if not paused then
    world:update(dt)
  end
end
```

We made an entity table with a list of brick entities in it, then in `love.draw` we made a for loop to draw each entity in the list. Before we change anything else try running the game and taking a look that the bricks appear and that everything works.

Rule of single responsibility

Our goal for the rest of this section will be to simplify entity management. One strategy we'll have for doing this is to think of each file in our game as having a single responsibility. A good sign that we're doing this is **main.lua** is very small and easy to scan over with the eyes and digest.

So what is the responsibility of **main.lua**?

- Create the callback functions necessary to run the game.

Here's some things it is doing that don't fit that responsibility:

- Load and store all the entities
- Figure out how to draw each type of entity in `love.draw`
- Store a map of keypresses

Imagine our game is an organization and each file is a role in the company. Our main file is like the secretary that knows how to handle requests from outsiders. If somebody called asking the secretary about building-maintenance issues, the secretary wouldn't grab plumbing tools and take care of the problem but rather dispatch the person whose responsibility is that exact kind of problem. As the owner of this organization we should know everyone's roles so it's easy to know where each responsibility lies. It will make it easier for us to grow the company to the size we desire.

One easy improvement is to not write out all the instructions for drawing each entity within the main file, but rather let each entity file be responsible for every feature of that entity, including how to draw that entity. We may want to get fancy later and draw bricks in different colors, for instance. That could get complicated and we don't want the main file to retain a bunch of code about brick colors and such.

Modifying the entities is as easy as creating `draw` functions in the entity tables:

```
-- entities/brick.lua

local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'static')
  entity.shape = love.physics.newRectangleShape(50, 20)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  entity.draw = function(self)
    love.graphics.polygon('fill', self.body:getWorldPoints(self.shape:getPoints
  end

  return entity
end
```

```
-- entities/paddle.lua

local world = require('world')

return function(pos_x, pos_y)
  local entity = {}
  entity.body = love.physics.newBody(world, pos_x, pos_y, 'static')
  entity.shape = love.physics.newRectangleShape(180, 20)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  entity.draw = function(self)
    love.graphics.polygon('line', self.body:getWorldPoints(self.shape:getPoints
  end

  return entity
end
```

```
-- entities/ball.lua

local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'dynamic')
  entity.body:setMass(32)
  entity.body:setLinearVelocity(300, 300)
  entity.shape = love.physics.newCircleShape(0, 0, 10)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setRestitution(1)
  entity.fixture:setUserData(entity)

  entity.draw = function(self)
    local self_x, self_y = self.body:getWorldCenter()
    love.graphics.circle('fill', self_x, self_y, self.shape:getRadius())
  end

  return entity
end
```

Go ahead and make *all* the entities return a function with `x_pos` and `y_pos` parameters and we'll just add everything to the entity list like the bricks. Don't forget to change out the numbers in the `love.physics.newBody(world, 200, 200, 'dynamic')` with the arguments being passed in by the function: `love.physics.newBody(world, x_pos, y_pos, 'dynamic')` . For the boundaries entity files there is no need for `entity.draw` functions, but still make them return functions with the two parameters. Now update the `entities` list in **main.lua** to include all the entities:

```
-- main.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_left = require('entities/boundary-left')
local boundary_right = require('entities/boundary-right')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local ball = require('entities/ball')
local brick = require('entities/brick')

local entities = {
  boundary_bottom(400, 606),
  boundary_left(-6, 300),
  boundary_right(806, 300),
  boundary_top(400, -6),
  paddle(300, 500),
  ball(200, 200),
  brick(100, 100),
  brick(200, 100),
  brick(300, 100)
}


local world = require('world')

-- Boolean to keep track of whether our game is paused or not
local paused = false

local key_map = {
  escape = function()
    love.event.quit()
  end,
  space = function()
    paused = not paused
  end
}

love.draw = function()
  for _, entity in ipairs(entities) do
    if entity.draw then entity:draw() end
  end
end

love.focus = function(focused)
  if not focused then
    paused = true
  end
end

love.keypressed = function(pressed_key)
  -- Check in the key map if there is a function
  -- that matches this pressed key's name
  if key_map[pressed_key] then
    key_map[pressed_key]()
  end
end

love.update = function(dt)
  if not paused then
    world:update(dt)
  end
end
```

Take a look at our `love.draw` function. It is much simpler now that it no longer needs to know how to draw each entity. It just asks the entity if it knows how to draw itself and if it does it tells it to do so. Remember that invoking `entity:draw()` is just shorthand for writing `entity.draw(entity)` because of the `:`.

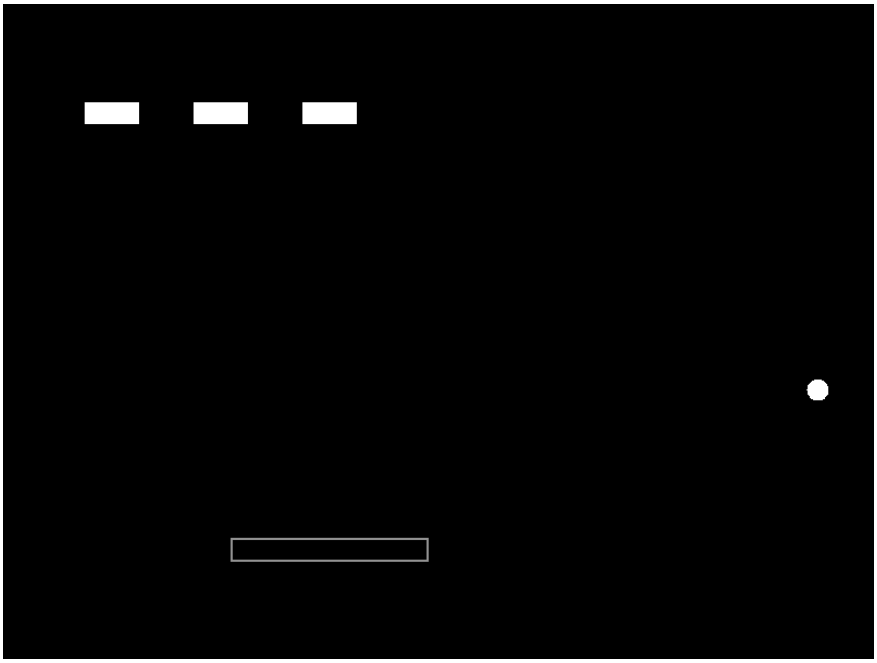
Ok, but putting the entities in a list didn't clean up this file. Now this file is responsible for knowing where to spawn the entities and having them in a list just makes this file bigger. Well you see the reason we put them in a list is because we want to make a new game file called **entities.lua** that will be responsible for loading, spawning, and storing all the entities when the game starts up. Create a new file then cut all the entity `require` statements and the entity list and paste it in the new file:

```
-- entities.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_left = require('entities/boundary-left')
local boundary_right = require('entities/boundary-right')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local ball = require('entities/ball')
local brick = require('entities/brick')

return {
  boundary_bottom(400, 606),
  boundary_left(-6, 300),
  boundary_right(806, 300),
  boundary_top(400, -6),
  paddle(300, 500),
  ball(200, 200),
  brick(100, 100),
  brick(200, 100),
  brick(300, 100)
}
```

And now the top of our main file only needs to load the entities file and it will have the list to use in `love.draw` and elsewhere as needed:

```
-- main.lua

local entities = require('entities')
local world = require('world')
```

When you run the game, you should be seeing something similar to this:



If you missed anything or are having issues, here's a copy of the completed source code for this section:
https://github.com/RVAGameJams/learn2love/tree/master/code/breakout-2

And that's about it for entity management. We'll figure out how to handle keypresses for the paddle and everything else in the next section. We'll finish the cleanup in our main file while we're at it.

Exercises

- Now that our entities have passed off knowledge on where they spawn over to **entities.lua**, our left and right boundaries are identical files. Replace **boundary-left.lua** and **boundary-right.lua** with a single **boundary-vertical.lua** file and spawn two copies of that in **entities.lua**. If you get stuck, check out the **entities.lua** file in the next section for how this is done.

Breakout (part 3): inputs

Review

In the previous section we reconstructed our entities to make room for bricks and additional functionality. We haven't completed any new items on our checklist:

- The objective of the game is to destroy all the bricks on the screen
- The player controls a "paddle" entity that hits a ball
- The ball destroys the bricks
- ✓ The ball needs to stay within the boundaries of the screen
- If the ball touches the bottom of the screen, the game ends

So let's come up with a system to handle user input and get the paddle moving.

Input service

Inside **main.lua** there is some functionality for this that we are going to remove and rewrite starting with a new file that specifically handles all the user input. This kind of file is typically called a service because it abstracts away tedious functionality into an easy-to-use service. I encourage you to write out the service instead of copying and pasting. Read through each function and try to understand what each one does.

```
-- input.lua

-- This table is the service and will contain some functions
-- that can be accessed from entities or the main.lua.
local input = {}
-- Map specific user inputs to game actions
local press_functions = {}
local release_functions = {}


-- For moving paddle left
input.left = false
-- For moving paddle right
input.right = false
-- Keep track of whether game is pause
input.paused = false
-- Look up in the map for actions that correspond to specific key presses
input.press = function(pressed_key)
  if press_functions[pressed_key] then
    press_functions[pressed_key]()
  end
end
-- Look up in the map for actions that correspond to specific key releases
input.release = function(released_key)
  if release_functions[released_key] then
    release_functions[released_key]()
  end
end
-- Handle window focusing/unfocusing
input.toggle_focus = function(focused)
  if not focused then
    input.paused = true
  end
end


press_functions.left = function()
  input.left = true
end
press_functions.right = function()
  input.right = true
end
press_functions.escape = function()
  love.event.quit()
end
press_functions.space = function()
  input.paused = not input.paused
end


release_functions.left = function()
  input.left = false
end
release_functions.right = function()
  input.right = false
end


return input
```

The input table is what gets returned, meaning when we `require('input')` in another file, we get back that table and its contents. Inside the input there are three boolean properties that get toggled by user input: `input.left`,

`input.right`, and `input.paused`. Along with these three properties, there are three functions exposed to us to make use of: `input.press`, `input.release`, and `input.toggle_focus`, all of which we will invoke from our callbacks in **main.lua**:

```
-- main.lua

local entities = require('entities')
local input = require('input')
local world = require('world')

love.draw = function()
  for _, entity in ipairs(entities) do
    if entity.draw then entity:draw() end
  end
end

love.focus = function(focused)
  input.toggle_focus(focused)
end

love.keypressed = function(pressed_key)
  input.press(pressed_key)
end

love.keyreleased = function(released_key)
  input.release(released_key)
end

love.update = function(dt)
  if not input.paused then
    for _, entity in ipairs(entities) do
      if entity.update then entity:update(dt) end
    end
    world:update(dt)
  end
end
```

In `love.update` we skip updates if `input.paused` is `true`. However if the game is **not** paused then it will loop through the entity list, calling `entity.update` if the entity has an update function. With this added functionality, we can append an `entity.update` function into our existing paddle code:

```
-- entities/paddle.lua

entity.update = function(self)
  -- Don't move if both keys are pressed. Just return
  -- instead of going through the rest of the function.
  if input.left and input.right then
    return
  end
  local self_x, self_y = self.body:getPosition()
  if input.left then
    self.body:setPosition(self_x - 10, self_y)
  elseif input.right then
    self.body:setPosition(self_x + 10, self_y)
  end
end
```

The left and right arrows will now move the paddle! There isn't much else to say here in the way of input. A bit unrelated to the actual input, but more so the paddle functionality is it moves off screen and doesn't adhere to the boundaries? Why is that?

If you remember when we created the paddle, it is a static entity. It doesn't have the ability to move on its own or by the effect of other entities. This will cause us some problems later (and we're learning the hard way)! Rather than forcing the paddle with an invisible push, we force a new position for the paddle when we call `body:setPosition` inside the paddle's `entity.update` function. It's like we're teleporting it on top of whatever space we want with a keystroke, ignoring all physics and collision. This is simpler to code and gets around the fact the paddle's static body won't respond to force. To fix this, we can artificially set the boundary on the paddle by checking if it is out of bounds before moving it.

```
-- entities/paddle.lua

entity.update = function(self)
  -- Don't move if both keys are pressed. Just return
  -- instead of going through the rest of the function.
  if input.left and input.right then
    return
  end
  local self_x, self_y = self.body:getPosition()
  if input.left then
    local new_x = math.max(self_x - 10, 108)
    self.body:setPosition(new_x, self_y)
  elseif input.right then
    local new_x = math.min(self_x + 10, 700)
    self.body:setPosition(new_x, self_y)
  end
end
```

Calling `math.max` means we will set the new x-position to either `self_x - 10` or `100`, whichever number is bigger. This prevents us from getting a number so small it runs off too far to the left. `math.min` does the opposite and takes care of the right side of the screen.

One issue you may or may not notice is movement isn't always a uniform speed, and depending on the speed of your computer the paddle may appear to go faster or slower. Remember the article on delta time? We need to scale the distance travelled to match the amount of time that has passed. Conveniently, we are getting the delta time from `love.update` already. Take a closer look at it:

```
-- main.lua

love.update = function(dt)
  if not input.paused then
    for _, entity in ipairs(entities) do
      -- Delta time is being passed
      -- to the entity.update function here
      --                                  |
      --                                  |
      --                                  V
      if entity.update then entity:update(dt) end
    end
    world:update(dt)
  end
end
```

Which means we can do this:

```
-- entities/paddle.lua

entity.update = function(self, dt)
  -- Don't move if both keys are pressed. Just return
  -- instead of going through the rest of the function.
  if input.left and input.right then
    return
  end
  local self_x, self_y = self.body:getPosition()
  if input.left then
    local new_x = math.max(self_x - (400 * dt), 100)
    self.body:setPosition(new_x, self_y)
  elseif input.right then
    local new_x = math.min(self_x + (400 * dt), 700)
    self.body:setPosition(new_x, self_y)
  end
end
```

The number `400` is arbitrary and can be whatever speed you want the paddle to move at. `dt` is a small number so it needs to be multiplied by a large number like 400 to match a speed similar to what we were seeing before when we simply were adding and subtracting `10`.

If you missed anything or are having issues, here's a copy of the completed source code for this section:
https://github.com/RVAGameJams/learn2love/tree/master/code/breakout-3

In the next section we will work on the physics more to give the ball movement a more realistic feel. We will also implement the ability to destroy bricks using the world collision callbacks.

Exercises

- Despite having a restitution of 1, the ball is losing momentum as it collides with other objects. This is due to friction. How can that be fixed?
- When the game is paused, make it display text on the screen so the player knows the game isn't just frozen. Hint: you'll need one of the draw functions from love.graphics to print the text.

The answers to these exercises will be in the next section's source code.

Breakout (part 4): physics

Review

In the previous exercises we discussed issues with the ball slowing down due to friction. With a bit of browsing through the `love.physics` documentation you might have seen that friction is a property of the fixture and can be set to 0 in `fixture:setFriction`.

How about creating the pause screen text? Were you able to do it without touching **main.lua**? Take a look at this entity that was created just for the single responsibility of displaying pause text:

```
-- entities/pause-text.lua

local input = require('input')

return function()
  local window_width, window_height = love.window.getMode()

  local entity = {}

  entity.draw = function(self)
    if input.paused then
      love.graphics.print(
        {{ 0.2, 1, 0.2, 1 }, 'PAUSED'},
        math.floor(window_width / 2) - 54,
        math.floor(window_height / 2),
        0,
        2,
        2
      )
    end
  end

  return entity
end
```

That's right. Even the pause screen is an entity. The first natural place to think to put it would be the main file but entity files are perfect because we can create as many as we need for each task and add it to **entities.lua** where it will be handled by the game loop. For centering the text the `love.window.getMode` function is used to get the full window dimensions then those numbers are divided in half. This saves us from manually coding in numbers that would need to be readjusted if the window size changed. Additionally, `math.floor` was used for good measure to make sure we are returning a whole number. It is recommended to round decimals off from numbers when passing coordinates to the drawing functions. Otherwise it may attempt to draw that object between pixels on the screen and cause some blurriness.

Physics updates

An issue we had with the game physics since we got the paddle moving is that the ball doesn't always ricochet off the paddle as you would expect. This is because we made the paddle static so the ball doesn't push it around, but this has the effect of the paddle not interacting with the ball correctly. This is where `kinematic` bodies come in. Kinematic bodies, like static bodies aren't affected by dynamic bodies. Kinematic bodies, unlike static bodies, can affect dynamic bodies.

We're going to make 3 changes to **paddle.lua**:

- Move the boundary dimensions, paddle dimensions, and paddle speed to easily-referenced variables at the top of the file.
- Change the body type to kinematic
- Overhaul the update code to move the body with linear velocity rather than manually setting a new location on the screen with every update

```
-- entities/paddle.lua

local input = require('input')
local world = require('world')

return function(pos_x, pos_y)
  local window_width = love.window.getMode()
  -- Variables to make these easier to adjust
  local entity_width = 120
  local entity_height = 20
  local entity_speed = 600
    -- The limit of how far left/right the entity can move towards
    -- the edges (with a little bit of padding thrown on).
  local left_boundary = (entity_width / 2) + 2
  local right_boundary = window_width - (entity_width / 2) - 2

  local entity = {}
  entity.body = love.physics.newBody(world, pos_x, pos_y, 'kinematic')
  entity.shape = love.physics.newRectangleShape(entity_width, entity_height)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  entity.draw = function(self)
    love.graphics.polygon('line', self.body:getWorldPoints(self.shape:getPoints
  end

  entity.update = function(self)
    -- Don't move if both keys are pressed. Just return
    -- instead of going through the rest of the function.
    if input.left and input.right then
      return
    end
    local self_x = self.body:getX()
    if input.left and self_x > left_boundary then
      self.body:setLinearVelocity(-entity_speed, 0)
    elseif input.right and self_x < right_boundary then
      self.body:setLinearVelocity(entity_speed, 0)
    else
      self.body:setLinearVelocity(0, 0)
    end
  end

  return entity
end
```

I took the liberty of adjusting the paddle size, but with our nice boundary-size calculations in place the paddle dimensions can easily be adjusted and the boundary size will take those changes into account. Let's drill into the `entity.update` function.

Once the inputs are checked to be true or false the current x-position of the paddle is checked to see if it goes out of the boundaries (calculated near the top). Notice that the calculations for the boundary locations are done at the top instead of in `entity.update`. This means those calculations aren't done on every update since they don't need to be.

A bit more complex than the paddle are the calculations for the ball:

```
-- entities/ball.lua

local world = require('world')

return function(x_pos, y_pos)
  local entity_max_speed = 880

  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'dynamic')
  entity.body:setLinearVelocity(300, 300)
  entity.shape = love.physics.newCircleShape(0, 0, 10)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setFriction(0)
  entity.fixture:setRestitution(1)
  entity.fixture:setUserData(entity)

  entity.draw = function(self)
    local self_x, self_y = self.body:getWorldCenter()
    love.graphics.circle('fill', self_x, self_y, self.shape:getRadius())
  end

  entity.update = function(self)
    local vel_x, vel_y = self.body:getLinearVelocity()
    local speed = math.abs(vel_x) + math.abs(vel_y)

    local vel_x_is_critical = math.abs(vel_x) > entity_max_speed * 2
    local vel_y_is_critical = math.abs(vel_y) > entity_max_speed * 2
    -- Ball is bouncing too fast to reasonably hit.
    -- Cut down its speed by 75% if so.
    if vel_x_is_critical or vel_y_is_critical then
      self.body:setLinearVelocity(vel_x * .75, vel_y * .75)
    end
    if speed > entity_max_speed then
      self.body:setLinearDamping(0.1)
    else
      self.body:setLinearDamping(0)
    end
  end

  return entity
end
```

In the first chunk we get the current x and y velocity, which tells us the x and y direction of the ball:

```
    local vel_x, vel_y = self.body:getLinearVelocity()
    local speed = math.abs(vel_x) + math.abs(vel_y)
```

An example `vel_x` / `vel_y` may be `212` / `-300`, which means the ball is moving up and towards the right. The speed is calculated by turning both these numbers into absolute numbers and adding them together (so `512` in the example).

In the next chunk there is a safety check to make sure the ball didn't ricochet with so much force that it's going too fast to possibly hit. If either boolean variable is true then the linear velocity is multiplied by a fraction of itself to quickly slow it down:

```
local vel_x_is_critical = math.abs(vel_x) > entity_max_speed * 2
local vel_y_is_critical = math.abs(vel_y) > entity_max_speed * 2
-- Ball is bouncing too fast to reasonably hit.
-- Cut down its speed by 75% if so.
if vel_x_is_critical or vel_y_is_critical then
  self.body:setLinearVelocity(vel_x * .75, vel_y * .75)
end
```

Now there is just a check to ease the ball back down to a comfortable maximum speed. If the ball's speed is greater than `entity_max_speed` a damping is applied which will reduce the balls speed below 880. Once the target speed is reached then the damping switches back to 0:

```
if speed > entity_max_speed then
  self.body:setLinearDamping(0.1)
else
  self.body:setLinearDamping(0)
end
```

Try out the changes to feel it in action compared to the previous physics and hopefully you will find that it's an improvement. It's not a perfect replica of the arcade game, but playing around with these tricks and features you can get it pretty darn close to something satisfactory. Another thing to try out if within the ball's `entity.update`, add a line under the speed variable that reads `print(speed)` and watch the number increase and decrease again as the damping kicks in. Pretty neat that most of the heavy calculations are handled by the physics engine for us.

Collision

There are 4 changes involved to make the bricks destructible:

- Update **world.lua** to check for collision functionality for the entities when they collide
- Update **brick.lua** to include a collision callback
- Add a new attribute on the brick entity to let us know its current condition and if it needs to be destroyed. We'll just call it `entity.health`.
- Update **main.lua** to remove/destroy any entities that have no more health

First the world:

```
-- world.lua

-- Called at the end of any contact in the world. Parameters:
-- {fixture} fixture_a - first fixture object in the collision.
-- {fixture} fixture_b - second fixture object in the collision.
-- {contact} contact - world object created on and at the point of contact
--   See further: https://love2d.org/wiki/Contact
local end_contact_callback = function(fixture_a, fixture_b, contact)
  local entity_a = fixture_a:getUserData()
  local entity_b = fixture_b:getUserData()
  if entity_a.end_contact then entity_a:end_contact() end
  if entity_b.end_contact then entity_b:end_contact() end
end

local world = love.physics.newWorld(0, 0)

world:setCallbacks(nil, end_contact_callback, nil, nil)

return world
```

The only callback we'll be using for this tutorial is the end-contact callback, so for `world:setCallbacks` we are going to returning `nil` for the rest to keep our code fast and clean. Take a look at what is happening inside `end_contact_callback`. Remember inside each entity when we invoked `entity.fixture:setUserData(entity)` ? With the entity attached to each fixture, we can get access to those entities by invoking `fixture:getUserData` in the callback above. Once we have access to each entity, we check to see if the entity has any `end_contact` functions, code specific to that entity that needs to run when ending the collision.

Now we can go to **brick.lua** and define that functionality:

```
-- entities/brick.lua

local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'static')
  entity.shape = love.physics.newRectangleShape(50, 20)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  -- How many times the brick can be hit before it is destroyed
  entity.health = 2

  entity.draw = function(self)
    love.graphics.polygon('fill', self.body:getWorldPoints(self.shape:getPoints
  end

  entity.end_contact = function(self)
    self.health = self.health - 1
  end

  return entity
end
```

Notice the two new values in the table, `entity.health` and `entity.end_contact`. Inside `end_contact` we are subtracting 1 health when the collision ends. Health could start at any number and that means the ball will need to collide with the brick that many times before the health reaches 0. Lastly, we need to go into **main.lua** and adjust `love.update` so it does something when it sees an entity with 0 health:

```
-- main.lua
love.update = function(dt)
  if not input.paused then
    local index = 1
    while index <= #entities do
      local entity = entities[index]
      if entity.update then entity:update(dt) end
      -- When an entity has no health (brick has been hit enough times
      -- then we remove it from the list of entities. Don't increment
      -- the index number if doing that though because we have shrunk
      -- the table and made all the items shift down by 1 in the index.
      if entity.health == 0 then
        table.remove(entities, index)
        entity.fixture:destroy()
      else
        index = index + 1
      end
    end
    world:update(dt)
  end
end
```

The entity is removed from `entities` as well as having its fixture destroyed from the world. This will only happen to bricks with 0 health. It won't happen to entities where we didn't define health because `nil` is not the same thing as `0`. Notice that a `while` loop was used here. This is because we may remove entities from the list we are looping over and this would throw off the index count for a regular `for` loop.

If you missed anything or are having issues, here's a copy of the completed source code for this section:
https://github.com/RVAGameJams/learn2love/tree/master/code/breakout-4

In the next section we'll review the checklist and see what is left to cover.

Exercises

- It would be great if the colors of the bricks changed depending how much health the brick has. Update the brick's `entity.draw` function with some colors. Hint: we covered colors in 2.4 - Game loop.
- Add more bricks to the screen. What's the easiest way to do that?

Breakout (part 5): game state

Review

We've gotten a bit done so let's look at the basic requirements again:

- The objective of the game is to destroy all the bricks on the screen
- ✓ The player controls a "paddle" entity that hits a ball
- ✓ The ball destroys the bricks
- ✓ The ball needs to stay within the boundaries of the screen
- If the ball touches the bottom of the screen, the game ends

In the previous exercise the question was brought up what would be the easiest way to draw a bunch of bricks across the screen. A simple, but *very tedious* answer to that would be to position the bricks one at a time in **entities.lua** like so:

```
  brick(40, 80),
  brick(100, 140)
  -- and so on...
```

If you want to make your bricks into a shape or sculpture then that might be the best approach. If you just want to arrange your bricks into a grid, then the easiest way would be to write a numeric for-loop.

```
-- entities.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_vertical = require('entities/boundary-vertical')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local pause_text = require('entities/pause-text')
local ball = require('entities/ball')
local brick = require('entities/brick')

local entities = {
  boundary_bottom(400, 606),
  boundary_vertical(-6, 300),
  boundary_vertical(806, 300),
  boundary_top(400, -6),
  paddle(300, 500),
  pause_text(),
  ball(200, 200)
}

local row_width = love.window.getMode() - 20
for number = 0, 38 do
  local brick_x = ((number * 60) % row_width) + 40
  local brick_y = (math.floor((number * 60) / row_width) * 40) + 80
  entities[#entities + 1] = brick(brick_x, brick_y)
end

return entities
```

Ok this admittedly looks more complicated at first, but if you remember the arithmetic and orders of operation covered in 1.1 - Interactive coding statements are processed from the inner parenthesis and worked outwards. So why the long calculation? Let's start off with a simpler calculation:

```
local brick_x = number * 60
```

Starting with the `number` 0 up to 38, there will be 39 loops and therefore 39 bricks drawn. On the first loop, `number` is 0. Since the bricks are 50 pixels wide this would draw the bricks with a 10 pixel space between each. First brick at 60, then 120, then 180... Ok, but then after only a dozen bricks we would start running off the screen. This is where the modulus comes in handy:

```
local brick_x = (number * 60) % row_width
```

`row_width` is how wide we want a row of bricks to be be. In this case `row_width` is the screen width, 800 pixels, subtract 20 pixels for padding. So draw the bricks every 60 pixels, but then when you get to 780 pixels, start back at 0 pixels and begin drawing a new row. Thanks modulus! Now just to give the bricks some spacing on the left side away from the wall, we can go ahead and add 40 pixels to the final result for the x-position:

```
local brick_x = ((number * 60) % row_width) + 40
```

The brick's y-position is calculated a little bit differently. What we need to find out is which row we're on so we know where on the y-axis to draw. If we take the `number` and multiply it by 60 then do a modulus we know that gives us the x-position. So let's take that chunk of code from above and make that the basis of our y-position calculation:

```
local brick_y = (number * 60) % row_width
```

Rather than using modulus, if we use regular division we get a small remainder every time `(number * 60)` exceeds the row width:

```
local brick_y = (number * 60) / row_width
```

This will give us a number with decimals so to keep things rounded we can use `math.floor` to snap the y-position down to the nearest whole number:

```
local brick_y = math.floor((number * 60) / row_width)
```

Great! Now every time the x-position exceeds the row width, we get back the number of the row we're on... 0 for the first, 1 for the second, 2 and so on. With this number we can now space out each row by 40 pixels:

```
local brick_y = math.floor((number * 60) / row_width) * 40
```

Then finally just to shift the bricks a little further down the screen we give it a padding that looks right, say 80:

```
local brick_y = (math.floor((number * 60) / row_width) * 40) + 80
```

And there you go. The entity can just be added to the end of the entities list so it doesn't get lost:

```
entities[#entities + 1] = brick(brick_x, brick_y)
```

In the previous exercises we also talked about drawing the bricks different colors to indicate their integrity/health left before they will be destroyed. Rather than review that now, let's dive into state management and we'll wrap coloring up along the way.

State management

Your average, every-day program has a lot of information it needs to story in memory. For our game to function with just the basic features, we need to store information about each entity, whether or not the game is currently paused, or if the game is won or lost. This information is called the *state*. The state is data that may change during the lifetime of the application. Think of the state of your lights in your room. Are they currently in an "on" or "off" state? The state can cause different effects on the application, like if the "pause" state of the game is "true" then the world will no longer receive updates.

One thing we must think of is how to organize the state of our application. This is something we take for granted often in the real world; We don't have to figure out where to store the state of our lights. It's a piece of information intrinsic to the lamp's design.

So why do we have to care so much about our game's state? To be fair, our game is small so we probably don't need to. However, it is crucial to reconcile such things while applications are small because it will be very difficult to go back and fix a bunch of code once the application is big. The way you should organize the state of your application should accomplish a few things:

- **It should be easy to find and access the necessary data that makes up the state.** For instance, how easy is it for our main file to access the entities and loop over them in the `love.update` function?
- **There should only be one copy of the state.** If we want to access the "paused" state of our game in multiple places that is fine, but we shouldn't have multiple "paused" variables floating around our game. If we had a "paused" variable inside an entity file and another inside the input service updating independently then they could get out of sync and our game would get confused on when it should be paused.
- **The state should only be accessed where it is needed.** If you were accessing or storing the "paused" state inside the ball entity, then if that ball

was destroyed then something bad will happen the next time the game checks to see if it is paused.

What files contain the state of our game?

- **entities.lua** - Each entity table is responsible for its own state. For instance, each brick stores the state of its own health. All the entities tables are generated and stored here. The entities are **not** stored in the entities folder. Those are just functions used to generate the entities. The blueprints.
- **input.lua** - This file is responsible for capturing user input, but also storing the state of what keys are currently being pressed.
- **world.lua** - This file is not only the blueprints for the game world, but it also stores the world instance that is generated when the game starts. We made the world instance easily accessible to the rest of the application by writing `return world` at the end. There would be no game if this wasn't easily accessible.

A few pieces of game state we need to add are a boolean of whether the game is over, another for if the stage is cleared, and also a list of colors to use in our game which we'll refer to as our *palette*. This information wouldn't really fit in any of the places we listed above, and we don't want to add it to **main.lua** because of our first rule that the game state should be easy to access where it is needed. Besides, that's not the main file's responsibility. We'll go ahead and just make a new file called **state.lua** and store the overall game state in this file. This is also a little matter of opinion but the "paused" and button states we'll also move in here since they affect the overall game's state. This will also make it so that **input.lua**'s only responsibility is to capture and translate the user input, **not** to handle any state whatsoever.

```
-- state.lua

-- The state of the game. This way our data is separate from our functionality

return {
  button_left = false,
  button_right = false,
  game_over = false,
  palette = {
    {1.0, 0.0, 0.0, 1.0},  -- red
    {0.0, 1.0, 0.0, 1.0},  -- green
    {0.4, 0.4, 1.0, 1.0},  -- blue
    {0.9, 1.0, 0.2, 1.0},  -- yellow
    {1.0, 1.0, 1.0, 1.0}   -- white
  },
  paused = false,
  stage_cleared = false
}
```

It's kind of a nice feeling to keep all the state together. We could even move the entities list into **state.lua** and get rid of **entities.lua**, but this doesn't seem necessary. Now with this shift in data we need to update **input.lua** and **main.lua** to reference the new file:

```
-- input.lua

local state = require('state')

-- Map specific user inputs to game states
local press_functions = {
  left = function()
    state.button_left = true
  end,
  right = function()
    state.button_right = true
  end,
  escape = function()
    love.event.quit()
  end,
  space = function()
    if state.game_over or state.stage_cleared then
      return
    end
    state.paused = not state.paused
  end
}

local release_functions = {
  left = function()
    state.button_left = false
  end,
  right = function()
    state.button_right = false
  end
}


-- This table is the service and will contain some functions
-- that can be accessed from entities or the main.lua.
return {
  -- Look up in the map for actions that correspond to specific key presses
  press = function(pressed_key)
    if press_functions[pressed_key] then
      press_functions[pressed_key]()
    end
  end,
  -- Look up in the map for actions that correspond to specific key releases
  release = function(released_key)
    if release_functions[released_key] then
      release_functions[released_key]()
    end
  end,
  -- Handle window focusing/unfocusing
  toggle_focus = function(focused)
    if not focused then
      state.paused = true
    end
  end
}
```

```
-- main.lua

local entities = require('entities')
local input = require('input')
local state = require('state')
local world = require('world')

love.draw = function()
  for _, entity in ipairs(entities) do
    if entity.draw then entity:draw() end
  end
end

love.focus = function(focused)
  input.toggle_focus(focused)
end

love.keypressed = function(pressed_key)
  input.press(pressed_key)
end

love.keyreleased = function(released_key)
  input.release(released_key)
end

love.update = function(dt)
  if state.game_over or state.paused or state.stage_cleared then
    return
  end

  local index = 1
  while index <= #entities do
    local entity = entities[index]
    if entity.update then entity:update(dt) end
    -- When an entity has no health (brick has been hit enough times
    -- then we remove it from the list of entities. Don't increment
    -- the index number if doing that though because we have shrunk
    -- the table and made all the items shift down by 1 in the index.
    if entity.health and entity.health < 1 then
      table.remove(entities, index)
      entity.fixture:destroy()
    else
      index = index + 1
    end
  end

  world:update(dt)
end
```

Notice the change to `love.update`. We check if `state.game_over`, `state.paused` or `state.stage_cleared` is true and if so, we return from `love.update` without doing any of the updates as these kind of game states merit freezing the screen.

Next up, update **paddle.lua** to require `state` instead of `input`. The `entity.update` function now needs to reference `state.button_left` and `state.button_right` to tell if the player has pressed any buttons. Try updating it on your own. If you do get stuck, the source code will be in the link at the bottom waiting for you.

Ok, now that we have a state where we stored the colors it is probably a good time to try and update **brick.lua**. First let's look at those colors stored in **state.lua**:

```
  palette = {
    {1.0, 0.0, 0.0, 1.0},  -- red
    {0.0, 1.0, 0.0, 1.0},  -- green
    {0.4, 0.4, 1.0, 1.0},  -- blue
    {0.9, 1.0, 0.2, 1.0},  -- yellow
    {1.0, 1.0, 1.0, 1.0}   -- white
  },
```

The `palette` table is a list of more tables. Each table in the list represents colors where the first number is the amount of red, 2nd the amount of green, 3rd the amount of blue, and 4th number the amount of opacity. Setting the last number to `0` means the color is 100% transparent and `1` means it is completely opaque. All of these values mix together to form a single color. In the case of the first color, we have the red value set to maximum opaque red with no other colors mixed in. I would encourage you to go back and edit the colors in this palette after everything is working. Now, inside **brick.lua** let's update `entity.draw`:

```
-- entities/brick.lua

local state = require('state')
local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'static')
  entity.shape = love.physics.newRectangleShape(50, 20)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  -- How many times the brick can be hit before it is destroyed
  entity.health = 2
  -- Used to check during update if this entity is a brick
  -- If no bricks are found then the level was cleared
  entity.type = 'brick'

  entity.draw = function(self)
    -- Draw the brick in a different color depending on health
    love.graphics.setColor(state.palette[self.health] or state.palette[5])
    love.graphics.polygon('fill', self.body:getWorldPoints(self.shape:getPoints
    -- Reset graphics drawer back to the default color (white)

    love.graphics.setColor(state.palette[5])
  end

  entity.end_contact = function(self)
    self.health = self.health - 1
  end

  return entity
end
```

Before drawing the brick's polygon, we set the graphics renderer to use one of the colors from `state.palette`. The color to use depends on what the brick's health is. So if the brick has 2 health then `state.palette[self.health]` will become `state.palette[2]` which will grab the 2nd color in the list... green. If the brick's health was 1, then the first color from the palette would be selected... red. After

the colored polygon is drawn, `entity.draw` finishes up by setting the renderer color back to white. If we didn't do this step, the ball and paddle would get drawn the same color as the bricks.

One last thing we need to do to get the game working is update **pause-text.lua** as it is incorrectly looking for the "pause" state in **input.lua** instead of the new **state.lua** location:

```
-- entities/pause-text.lua

local state = require('state')

return function()
  local window_width, window_height = love.window.getMode()

  local entity = {}

  entity.draw = function(self)
    if state.paused then
      love.graphics.print(
        {state.palette[3], 'PAUSED'},
        math.floor(window_width / 2) - 54,
        math.floor(window_height / 2),
        0,
        2,
        2
      )
    end
  end

  return entity
end
```

Final touches

We need the game to end when the player destroys all the bricks or loses the ball. Just like the pause-text entity, display some messages based on the game state.

```
-- entities/game-over-text.lua

local state = require('state')

return function()
  local window_width, window_height = love.window.getMode()

  local entity = {}

  entity.draw = function(self)
    if state.game_over then
      love.graphics.print(
        {state.palette[5], 'GAME OVER'},
        math.floor(window_width / 2) - 100,
        math.floor(window_height / 2),
        0,
        2,
        2
      )
    end
  end

  return entity
end
```

```
-- entities/stage-clear-text.lua

local state = require('state')

return function()
  local window_width, window_height = love.window.getMode()

  local entity = {}

  entity.draw = function(self)
    if state.stage_cleared then
      love.graphics.print(
        {state.palette[4], 'STAGE CLEARED'},
        math.floor(window_width / 2) - 110,
        math.floor(window_height / 2),
        0,
        2,
        2
      )
    end
  end

  return entity
end
```

To trigger the "GAME OVER" text is easy enough. We need to add a collision callback to **boundary-bottom.lua** to set the game's `state.game_over` to true on any collision:

```
-- entities/boundary-bottom.lua

local state = require('state')
local world = require('world')

return function(x_pos, y_pos)
  local entity = {}
  entity.body = love.physics.newBody(world, x_pos, y_pos, 'static')
  entity.shape = love.physics.newRectangleShape(800, 10)
  entity.fixture = love.physics.newFixture(entity.body, entity.shape)
  entity.fixture:setUserData(entity)

  entity.end_contact = function(self)
    state.game_over = true
  end

  return entity
end
```

Don't forget we need to update **entities.lua** to add our two new entities:

```
-- entities.lua

local boundary_bottom = require('entities/boundary-bottom')
local boundary_vertical = require('entities/boundary-vertical')
local boundary_top = require('entities/boundary-top')
local paddle = require('entities/paddle')
local game_over_text = require('entities/game-over-text')
local pause_text = require('entities/pause-text')
local stage_clear_text = require('entities/stage-clear-text')
local ball = require('entities/ball')
local brick = require('entities/brick')

local entities = {
  boundary_bottom(400, 606),
  boundary_vertical(-6, 300),
  boundary_vertical(806, 300),
  boundary_top(400, -6),
  paddle(300, 500),
  game_over_text(),
  pause_text(),
  stage_clear_text(),
  ball(200, 200)
}

local row_width = love.window.getMode() - 20
for number = 0, 38 do
  local brick_x = ((number * 60) % row_width) + 40
  local brick_y = (math.floor((number * 60) / row_width) * 40) + 80
  entities[#entities + 1] = brick(brick_x, brick_y)
end

return entities
```

Ok, test that out and check that the "GAME OVER" text works. If it does, then let's continue on and add the conditions for how to win the game. This involves checking through all the entities in `love.update` to make sure we still have bricks. If we don't have any bricks left, then the player destroyed them all and the stage is cleared.

```
-- main.lua
love.update = function(dt)
  if state.game_over or state.paused or state.stage_cleared then
    return
  end

  -- Switch to true if we have bricks left
  local have_bricks = false

  local index = 1
  while index <= #entities do
    local entity = entities[index]
    if entity.type == 'brick' then have_bricks = true end
    if entity.update then entity:update(dt) end
    -- When an entity has no health (brick has been hit enough times
    -- then we remove it from the list of entities. Don't increment
    -- the index number if doing that though because we have shrunk
    -- the table and made all the items shift down by 1 in the index.
    if entity.health and entity.health < 1 then
      table.remove(entities, index)
      entity.fixture:destroy()
    else
      index = index + 1
    end
  end

  -- Flag the stage cleared if there are no more bricks
  state.stage_cleared = not have_bricks
  world:update(dt)
end
```

Every time `love.update` is ran, we set a variable `have_bricks` to false. If this boolean stays `false` all the way to the bottom of the function then `state.stage_cleared` gets switched to true and the game is won. Inside the `while` loop, however, we check every entity to see if we find an `entity.type` of `'bricks'` and if so, `have_bricks` gets flipped to `true` to stop the game from being won yet.

So that about does it for completing our checklist. The game may not be as feature-complete as a true breakout game, but that room for improvement leaves opportunity for you to modify the game to work how you want it to. It's really up to your imagination. Try out a few exercises if you can't think up any new features. If you are having trouble running the game, be sure to check out the source code:

https://github.com/RVAGameJams/learn2love/tree/master/code/breakout-5

Exercises

- Instead of getting a game over as soon as the ball touches the ground once, add a new property in **state.lua** named `lives` and set it to as many lives as you want the player to have. Make is so the `state.lives` decreases when the ball hits the ground and make the `game_over` not trigger unless `state.lives < 1`.
- Try setting the paddle to different shape to make the game play differently
- Come up with new features to make the game play better and feel more polished

- Change the ball and paddle colors
- Add a background color
- Figure out how to play a sound effect when the ball collides with things
- Create some kind of power-up entity

Binary and bitmasks

In 2.10 - Collision callbacks we saw how to react to entities colliding. In this section we're going to discuss how we can better control when collisions happen. As the title suggests, this will involve understanding some binary.

Let's say we have a beat-em-up game where two players are fighting bad guys and we don't want players to collide with each other and instead only collide with enemies. The collision callback could look something like this:

```
local begin_contact_callback = function(fixture_a, fixture_b)
  local entity_a_type = fixture_a:getUserData()
  local entity_b_type = fixture_b:getUserData()
  -- Check these aren't the same type of entity
  if entity_a_type ~= entity_b_type then
    -- Some code to handle the collision goes here...

  end
end
```

But what if you had power-ups and you want players to collide with the power-ups but you don't want enemies touching the power-ups? Things can get complicated pretty quickly:

```
local begin_contact_callback = function(fixture_a, fixture_b)
  local a = fixture_a:getUserData()
  local b = fixture_b:getUserData()
  if (a == 'powerup' and b == 'player') or (a == 'player' and b == 'powerup') t
    -- Some power-up code...

  -- Don't let power-ups collide with other entity types like bad guys
  elseif a ~= b and a ~= 'powerup' and b~= 'powerup' then
    -- Code to handle the rest of the collisions...

  end
end
```

Let's find a better way!

Binary operations

Back in 1.0 - Programming basics we discussed *operations*– how to operate on strings with equality (`==`) checks, how to operate on numbers with arithmetic operations, and even how to perform boolean operations like `and` and `or`. Binary numbers have their own operations, often called bitwise operations. To perform binary operations, let's first look at how to represent binary numbers. Typing a number like `101`, Lua will interpret it as a decimal number (literally one-hundred one) so we need to represent it as a string and convert it to a number. To convert a string to a number, you pass in the number and the base (base-2 in this case) like so:

```
print(tonumber('101', 2))
```

Which converts the binary number `101` to decimal when it prints out:

```
5
```

For counting in binary and learning how to read and convert between binary and decimal, there are many resources that already explain it in much better. Learning how to do the conversions isn't necessary to learning these basic binary operations, but is an essential skill to have in the field of computer science.

Moving on, let's take a look at some of the basic operations.

AND

Binary numbers are similar to booleans in that binary only has 1's and 0's. The AND operator also works similarly to the boolean `and`. You give it two digits and both must be `1` (`true`) for the output to be `1`.

Unfortunately at the time of writing this the online REPL has an outdated version of Lua that doesn't support binary operations. No worries, let's create a **main.lua** file and try it out using LÖVE. To perform binary operations, the included `'bit'` library must be loaded. When required, it will return a table with many functions in it related to binary operations. The first function we'll try, `bit.band()` performs a **b**inary **AND** operation.

```
local bit = require('bit')

print(bit.band(0, 0))
print(bit.band(0, 1))
print(bit.band(1, 0))
print(bit.band(1, 1))
```

This will print to the debug console:

```
0
0
0
1
```

You can pass it the decimals `1` and `0` as those numbers are the same in binary and decimal.

The operation is not limited to two inputs:

```
print(bit.band(1, 0, 1))
```

You can also pass it multi-digit numbers:

```
print(bit.band(
  tonumber('111', 2),
  tonumber('101', 2)
))
```

Note that you need to always use `tonumber()` to convert your binary string to a number as the function always expects a decimal number. Likewise the output will always be a decimal number:

```
5
```

Lay it out like an arithmetic table and you can solve it just as easily:

```
  111
  101
  ---
  101 --> 5


  11011010
  10111100
  --------
  10011000 --> 152
```

OR

Like with the boolean `or`, the binary OR output will be `1` if either the first **or** the second number is 1. You could say it is the least picky operator in that it doesn't care as long as it gets a 1 somewhere at least once.

```
-- main.lua
local bit = require('bit')

print(bit.bor(1, 1))
print(bit.bor(1, 0))
print(bit.bor(0, 0))
print(bit.bor(0, 0, 0, 1, 0))
```

```
1
1
0
1
```

XOR

Xor (e**x**clusive **or**), returns 1 only when it gets one 1. Let's compare XOR in a table to the others:

AND

input A	input B	output
0	0	0
0	1	0
1	0	0
1	1	1

OR

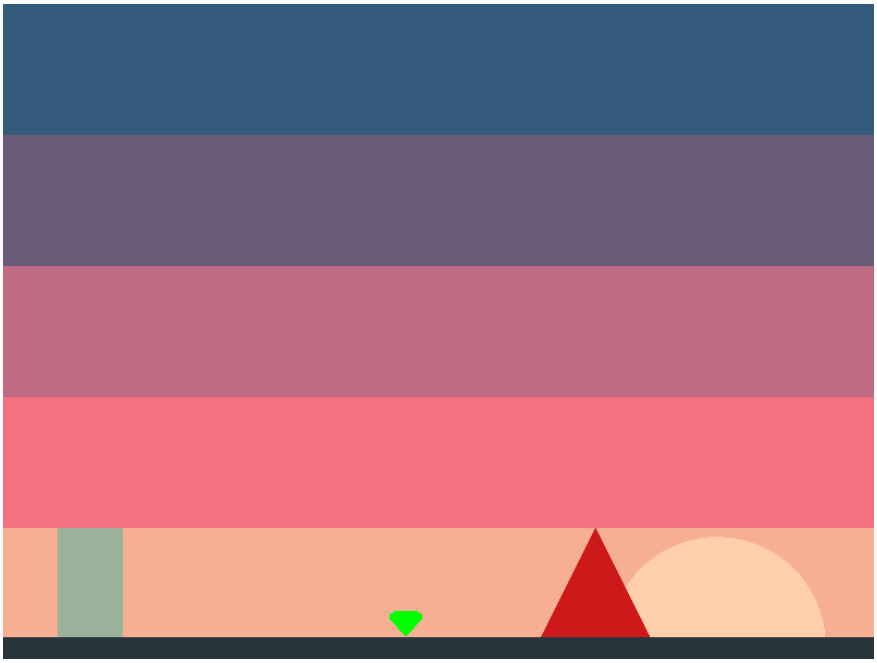
input A	input B	output
0	0	0
0	1	1
1	0	1
1	1	1

XOR

input A	input B	output
0	0	0
0	1	1
1	0	1
1	1	0

Binary operations are some of the most fundamental computer operations and can be physically built with a few transistors. Given the simplicity of these operations, it also makes for a fast method of calculating collisions.

Bitmasks



Let's take a look at this scene for a moment and identify from the crudely drawn shapes some potential entities:

All these entities fall into unique categories in that we want each of them to collide with certain other entities. If this were a game, we'd define each category with a unique binary digit, or bit, so let's first do that:

entity	category
sun	0000
player	0001
powerup	0010
enemy	0100
ground	1000

Let's set some rules for each of these entities. For instance, we want the player to collide with the powerup (0010), enemy (0100), and of course the ground (1000). To tell the game engine this, we create a *bitmask* for the fixture. This is a binary number with all the bits switched on that we want the entity to collide with. In other words, the player's bitmask would be (1110). We left the first bit blank so that the player can't collide with other potential players (player 2). Let's update the table with the bitmask we want each entity to have:

entity	category	bitmask
sun	0000	0000
player	0001	1110
powerup	0010	1001
enemy	0100	1001
ground	1000	1111

So how does it all come together and work? When two entity's fixtures contact, a binary AND operation is performed again the entity's bitmask and the other entity's category. If the resulting number isn't 0000 then we have a collision. Take the player and enemy for instance:

```
0001   player's category
1001   enemy's bitmask
----
0001   we have a collision
```

And how about the enemy and the powerup:

```
0100   enemy's category
1001   powerup's bitmask
----
0000   we have NO collision
```

Armed with this knowledge, we can assert the following information from the table above:

- The sun collides with nothing (and doesn't even get a category). It's just in the background and non-interactive.
- The player collides with everything except other players (and of course the sun).
- The powerup collides only with the ground and players.

- The enemy collides only with the ground and players.
- The ground collides with everything.

Copy or download the "collision" game from the example code and run it:

https://github.com/RVAGameJams/learn2love/tree/master/code/collision

Do the entities interact as expected? Take a look inside the *entities* folder to see the particular function being called to accomplish apply the categories and bitmasks to each entity– Fixture:setFilterData

```
-- square.lua
...
square.category = tonumber('0001', 2)
square.mask = tonumber('1110', 2)
square.group = 0
...
square.fixture:setFilterData(square.category, square.mask, square.group)
```

The examples above only use 4 bits for the category and mask bceause that's all we needed, however LÖVE supports up to 16 bits for the category and bitmask (`0000000000000000`). The *group* property isn't used and should be set to 0 when it isn't. We haven't mentioned groups before because if you know how to use categories and bitmasks then you don't need to use groups as categories and bitmasks offer a more powerful way of doing the same thing. That being said, collision groups should be relatively straight-forward to learn about so it will be left up as an exercise to read and study.

Exercises

- Play with the bitmasks. Can you make the enemy collide with the powerup instead of the player?
- Take a look at how groups work as described in Fixture:setGroupIndex. This is a simpler, but more limited method of detecting collision. Can it be used to imitate the collision rules above?

Networking (part 1)

When creating a program such as a game, one of the first things to consider should be whether it is a networked application. Such a choice will radically change the structure and complexity of the application. To build a networked ("online") multiplayer game, we must understand some networking basics. Some of this information is oversimplified, but let's establish a baseline of knowledge.

Internet protocol (IP)

Networks are possible because computers agree on a way to communicate with each other. Like ogres, messages sent across the internet have many layers. Each layer represents a different protocol that interprets how the message should be handled. The internet protocol (IP) tells computers how to relay messages to their intended destination. There are two things we need to know about this protocol: **IP addresses** and **ports**.

Every device connected to the internet has an IP address assigned to it when it connects. Messages sent out from your device are sent with a destination IP address attached so it knows where to go. Messages are relayed from one machine to another until it reaches the destination machine's address.

If you open your terminal or command prompt and type `ping google.com` you will get a response back that tells you the destination IP address; The IP address of the server running the google.com homepage you see. You may even be able to type that IP address into your web browser and it will direct you to the website in the same fashion typing google.com in the address bar would (although this won't work for all websites because of unrelated, complicated reasons). Let's say you connected to google.com through the IP address 172.217.7.14. You're actually connecting to that IP through a specific port. Ports are represented as numbers, so that IP like most every other website on the internet is accessed through port 443.

IP ports are like the maritime ports that harbor ships. A single destination can have multiple ports for different purposes. If I were bringing in a military vessel I may go to a different port than a commercial vessel.

Depending on your intentions for a network connection you will use different IP ports. For instance, if you are trying to view a website located at 172.217.7.14 you will use port 443 for an HTTPS connection, port 80 for an HTTP connection (if allowed), and if I am an administrator of the machine running on 172.217.7.14 I will use a completely different port to establish a backdoor connection such as port 22.

For our example program we will try connecting to a special reserved IP address, 127.0.0.1. This IP address is your machine's own IP address it uses when it wants to connect to itself. Since we'll be testing our program by running both copies on the same computer we won't need to worry about multiple IP addresses for now.

For the port you have a range from 0 to 65535 and it doesn't really matter which one you use so long as it's not already in use or being reserved for other purposes. We'll pick a random one that is unlikely to be in use by other programs... 6789.

Transport layer

The transport layer decides how your data will be packaged and streamed. You have a choice on a few different protocols for the transport layer. Understanding the details of each protocol in the transport layer isn't too important for this section of the book but let's discuss why we may want to use one or the other.

- **TCP** - This protocol provides different features to make sure data doesn't get corrupt. Most notably, it waits for a confirmation response from the other end to make sure the message was received. If a response isn't received by a certain timeout then the connection is considered a failure. Websites use TCP 99% of the time because of its reliability and ensuring you've received the site's full content.
- **UDP** - This protocol sends data to a server and expects no response back. Sending data without confirming it reaches the destination could lead to less reliable data transportation. However, less back and forth communication could mean a faster connection. This protocol is used by applications needing to send lots of data quickly, like an audio stream or a video game. This is the protocol we'll use.

Imagine you have two players needing to communicate their position with each other so we decide to use UDP. You may send messages back and forth several times a second to communicate your positions. Since you are sending data so rapidly, if one of those messages is lost then the player position can be re-synchronized next message. This is fast and unless one of the players has a faulty internet connection you typically won't notice a small jitter or hiccup every now and then.

Now imagine another scenario where we want to send a message that a player gained an extra life. If we were using UDP and that message got lost, we could have two online players with out-of-sync information that would ultimately jeopardize gameplay. One solution around this would be to use TCP for mission-critical messages and UDP for everything else. Another solution is to keep all messages in UDP, but to write a callback in Lua around our mission-critical messages to check that we get a reply. Yup, you can have your application send UDP messages and expect a response but even though UDP doesn't have this feature as part of its protocol you can still program in your application a timeout that expects a response. This sounds like a lot of work, but Lua and many other languages have libraries available you can require in your project that do this for you. We'll see how easy this is later on.

Application layer

Finally we have the protocol we create for each running copy of a game to know how to communicate once a connection is established. For instance if a message with the string `"ping"` is being received, we may want to respond `"pong"`. The more complicated the game is, the more complicated the protocol will be. Let's check out one of the libraries Lua offers for networking and build a test program with a basic application protocol where the server responds to `"meow"` with `"bark"` and the client responds to `"bark"` with `"meow"`. As you can guess this will lead to an infinite back-and-forth conversation between the two hosts if we are successful.

ENet

There are several third-party libraries for Lua for networking. LÖVE includes two of the most popular, LuaSocket and lua-enet. LuaSocket is very flexible and allows you to create TCP and UDP connections. Lua-enet is built on top of the ENet library, a simple yet high performance networking library. It uses UDP, but handles everything around the transport layer for us so we can focus on our application layer. It even does message confirmation over UDP for us when we need it to so we get the best of both worlds. Let's create a server and client program in LÖVE and we'll run them separately, connecting them to each other.

Our server application

Create a folder called `server` and in it create a file named `server.lua`. We'll start by requiring `enet`:

```
-- server/server.lua
local enet = require('enet')
```

This file will return a table of functions for starting and stopping the server. To start the server, we need to create a host and tell it which IP address and port it is running on. Let's create a `server.start` function that does just that:

```
-- server/server.lua
local enet = require('enet')

local host
local server = {}

server.start = function()
  host = enet.host_create('127.0.0.1:6789')
end

return server
```

The IP address is `127.0.0.1` as we said we would use. That is telling ENet we want to start the server on our machine's local address. The IP address is followed by a colon (`:`) then the port number (`6789`) which is an arbitrary port that should be free to use. If we create a main.lua file we can require server.lua and create a server when LÖVE starts.

```
-- server/main.lua
-- Our server application
local server = require('server')

love.load = function()
  server.start()
end
```

If we try and run this, nothing will happen. Let's define `love.draw` and print some text to tell us when someone connects to our server:

```
-- server/main.lua
-- Our server application
local server = require('server')

love.load = function()
  -- Keep text pixels sharp and intact instead of blurring
  -- https://love2d.org/wiki/FilterMode
  love.graphics.setDefaultFilter('nearest', 'nearest')

  server.start()
end

love.draw = function()
  -- Scale up the size of the text being printed
  local transform = love.math.newTransform(0, 0, 0, 3)
  if server.is_connected() then
    love.graphics.print('client connected to us (see console)', transform)
  else
    love.graphics.print('server started... awaiting clients', transform)
  end
end

-- It's convenient to be able to press escape to close the program
love.keypressed = function(pressed_key)
  if pressed_key == 'escape' then
    love.event.quit()
  end
end
```

With this done, we need to figure out how the server knows someone is connected. We call `server.is_connected()` in `love.draw`, so let's start by defining that:

```
-- server/server.lua
local enet = require('enet')

local host
local received_data = false
local server = {}

server.start = function()
  host = enet.host_create('127.0.0.1:6789')
end

server.is_connected = function()
  return received_data
end

return server
```

Ok, so `server.is_connected()` will return the value of `received_data` which defaults to `false`. Now the part that does all the action:

```
-- server/server.lua
local enet = require('enet')

local host
local peer
local received_data = false
local server = {}

server.start = function()
  host = enet.host_create('127.0.0.1:6789')
end

server.is_connected = function()
  return received_data
end

server.update = function()
  if not host then return end
  local event = host:service()
  if event then
    received_data = true
    peer = event.peer
    print('----')
    for k, v in pairs(event) do
      print(k, v)
    end
    event.peer:send('bark')
  end
end

return server
```

Let's take a close look at `server.update` piece by piece. First thing is an `if` statement to check that `host` is defined. If `server.update` is called before `server.start` then it won't be so there is no server update to be made. If our server host was created and we get past the if-statement check, we call `host:service()`. If we read the documentation for `host:service` we can see the purpose of calling this is to check for any incoming packets (messages) and send out any we have queued up. If we receive any, we will get back an `event` table. If we do get an event table, we'll change `received_data` to `true` (which in turn means `server.is_connected()` now returns `true`). Next we will capture the peer (the client) that sent us this data which we can use to send messages to later:

```
peer = event.peer
```

While we have the event table, let's just iterate over it and print its contents to the console:

```
for k, v in pairs(event) do
  print(k, v)
end
```

Then finally we'll send the client a message that simply reads "bark".

```
event.peer:send('bark')
```

We can now call `server.update` inside our game loop's `love.update` function:

```
love.update = function()
  server.update()
end
```

We need to test our server, but to test our server, we need a client.

Our client application

Create a "client" folder like the "server" folder created above. Most of the code will be identical to our server. The main difference is that when we create a host we won't pass it an IP address and port to serve on, but instead will tell it to connect to the address and port the server is running on.

```
-- client/main.lua
-- Our client application
local client = require('client')

love.load = function()
  -- Keep text pixels sharp and intact instead of blurring
  -- https://love2d.org/wiki/FilterMode
  love.graphics.setDefaultFilter('nearest', 'nearest')

  client.start()
end

love.draw = function()
  -- Scale up the size of the text being printed
  local transform = love.math.newTransform(0, 0, 0, 3)
  if client.is_connected() then
    love.graphics.print('connected to server (see console)', transform)
  else
    love.graphics.print('establishing a connection...', transform)
  end
end

love.keypressed = function(pressed_key)
  if pressed_key == 'escape' then
    love.event.quit()
  end
end

love.update = function()
  client.update()
end
```

```
-- client/client.lua
local enet = require('enet')
local client = {}
local host
local peer
local received_data = false

client.start = function()
  host = enet.host_create()
  peer = host:connect('127.0.0.1:6789')
end

client.is_connected = function()
  return received_data
end

client.update = function()
  if host then
    local event = host:service()
    if event then
      received_data = true
      print('----')
      for k, v in pairs(event) do
        print(k, v)
      end
      event.peer:send('meow')
    end
  end
end

return client
```

If we receive a message from the server we'll "meow" back at it.

Testing things out

If you run the server you will see a message saying "server started... awaiting clients". Since we are printing to the console, if you are running this code on Windows remember that you will need to enable the console. This can be done by creating a conf.lua file in both the client and server folders.

```
-- LÖVE configuration file

love.conf = function(t)
  t.console = true         -- Enable the debug console for Windows.
  t.window.width = 800     -- Game's screen width (number of pixels)
  t.window.height = 600    -- Game's screen height (number of pixels)
end
```

If the server is up and running with the console enabled, go ahead and start the client with its console enabled too. You should immediately see a flood of events printing out in the server and client consoles.

Server console:

```
----
peer    127.0.0.1:58384
channel 0
data    meow
type    receive
```

Client console:

```
----
peer    127.0.0.1:6789
channel 0
data    bark
type    receive
```

This will go back and forth until you close either one of them. If you close one though, the messages will stop and it will just sit there. If you close the server first, for instance, the client will sit there then after several seconds a message will appear:

```
----
peer    127.0.0.1:6789
data    0
type    disconnect
```

Normally a disconnect like this wouldn't be detected with UDP, but the ENet library sends "heartbeat" messages back and forth to make sure both peers are still connected to each other. The timeout is defined to be somewhere between 5 and 30 seconds before the peer realizes it has been disconnnected from the other one. Just to polish things off here, let's make ENet send a disconnect event to the other peer immediately when we are closing our application. The lua-enet documentation lists a function we can invoke to do that, `peer:disconnect_now`. LÖVE has a `love.quit` callback that is called when our application is closing. We can write a `server.disconnect` function and call it from `love.quit`.

Server:

```
-- server/main.lua

...

love.quit = function()
  server.disconnect()
end
```

```
-- server/server.lua

...

server.disconnect = function()
  if peer then
    peer:disconnect_now()
    peer = nil
  end
  host = nil
  received_data = false
end
```

The `client.disconnect` code would be identical.

To see this full example or if you have any problems getting your code to run check out the code on GitHub: https://github.com/RVAGameJams/learn2love/tree/master/code/networking-1

In the next part we will look at network architecture and add entities to the screen to work with.

Exercises

- What happens if you try to connect multiple clients to the server? What about running multiple servers on the same IP address and port? Why does it behave like it does?

Networking (part 2)

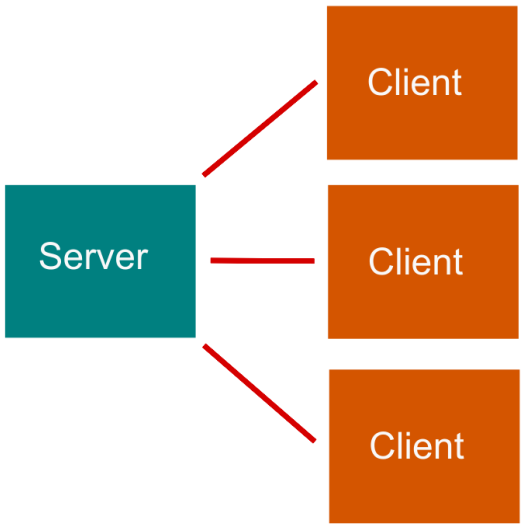
In the previous section we made two applications that could talk to each other. One application was the server and the second one connecting to it was the client. In game design this style of networking can be described as a direct connection.

Direct connection

In a direct connection one of the players takes on the role of server, meaning their game world is the ultimate authority if there are any discrepancies or out-of-sync communication between the two. This also means the server player can find ways to cheat and exploit the game.

One of the advantages to this setup is since you are directly connected to each other, you get as minimal lag as possible. This advantage doesn't hold true if there are more than 2 players. If player 1 is the server and player 2 and 3 are connected to player 1, then player 2 and 3 have to relay updates to each other through player 1 instead of directly to each other. Outside of 2-player games, this setup isn't as popular as having a dedicated server.

Dedicated server



Dedicated servers are exactly what they sound like. They are hosts dedicated to serving players. The difference here is all players are clients and the server is a neutral ground where players can connect and communicate indirectly with each other through it. Servers typically run a modified version of the game code that has no user interface and therefore can run on a less expensive computer. If one of the players is detected cheating the server can detect that something is wrong and kick them from the game. The server is the ultimate authority over the state of the game world.

Our network setup will sort of be a mix between the two styles of networking. We'll have a dedicated server that doesn't participate in the gameplay, but the server will have a graphical interface so we can view what is going on during our testing.

Consolidating our code

Rather than managing two folders of code like in the previous section, we'll combine the code and use a menu system to select between being a server and being a client. The menu code isn't important to this tutorial so try to focus on the client and server code as before. The refactored code can be found in the code repository here.

Given the amount of files it is easiest to download the zip of the whole project where you will find the relevant files inside `code/networking-2`: https://github.com/RVAGameJams/learn2love/archive/master.zip

Once downloaded, when you run the program you should be greeted with a menu screen like so:

Test it out and confirm you can connect a server and client instance with the new code.

Ok. The modifications to `main.lua` should be easy enough to understand. Let's take a look at that and the new "net" service first before we begin making any modifications. At the top of the file we're loading the net and menu services then telling the menu service which menu to load on startup:

```
-- main.lua

local menu_service = require('services/menu')
local net_service = require('services/net')

love.load = function()
  -- Keep text pixels sharp and intact instead of blurring
  -- https://love2d.org/wiki/FilterMode
  love.graphics.setDefaultFilter('nearest', 'nearest')

  menu_service.load('main-menu')
end
```

Next, if a key is pressed it will pass that pressed-key event to the menu service. If we are in the game and no menu is loaded, the menu service will do nothing with the event.

```
love.keypressed = function(pressed_key)
  menu_service.handle_keypress(pressed_key)
end
```

Inside `love.draw` we have a similar story. If we have an active menu then `menu_service.draw()` will draw it Otherwise it won't do anything. (If you open `services/menu.lua` you will see the draw function where this all happens.)

```
love.draw = function()
  menu_service.draw()
  -- Scale up the size of the text being printed
  local transform = love.math.newTransform(0, 0, 0, 3)
  if net_service.is_connected() then
    love.graphics.print('peer connected (see console)', transform)
  end
end
```

Another thing inside `love.draw` is a check to see if we've made a connection either as server or client (using `net_service.is_connected()`) then draw the "peer connected" text on the screen as before. We use the word "peer" as a generic term to refer to either the client we're connected to (if we're the server) or the server (if we're the client).

Inside `love.update` and `love.quit` we have combined the code we had before and added a `menu.update()` call. If there is a menu, update it. If either a server host or client host is running, `net.update()` will update it.

```
love.update = function()
  menu_service.update()
  net_service.update()
end

love.quit = function()
  net_service.disconnect()
end
```

So everywhere we were calling "server" or "client" we just call `net_service` and it will do it's thing no matter the type of connection. Let's open up `net.lua` and we'll see something very close to the original code:

```
-- net.lua

local enet = require('enet')

-- Populate one or the other depending if we start a server or client host
local client_host
local server_host

-- As a server, we want to keep track of all the connected clients
local peers = {}
local received_data = false

-- The service we will be returning
local net = {}
```

At the top of the file we create some empty local variables. The `net` table is full of functions that are being used in `main.lua` and elsewhere.

```
net.start_server = function()
  server_host = enet.host_create('localhost:6789')
end

net.start_client = function()
  client_host = enet.host_create()
  server_host = client_host:connect('localhost:6789')
end
```

On the menu when you select "Host" or "Join", the `net.start_server` and `net.start_client` functions are being called respectively.

Below that are some functions to check what kind of connection we have:

```
net.is_connected = function()
  return received_data
end

net.is_client = function()
  return client_host and true or false
end

net.is_server = function()
  return server_host and not client_host
end
```

Then finally we have the `update` and `disconnect` code like we originally had in the server and client services, but combined. One addition to the disconnect function is we are looping over the `peers` list. The `peers` list exists because we are expecting to have multiple players connect to the server and if the server is running, it will want to disconnect from them all when the game quits.

Communication layer

Before we update the code, let's discuss the functionality and draw out the network communication for that functionality.

- When connecting to the server you should have a controllable player spawn on screen
- When you move your player, your peers should be able to see your player move
- When other peers move their players, you should be able to see them move
- Each player should look different so you know which one is you

Client	Server	Description
(Create a host)	(Create a host)	Both client and server's net services are booted. No communication has been made between the two at this point.
Send "connect" message to server		Client sends a connect message automatically. This will be interpreted as a request to join the game.
	(Spawn entity)	Server generates and stores all the entities in the game. When spawning an entity, associate the client's connection ID with it.
	Respond "your-id|4177457821|100|500"	Server responds by passing the client a `"your-id"` message. All messages sent must be strings so in this case where multiple values need to be embedded in the message, each value is separated by a pipe ("|") character to help us re-separate the values when receiving the message client side. The first value is the unique player ID that the server has assigned the client, followed by the X position then Y position.
	Send "peer-id|233142890|326|177"	Server sends the new client other peers that need to be spawned on screen. Included are the ID, X and Y position of that player.
	Send "move|233142890|327|177"	Server is letting the client know the player with the ID "233142890" has changed position. Notice the updated X position.
	Send "move|233142890|328|177"	Another move update.
Send "move|100|502"		Client is letting the server know it is moving its player too.

Client	Server	Description
	Send "peer-id|81850530|500|500"	Another player has joined the server.

Adding entities

On connection from a client, the server needs to spawn entities so let's create an entity service for handling all our entity-related needs:

```
-- entity.lua

local entity_service = {}

-- All player entities
entity_service.entities = {}

entity_service.spawn = function(player_id, x_pos, y_pos)
  return {
    -- TODO: We'll add a color randomizer later
    color = {1, 1, 1, 1},
    id = player_id,
    -- TODO: We'll add a shape randomizer later too
    shape = love.physics.newPolygonShape(0, 0, 50, 0, 50, 50, 0, 50),
    x_pos = x_pos,
    y_pos = y_pos
  }
end

entity_service.draw = function(entity)
  love.graphics.setColor(entity.color)
  local points = { entity.shape:getPoints() }
  for idx, point in ipairs(points) do
    if idx % 2 == 1 then
      points[idx] = point + entity.x_pos
    else
      points[idx] = point + entity.y_pos
    end
  end
  love.graphics.polygon('line', points)
end

return entity_service
```

These entities will just be basic shapes. No world, body, or fixtures to worry about as dealing with physics is a bit out of the scope of this section.

Now we'll do some heavy upgrades to the net service. Somewhere near the top of the file we'll define two tables with callbacks that will get invoked when an event comes in. They'll be empty functions for now and we'll fill them out as we go.

```
-- Callbacks to invoke when certain events are received from a peer
-- Define a callback to handle every type of message in our application protocc
local message_handlers = {
  ['your-id'] = function(message, event, is_server)
  end,
  ['peer-id'] = function(message, event, is_server)
  end,
  ['move'] = function(message, event, is_server)
  end
}
-- These event types are defined by Lua-enet. A "receive" type of event
-- is a generic event that carries any of the messages above.
local event_handlers = {
  connect = function(event, is_server)
  end,
  disconnect = function(event, is_server)
  end,
  receive = function(event, is_server)
  end
}
```

Then we'll modify the `net.update` function so it can call one of the three callbacks inside the `event_handlers` table:

```
net.update = function()
  local host = client_host or server_host
  if not host then return end
  local event = host:service()
  if event then
    received_data = true
    -- event.type will be either "connect", "disconnect", or "receive"
    event_handlers[event.type](event, net.is_server())
    -- Print out the event table for debug purposes
    print('----')
    for k, v in pairs(event) do
      print(k, v)
    end
  end
end
```

So you see, `event_handlers.connect` will be called when a "connect" type event comes in and we'll pass it the event and `net.is_server()` boolean as its two parameters. Now we can go back and fill out the "connect" event handler. Read each code comment as there is a lot going on here in just a few lines of code.

```
local event_handlers = {
  connect = function(event, is_server)
    -- Only the server needs to do stuff here on connect
    if is_server then
      -- event.peer:connect_id() provides us with a unique number.
      -- We'll convert that number to a string and use it as the player ID.
      local player = entity_service.spawn(tostring(event.peer:connect_id()), 1
      -- Store this player in the player table with the player ID as the key.
      entity_service.entities[player.id] = player
      -- Send the initial "your-id" message back to the connecting client so th
      event.peer:send('your-id|' .. player.id .. '|' .. player.x_pos .. '|' ..
    end
  end,
  disconnect = function(event, is_server)
    -- TODO: Add code to remove entities when a client disconnects
  end,
  receive = function(event, is_server)
    -- TODO: Add code to parse "receive" events and call the appropriate messag
  end
}
```

Since we're using the entity service in net.lua we'll need to require it at the very top:

```
local entity_service = require('services/entity')
```

Now that the player receives the message to spawn an entity, we can fill out the "receive" event handler.

```
local event_handlers = {
  connect = function(event, is_server)
    -- Only the server needs to do stuff here on connect
    if is_server then
      -- event.peer:connect_id() provides us with a unique number.
      -- We'll convert that number to a string and use it as the player ID.
      local player = entity_service.spawn(tostring(event.peer:connect_id()), 1
      -- Store this player in the player table with the player ID as the key.
      entity_service.entities[player.id] = player
      -- Send the initial "your-id" message back to the connecting client so th
      event.peer:send('your-id|' .. player.id .. '|' .. player.x_pos .. '|' ..
    end
  end,
  disconnect = function(event, is_server)
    -- TODO: Add code to remove entities when a client disconnects
  end,
  receive = function(event, is_server)
    -- Extract the message out from the event and call the appropriate callbac
    local message = {}
    for match in (event.data .. '|'):gmatch('(.-)|') do
      table.insert(message, match)
    end
    message_handlers[message[1]](message, event, is_server)
  end
}
```

Don't let this code feel intimidating as it's only taking the string message from the event (`"your-id:87335:500:500"`) and splitting it into a list table (`{ "your-id", "87335", "500", "500" }`) so we can work with the data. Once we have the message fragments, we call `message_handlers[message[1]]()` , where `message[1]` will be one of the keys in the `message_handlers` table: `"your-id"` , `"peer-id"` , or `"move"` . Let's fill out the "your-id" handler first:

```
-- Callbacks to invoke when certain events are received from a peer
-- Define a callback to handle every type of message in our application protocc
local message_handlers = {
  ['your-id'] = function(message, event, is_server)
    local player_id = message[2]
    local x_pos = message[3]
    local y_pos = message[4]
    entity_service.player_id = player_id
    entity_service.entities[player_id] = entity_service.spawn(player_id, x_pos,
  end,
  ['peer-id'] = function(message, event_is_server)
  end,
  ['move'] = function(message, event, is_server)
  end
}
```

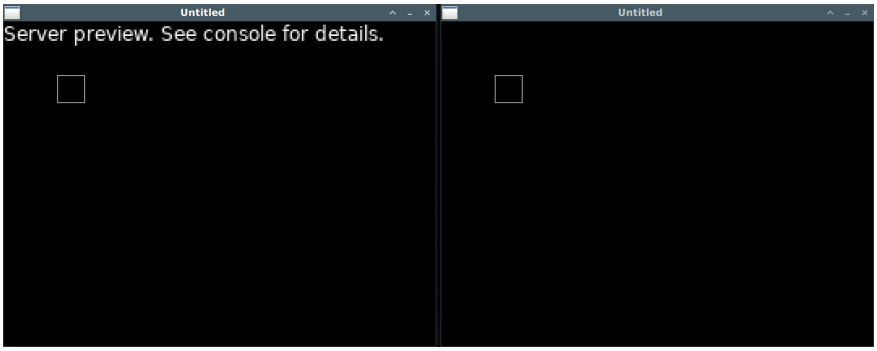
Now when the client gets the "your-id" message, it can spawn an entity too. Let's add the appropriate code to **main.lua** for drawing entities so we can see if our protocol is working so far:

```
-- main.lua

local entity_service = require('services/entity')
```

...

```
love.draw = function()
  menu_service.draw()
  -- Scale up the size of the text being printed
  local transform = love.math.newTransform(0, 0, 0, 3)
  if net_service.is_connected() and net_service.is_server() then
    love.graphics.setColor({1, 1, 1, 1})
    love.graphics.print('Server preview. See console for details.', transform)
  end
  for _, entity in pairs(entity_service.entities) do
    entity_service.draw(entity)
  end
end
```

If you've updated everything correctly, you'll see this happen when a server and client run side-by-side:



If you get an error, remember to check the line number and filename where the error occurred and make sure the code looks similar to how it is above. If you get stuck, there will be a full example available at the bottom of this section.

If everything is working for you then fantastic. This means the server and client are successfully syncing an entity state with each other.

Next let's add movement so we can see live updates happening between the two game windows. Inside the services folder, we'll create an **input.lua** file that returns an empty table:

```
-- input.lua

return {}
```

Why are we returning an empty table? Well, it's only empty right now, but as keys are pressed and released our table will get updated. Let's update `love.keypressed` and add a `love.keyreleased` function to **main.lua** to see how that works:

```
-- main.lua

local input_service = require('services/input')
```

...

```
love.keypressed = function(pressed_key)
  menu_service.handle_keypress(pressed_key)
  input_service[pressed_key] = true
end

love.keyreleased = function(released_key)
  input_service[released_key] = false
end
```

Now when we press some arrow keys for instance, our table in **input.lua** will look more like this in memory:

```
{
  left = true,
  right = false,
  up = true,
  down = false
}
```

Now inside **entity.lua** we'll add an `entity_service.move` function that checks for input changes and moves the entity if any of the arrow keys are pressed:

```
-- entity.lua

local input_service = require('services/input')
```

...

```
entity_service.move = function()
  local player = entity_service.entities[entity_service.player_id]

  -- Don't let the player press up and down at the same time
  if input_service.up and not input_service.down then
    player.y_pos = player.y_pos - 2
  elseif input_service.down and not input_service.up then
    player.y_pos = player.y_pos + 2
  end

  -- Don't let the player press left and right at the same time
  if input_service.left and not input_service.right then
    player.x_pos = player.x_pos - 2
  elseif input_service.right and not input_service.left then
    player.x_pos = player.x_pos + 2
  end
end
```

This will cause our entity to move across the client's screen, but the server won't get these updates unless the client sends them over. We need to go back to **main.lua** and send a "move" message. Inside `love.update` check to see if the player position has changed and send a move message to the server if so:

```
love.update = function()
  menu_service.update()
  -- Check to see if a player has spawned and update its movement if any direct
  if entity_service.player_id then
    local player = entity_service.entities[entity_service.player_id]
    local old_x = player.x_pos
    local old_y = player.y_pos
    entity_service.move()
    if player.x_pos ~= old_x or player.y_pos ~= old_y then
      net_service.send('move|' .. player.id .. '|' .. player.x_pos .. '|' .. p
    end
  end
  net_service.update()
end
```

The `net_service.send` function we haven't defined that yet. Jump back over to **net.lua** and we'll define that for sending either client or server messages if needed:

```
net.send = function(message)
  if net.is_client() then
    server_host:send(message)
  else
    server_host:broadcast(message)
  end
end
```

Now we are sending a message to the server when we move, but we need to have the server read the message and update the entity position on its end too. Let's fill out the "move" message handler in net.lua to accomplish this:

```
local message_handlers = {
  ['your-id'] = function(message, event, is_server)
    local player_id = message[2]
    local x_pos = message[3]
    local y_pos = message[4]
    entity_service.player_id = player_id
    entity_service.entities[player_id] = entity_service.spawn(player_id, x_pos,
  end,
  ['peer-id'] = function(message, event, is_server)
    -- TODO: handle peer-id messages for when more players join the server
  end,
  ['move'] = function(message, event, is_server)
    local player_id = message[2]
    local x_pos = message[3]
    local y_pos = message[4]
    entity_service.entities[player_id].x_pos = x_pos
    entity_service.entities[player_id].y_pos = y_pos
    if is_server then
      -- Relay this message to the other players
      for id, peer in pairs(peers) do
        if id ~= player_id then
          peer:send(event.data)
        end
      end
    end
  end
}
```

Notice the extra server check. If the server received the "move" command it should relay it over to any other connected player so they can see you moving too. The `peers` table needs to be defined inside net.lua *above* the message handlers or you may get an error when it tries to loop over the table and sees a nil value that hasn't been defined yet.

Try it out again and we should see the square moving on both screens now.

All that hard work is starting to pay off! We have a few more changes to make for this to be fully functional though. If you try and run the game now with multiple clients, the players won't see each other. In our application protocol we defined a `peer-id` message so that when new players connect we receive information to spawn them.

Inside net.lua, go ahead and fill out the "peer-id" message handler so that it spawns an entity when invoked:

```
['peer-id'] = function(message, event, is_server)
  local player_id = message[2]
  local x_pos = message[3]
  local y_pos = message[4]
  entity_service.entities[player_id] = entity_service.spawn(player_id, x_pos,
end,
```

Now the server needs to send all the peers to a new player connecting, but it also needs to send new players to pre-existing peers during a connect event. We'll update the "connect" event handler to do that:

```
connect = function(event, is_server)
  -- Only the server needs to do stuff here on connect
  if is_server then
    -- event.peer:connect_id() provides us with a unique number.
    -- We'll convert that number to a string and use it as the player ID.
    local player = entity_service.spawn(tostring(event.peer:connect_id()), 1
    -- Store this player in the player table with the player ID as the key.
    entity_service.entities[player.id] = player
    -- Send the initial "your-id" message back to the connecting client so th
    event.peer:send('your-id|' .. player.id .. '|' .. player.x_pos .. '|' ..
    -- Let all the other peers know about this player
    for _, peer in pairs(peers) do
      local peer_player = entity_service.entities[tostring(peer:connect_id())
      peer:send('peer-id|' .. player.id .. '|' .. player.x_pos .. '|' .. play
      event.peer:send('peer-id|' .. peer_player.id .. '|' .. peer_player.x_po
    end
    -- Add this peer to the peer list
    peers[tostring(event.peer:connect_id())] = event.peer
  end
end,
```

After sending the new client their id as "your-id", we send that id to every other client as "peer-id" in the for loop. Notice we add the new connecting client to the peer table at the very end. We do this after looping over the peer list so we don't send that client a "peer-id" message of themselves. Again, when registering peers and entities we call `tostring()` on the `connect_id()` to ensure we are storing IDs as strings rather than numbers. Storing data in tables it matters whether you store them as keys or numbers. See 1.14 - Tables (part 2) to see what I mean.

Anyways, try running the game now with multiple clients connecting to the server and you will see each entity can move separately and the changes will be synchronized across all clients. One change to make the entities easier to distinguish would be to randomize their color and shape. Let's modify `entity_service.spawn` inside **entity.lua**:

```
entity_service.spawn = function(player_id, x_pos, y_pos)
  local colors = {
    {1, 0, 0, 1},
    {0, 1, 0, 1},
    {0, 0, 1, 1},
    {0, 1, 1, 1},
    {1, 0, 1, 1},
    {1, 1, 0, 1},
    {1, 1, 1, 1}
  }
  local shapes = {
    love.physics.newPolygonShape(25, 0, 50, 50, 0, 50),
    love.physics.newPolygonShape(0, 0, 50, 0, 50, 50, 0, 50),
    love.physics.newPolygonShape(12, 0, 36, 0, 49, 15, 49, 33, 36, 49, 12, 49,
  }
  return {
    -- Cycle through the list of colors based on whatever the player id is
    -- Calling tonumber() to make player_id a number instead of a string so we
    color = colors[(tonumber(player_id) % #colors) + 1],
    id = player_id,
    shape = shapes[(tonumber(player_id) % #shapes) + 1],
    x_pos = x_pos,
    y_pos = y_pos
  }
end
```

Here we use a modulus to cycle through the list of colors and shapes and assign one based on the pseudo-random player ID we received. This also means a player will look the same on every other players' client. Try running it again and you should see something similar:

The final networking code can be found here. Again, given the amount of files if you need the whole folder then it is easiest to download the zip of the whole project. You will find the relevant files inside `code/networking-3`: https://github.com/RVAGameJams/learn2love/archive/master.zip

Conclusion

This is about as basic in functionality as a multiplayer game can get. There are many features missing from our code. Just to name a few major ones:

- Sanity checks on messages. You can crash the server by sending it an invalid message.
- The "move" message expects a player id when the server should be able to figure this out on its own. The server can easily be fooled into accepting "move" messages from a client to move another client.
- To make things simpler there is no world or physics nor any net code to handle collisions or anything of the like.
- The entities just stick around when you close/disconnect a client.
- If you lose connection, there is no attempt to restore the connection.

There is a great set of articles by Gabriel Gambetta on the problems faced by making a action-based multiplayer games and it's worth a read to get a high-level overview of the challenges and how to reason about them.

Exercises

- Having all the players spawn at the exact same point isn't ideal. Inside net.lua in the "connect" event handler, make a list of spawn points and make the players spawn at one of the points based on their `peer:connect_id()`. Hint: the code should look similar to the color cycle code inside `entity.spawn` in entity.lua.
- Inside net.lua, complete the "disconnect" event handler and make is so when a client quits their entity is removed from the game for all peers.

Chapter 3: Programming in depth

The goal of this chapter is to touch on a variety of topics and problems faced while programming and to understand and solve them using Lua. A wider variety of topics will be covered here. One topic will lead into another which will then build on top of concepts introduced in the previous topic.

Primitives and references

Take a look at this code:

```
local string1 = "hello"
local string2 = 'hello'

print(string1 == string2)

local number1 = 14
local number2 = 14

print(number1 == number2)

local table1 = {}
local table2 = {}

print(table1 == table2)

local function1 = function() end
local function2 = function() end

print(function1 == function2)
```

What would happen if you were to run this?

In chapter 1 we learned about comparing strings with the `==` operator when we talked about booleans. Run the code above in the REPL and see what it returns:

```
true
true
false
false
```

The strings equal and the numbers equal, but why aren't the tables and functions equal since they are both empty? Try printing the tables and functions and look what happens:

```
local table1 = {}
local table2 = {}

print(table1)
print(table2)

local function1 = function() end
local function2 = function() end

print(function1)
print(function2)
```

```
table: 0x16af270
table: 0x16af220
function: 0x16ae840
function: 0x16aeff0
```

Attempting to print each value you are given back a hexadecimal number, the place in memory where those values are located. Each table and function resides in a different place in memory. So how is this relevant?

When checking data like strings and numbers, the `==` operator does indeed check that the data matches. These data types are simple and take very little effort for a computer to check that they are equal. Booleans, strings, numbers, and `nil` are all *primitive* types of data and behave this way.

When checking data like functions and tables, however, the `==` operator checks the memory location of the data on both sides of the operator and if the variables reference the same location then they are equal. In other words, the `==` operator checks these data types to see if they have the same identity. No matter how many empty tables or functions you have, each one is created with a unique identity.

```
local string1 = 'hello'
local string2 = "hello"
-- Another copy of "hello" is created in memory:
local string3 = string2
-- But these two copies are equal
print(string2 == string3)

local table1 = {}
local table2 = table1
print(table1 == table2)
```

What is the result of `print(table1 == table2)` ? Aha! Both these variables reference the same data. Quick– a magician waves two wands in front of your face and asks you to count how many wands there are. How do you know if there are really two wands or if this is just a trick with mirrors? What do you do? You take one of the wands and break it of course. If the other wand breaks then they were the same wand the entire time. Let's try that with the two objects:

```
local table1 = {}
local table2 = table1
table1.rabbit = 'white'

print(table2.rabbit) -- Equals 'white' too
```

As long as your variables reference the same table, updating the table from one variable you will see the result when checking the other variable. This doesn't work with primitive data because you're always making a copy when assigning it to a new variable name:

```
local string1 = 'hello'
local string2 = string1
string1 = 'world'

return string2
```

```
=> hello
```

Primitive versus non-primitive data types

Whenever we assign non-primitive data to a new variable, we're always referencing the original data:

```
local grocery_list = {
  'carrots',
  'celery',
  'pecans'
}

local same_list = grocery_list

grocery_list[1] = 'grapes'

return same_list[1]
```

But assigning primitive data to a variable, even primitive data inside tables, we're always making a unique copy:

```
local grocery_list = {
  'carrots',
  'celery',
  'pecans'
}

local item_copy = grocery_list[1]

print('item_copy is ' .. item_copy)

grocery_list[1] = 'grapes'

print('item_copy still is ' .. item_copy)
```

If you need to make each item in your table reference-able, you need to make each item a non-primitive data type:

```
local grocery_list = {
  { name = 'carrots', location = 'produce' },
  { name = 'celery', location = 'produce' },
  { name = 'pecans', location = 'baking' }
}

local item_reference = grocery_list[1]

print('item_reference is ' .. item_reference.name)

grocery_list[1].name = 'grapes'

print('item_reference is now ' .. item_reference.name)
```

So rather than replacing the first item in the list, the first item was retained and only modified:

```
item_reference is carrots
item_reference is now grapes
```

Cloning non-primitive data types

As we are familiar with at this point, tables are a special data type that can contain other data types. You can build structures containing strings, variables, and even other tables. That makes the table a *composite* data type, in other words, a data type with distinguishable parts. Not all languages have composite data types, but for Lua the table is one of its primary features.

One thing a programmer may want to do with a table is once constructed, create a copy of it. If there was a table for a monster in a video game, you may want to have more than one table. If you did this:

```
local enemy1 = { health = 10, strength = 12, type = 'orc' }
local enemy2 = enemy1
```

You would still only have one table. You could use a loop to copy all the values out of a table and into a brand new table. A function to do that may look like this, more or less:

```
local copy = function(orig_table)
  local new_table = {}
  for key, value in pairs(orig_table) do
    new_table[key] = value
  end
  return new_table
end

local enemy1 = { health = 10, strength = 12, type = 'orc' }
local enemy2 = copy(enemy1)
```

There is nothing terribly wrong with this method, but a more efficient way to do such a thing would be to construct each monster table inside a function instead of copying one from another. This method will be familiar already if you read and followed through the breakout game.

```
local create_orc = function(strength)
  return {
    health = 10,
    strength = strength,
    type = 'orc'
  }
end

local enemy1 = create_orc(12)
local enemy2 = create_orc(12)
```

Every time the function `create_orc` is ran, it constructs a new table from scratch. You define an orc-style table only once and don't need to read values in from one table to another. A function that constructs tables for you is a common paradigm in programming known as a *factory* function. You made a factory that builds orcs! Of course this factory function paradigm works with other non-primitive types of data as well:

```
local create_function = function()
  return function() return 1 + 1 end
end

local fn1 = create_function()
local fn2 = create_function()

print(fn1)
print(fn2)
```

A function that generates other functions? This may seem like an odd thing to want to do, but this method of programming can be quite useful as we'll see in 3.02 - Higher-order functions and later follow-up sections. One thing that should be mentioned though is that functions can also be considered a composite data type as it can return other data types, and even other functions. Composite in that you can compose higher-order functionality in the way tables can be used to compose higher-order structures.

Conclusion

When comparing or referencing data, always keep in mind whether you handling primitive or non-primitive data. If you are modifying data in one place, think if this might be affecting you somewhere else in your program. Even when writing out a `local some_module = require('some-module')` in your code `some_module` is just a table and like every other table, every reference to it can affect each other. So modifying `some_module` in two different files can have either beneficial or disastrous consequences depending how much care and regard you give your code.

Higher-order functions

In 1.07 Making functions we learned about, well, making functions. So what about higher-order functions? What are they and how do we make them? Simply put, higher-order functions are functions built on top of other functions. Here's a basic example:

```
local run_twice = function(some_function, some_data)
  some_function(some_data)
  some_function(some_data)
end

run_twice(print, 'Hello World!')
```

It can take any function and run it twice for you, in this case the `print` function, but it could be any function you pass it. Typically higher-order functions return data. Here's a trickier example that does just that:

```
local twice = function(fn, val)
    return fn(fn(val))
end

local add_four = function(num)
    return num + 4
end

return twice(add_four, 12)
```

Take a look at the bottom line for a second. We are calling the function `twice` with two arguments, the `add_four` function and the number `12`. The purpose of the `twice` function is to take a value, `12` in this case, and run it through the given function (`add_four`) twice. Now take a look inside the `twice` function. Inside it returns `fn(fn(val))`. Given what we know is being passed to this function, this can be read as saying `add_four(add_four(12))`. The order of operation says to start from the inner-most parenthesis and work your way out:

```
add_four(add_four(12))
```

becomes

```
add_four(16)
```

which becomes

```
20
```

and that is what is returned when you run the code. The power of these higher-order functions is that they are re-usable. You can give the `twice` function anything that takes and returns a value:

```
local twice = function(fn, val)
    return fn(fn(val))
end

local double = function(number)
    return number * 2
end

return twice(double, 3)
```

...or similar to our original example:

```
local twice = function(fn, val)
    return fn(fn(val))
end

local shout = function(message)
  print(message .. '!!')
    return message
end

return twice(shout, 'hello')
```

There are all examples of higher-order functions that accept a function as an argument. Another kind of higher-order function is one that *returns* another function:

```
local wrapper = function()
  return function()
    return 'You found the treasure!'
  end
end

local kinder_surprise = wrapper()
local secret = kinder_surprise()
return secret
```

When we ran `wrapper` it returned us *another function* that we had to invoke to get to the innermost value. To avoid all the variable names, you can save some time and invoke such kinds of functions like so:

```
local wrapper = function()
  return function()
        return 'You found the treasure!'
    end
end

return wrapper()()
```

Closures

Which number will print out by running the following code?

```
local number = 3

local closure = function()
  local number = 5
  return function()
    print(number)
  end
end


local print_number = closure()
print_number()
```

Strange?

Ok, so let's try a this same function-returning-a-function thing but passing in some data:

```
local adder = function(a)
  return function(b)
    return a + b
  end
end

local add_three = adder(3)

return add_three(1)
```

The `add_three` variable is assigned a unique and special function. It is assigned the inner function within the adder function, but with the data we passed in now assigned to the `a` variable. Even though the function was returned outside of the scope it was defined in, the scope's data was enclosed inside the returned function until the function was discarded and the program exited. These types of functions are common in situations where a function needs to be generated multiple times but with different data sets.

The data in the closure can also continue to be updated, giving you the ability to make storage containers for your data. Try this out:

```
local make_counter = function()
  local number = 0
  return function()
    number = number + 1
    return number
  end
end

local count = make_counter()
print(count())
print(count())
print(count())
print(count())
```

In programs like LÖVE there are callback systems where a similar effect happens:

```
local entity = require('entity')

love.draw = function()
  entity:draw()
end
```

As seen in the previous chapter, the `love.draw` callback is defined in a **main.lua** file and later invoked somewhere within the game engine. Since `love.draw` was defined in the scope where the entity variable is defined, the entity variable lives on and can be used inside `love.draw` long after the **main.lua** file is done being invoked.

Conclusion

Closures take some practice to understand and appreciate, but once you see practical examples of where and how to use them they become an indispensable item on your programming toolbelt. In the previous section we used the term *composite data* to compare primitive and non-primitive data types. In this section we saw how to go about *composing* higher-order functions. In the following pages we will cover some higher-order functions that are the building blocks for old and modern software alike.

Exercises

- In the `make_counter` example above, try generating multiple counters:

  ```
  -- Do the numbers in each counter stay in
  -- sync or are they tracked independently?
  local count_a = make_counter()
  local count_b = make_counter()
  ```

- Using the same `make_counter` example, modify it to return a table instead of a function. Within this table, define an `increment` and `decrement` function so that you can make the counter number go up or down. How would you use such a function?

Map and filter

In the previous section we practiced creating some higher order functions. In this sections we'll compose two higher-order functions commonly used in internet applications for transforming lists.

We'll start by taking a look at our grocery list to see what items we need to pick up:

```
local grocery_list = {
  {
    name = 'grapes',
    price = '7.20',
    location = 'produce'
  },
  {
    name = 'celery',
    price = '5.50',
    location = 'produce'
  },
  {
    name = 'walnuts',
    price = '6.20',
    location = 'baking'
  },
  {
    name = 'sugar',
    price = '8.00',
    location = 'baking'
  },
  {
    name = 'mayonnaise',
    price = '3.50',
    location = 'dressings'
  },
  {
    name = 'cream',
    price = '3.00',
    location = 'dairy'
  }
}
```

This list has more information than we want to see at a quick glance. If we wanted to only display a numbered list of item names, we could do so by writing a for-loop that generates a new list for us:

```
local new_grocery_list = {}
for key, value in ipairs(grocery_list) do
  new_grocery_list[key] = key .. '. ' .. value.name
end

for _, value in ipairs(new_grocery_list) do
  print(value)
end
```

Here we generated a list with a loop then looped over the list again to print our results:

```
1. grapes
2. celery
3. walnuts
4. sugar
5. mayonnaise
6. cream
```

This works great for simple code like this example, but it can get messy if you are working with many lists or if you want to transform lists to different formats.

Map

Here's our higher order function, `map`. It takes a list and a function as arguments then returns a new list.

```
local map = function(list, transform_fn)
  local new_list = {}
  for key, value in ipairs(list) do
    new_list[key] = transform_fn(value, key)
  end
  return new_list
end
```

A new list is created by looping over each item in the original list, applying your function to the item, then assigning the transformed data to the new list. Our code can be re-written to use the map function:

```
local map = function(list, transform_fn)
  local new_list = {}
  for key, value in ipairs(list) do
    new_list[key] = transform_fn(value, key)
  end
  return new_list
end

local grocery_list = {
  {
    name = 'grapes',
    price = '7.20',
    location = 'produce'
  },
  {
    name = 'celery',
    price = '5.50',
    location = 'produce'
  },
  {
    name = 'walnuts',
    price = '6.20',
    location = 'baking'
  },
  {
    name = 'sugar',
    price = '8.00',
    location = 'baking'
  },
  {
    name = 'mayonnaise',
    price = '3.50',
    location = 'dressings'
  },
  {
    name = 'cream',
    price = '3.00',
    location = 'dairy'
  }
}

local new_grocery_list = map(grocery_list, function(item, index)
  return index .. '. ' .. item.name
end)

for _, value in ipairs(new_grocery_list) do
  print(value)
end
```

Calling `map(...)` we get back the new list then we loop over it again just to print our results out. Notice how the second argument we passed into map is just a function with no name. Functions with no names are sometimes called anonymous functions. In some languages they're called lambdas, especially when used inside a higher-order function in a situation like this. The transform function takes in the item and its index and must return back a new result for `map` to put inside the new function.

Maybe a few more examples will help out, so what if we want to return another list with just the prices so we can add up how much we need to spend?

```
local price_list = map(grocery_list, function(item)
  print(item.price)
  return item.price
end)
```

```
7.20
5.50
6.20
8.00
3.50
3.00
```

Here the map function is passed in a transform function with a print statement inside it. That way it will print the item prices as it builds the list so you can see what each value will be.

If you had other lists for which you wanted to print prices, it could be done quite easily with `map`:

```
local transform_fn = function(item) return item.price end

map(grocery_list, transform_fn)
map(car_parts, transform_fn)
map(card_transactions, transform_fn)
```

Filter

Let's say we wanted to only see the things on our grocery list that are in the baking aisle. We could write a loop to do that:

```
local filtered_list = {}
for _, value in ipairs(grocery_list) do
  if value.location == 'baking' then
    filtered_list[#filtered_list + 1] = value
  end
end

for _, value in ipairs(filtered_list) do
  print(value.name)
end
```

Try running that and once it makes sense, let's think about how to turn this into a re-usable higher-order function like `map`. We'll make a function called `filter` that, like `map`, takes a list and a function. The function will return `true` if it wants to put an item in the new list or `false` if it doesn't. We'll call it the *predicate function*.

```
local filter = function(list, predicate_fn)
  local new_list = {}
  for key, value in ipairs(list) do
    -- The predicate_fn that was passed in should return
    -- a value that evaluates to either true or false.
    if predicate_fn(value, key) then
      new_list[#new_list + 1] = value
    end
  end
  return new_list
end
```

And we can use this function to filter down to just our baking items like this:

```
local filter = function(list, predicate_fn)
  local new_list = {}
  for key, value in ipairs(list) do
    if predicate_fn(value, key) then
      new_list[#new_list + 1] = value
    end
  end
  return new_list
end

local grocery_list = {
  {
    name = 'grapes',
    price = '7.20',
    location = 'produce'
  },
  {
    name = 'celery',
    price = '5.50',
    location = 'produce'
  },
  {
    name = 'walnuts',
    price = '6.20',
    location = 'baking'
  },
  {
    name = 'sugar',
    price = '8.00',
    location = 'baking'
  },
  {
    name = 'mayonnaise',
    price = '3.50',
    location = 'dressings'
  },
  {
    name = 'cream',
    price = '3.00',
    location = 'dairy'
  }
}

local filtered_list = filter(grocery_list, function(item)
  return item.location == 'baking'
end)

for _, value in ipairs(filtered_list) do
  print(value.name)
end
```

```
walnuts
sugar
```

Notice our predicate function we wrote:

```
function(item)
  return item.location == 'baking'
end
```

The operation after the `return` always returns a boolean `true` or `false`, so `filter` knows exactly what to do with the item based on those results.

You can imagine the `filter` function could be useful for processing a search query. For instance, if we wanted to see only medium-sized shirts that fit a specific price range:

```
filter(products, function(item)
  if item.type == 'shirt' then
    if item.size == 'M' then
      return item.price < 40
    end
  end
  return false
end)
```

Caveats

The filter function returns a new list, but the items in the list still reference the old list if they aren't primitives. For instance if we modified the grocery list, the filtered copy would be updated.

```
local filtered_list = filter(grocery_list, function(item)
  return item.location == 'baking'
end)

grocery_list[3].name = 'peanuts'
print(filtered_list[1].name)
```

```
peanuts
```

This behavior can be advantageous if it's expected, but it's something that should be understood about how Lua and similar programming languages work. This is explained more in 3.1 - Primitives and references.

Another thing to consider is whether or not to write the functions yourself or to use a pre-written library you can `require` into your project. Not all implementations are the same and some may perform better than others, or behave differently. Some languages have built-in versions of these functions to standardize things. Unfortunately Lua doesn't provide these functions built in or as a standard library.

At least you now know how to write them yourself if the need arises.

Exercises

- Try filtering the grocery list to only "produce" items, then mapping those results down to just the names.
- Using `filter`, now can you return the number of items in the grocery list with a price of more than 5? Hint: you will need to use tonumber() to convert the item prices to numbers for comparing.

Stack and recursion

When running a program, the interpreter (Lua in this case) keeps track of variables defined in a scope and which function you are currently in. It organizes this information into a list in memory called *the stack*. The first item in the stack is the starting point - the root of your application. Take the following example:

```
local two = function()
  print('two')
end

local one = function()
  print('one')
  two()
end

one()
```

When starting the program, the start of the stack is the top level of the module. The Lua stack calls this the "main chunk". When a function is invoked, another layer is added to the stack. Every time a function is called from another function, the stack continues to build. So with the example code above, The stack will follow the progression:

- Stack is `{ "main chunk" }`.
- Now start executing `one`. Stack is `{ "main chunk", "one" }`.
- Now start executing `two` while still in `one`. Stack is `{ "main chunk", "one", "two" }`.
- `two` is done executing. Stack is now `{ "main chunk", "one" }`.
- `one` is done executing. Stack is now `{ "main chunk" }`.
- Program exits.

This can be visualized by throwing an error at any point the program. The interpreter will give you back a *stack trace* that details where it was when the problem occurred. Lua provides a helpful `error` function for debugging that we can use here:

```
local three = function()
  error('This is an error.')
end

local two = function()
  print('two')
  three()
end

local one = function()
  print('one')
  two()
end

one()
```

Unfortunately the REPL doesn't provide us with stack traces, but if you have a Lua interpreter on your computer (`lua` command, `luajit` , or LÖVE) you will see the error message and a stack trace like this:

```
lua: test.lua:2: This is an error.
stack traceback:
        [C]: in function 'error'
        test.lua:2: in upvalue 'three'
        test.lua:7: in upvalue 'two'
        test.lua:12: in local 'one'
        test.lua:15: in main chunk
        [C]: in ?
```

From the "stack traceback" you can see the newest from the top of the stack to the oldest on the bottom. In complex programs is can be very beneficial to see which function invoked another function to help trace down how an error came about.

Understanding the stack is beneficial for more than just reading errors. Let's switch the conversation over to something seemingly unrelated for a bit.

Recursion

When thinking of loops, many programmers first think of the `for` loop or the `while` loop. Another common method is to make a function call itself. Similar to the `while` loop, you can create infinite loops like this one

```
local loop
loop = function()
  print('hello!')
  loop()
end
```

When a function invokes itself, whether directly or indirectly, this is called *recursion*. The same function will recur again and again until a condition changes. Or in the case above, `loop()` will be called unconditionally. Without a condition, any kind of loop will run infinitely (or crash trying). Here's a loop that is a little safer to run:

```
local count_to_5
count_to_5 = function(current_number)
  print(current_number)
  if current_number < 5 then
    count_to_5(current_number + 1)
  end
end

count_to_5(1)
```

Which prints:

```
1
2
3
4
5
```

One quick little aside; Notice how the function was defined in both these situations:

```
local loop
loop = function()
  ...
```

The variable was defined before the function was created. Since the function needs to access the variable inside itself, the variable needs to exist at the time the function's scope is created. Variables created after the function are unknown to the function. This is discussed in 1.17 - Scopes and is a limitation of Lua's design. Fortunately there is shorthand syntax for writing recursive functions:

```
local function count_to_5(current_number)
  print(current_number)
  if current_number < 5 then
    count_to_5(current_number + 1)
  end
end

count_to_5(1)
```

is the same as writing:

```
local count_to_5
count_to_5 = function(current_number)
  ...
```

Let's try another recursive loop:

```
local grocery_list = {
  'pumpkin',
  'pecans',
  'butter',
  'flour',
  'sugar'
}

local function print_items(list, index)
  index = index or 1
  if index <= #list then
    print(list[index])
    print_items(list, index + 1)
  end
end

print_items(grocery_list)
```

Which prints the grocery list. Don't forget the `local` at the beginning of `local function print_items`, otherwise you will accidentally generate global variables in your code when trying to define functions.

We can even re-implement our `map` function from earlier to use recursion instead of a `for` loop.

```
local grocery_list = {
  'pumpkin',
  'pecans',
  'butter',
  'flour',
  'sugar'
}

local function map(orig_list, transform_fn, new_list)
  new_list = new_list or {}
  if #new_list < #orig_list then
    local index = #new_list + 1
    new_list[index] = transform_fn(orig_list[index], index)
    return map(orig_list, transform_fn, new_list)
  end
  return new_list
end

local new_list = map(grocery_list, function(value, index)
  return index .. '. ' .. value
end)

map(new_list, function(value)
  print(value)
  return value
end)
```

Which prints:

```
1. pumpkin
2. pecans
3. butter
4. flour
5. sugar
```

Stack overflow

So what does the stack look like during recursion when a function enters itself? Here's a script to test:

```
local function recur(n)
  -- assert is like error, but takes an expression to test. If the
  -- expression passed becomes false then it throws the error message.
  assert(n < 5, 'This is a conditional error')
  print(n)
  recur(n + 1)
end

recur(1)
```

```
lua: test2.lua:2: This is a conditional error
stack traceback:
        [C]: in function 'assert'
        test2.lua:2: in upvalue 'recur'
        test2.lua:4: in upvalue 'recur'
        test2.lua:4: in upvalue 'recur'
        test2.lua:4: in upvalue 'recur'
        test2.lua:4: in local 'recur'
        test2.lua:7: in main chunk
        [C]: in ?
```

Every time the function recurs we get another addition to the stack. This can be a problem if you are looping over a large set of data because the stack will consume more and more memory as it stacks up. This can be accomplished by creating a recursive loop that runs infinitely. If you haven't tried so already, here's an easy example:

```
local function recur()
  recur()
end

recur()
```

When the stack reaches a critical size, you get a *stack overflow* error:

```
lua: test3.lua:2: stack overflow
stack traceback:
        test3.lua:2: in upvalue 'recur'
        test3.lua:2: in upvalue 'recur'
        ...
        test3.lua:2: in upvalue 'recur'
        test3.lua:2: in upvalue 'recur'
        test3.lua:2: in local 'recur'
        test3.lua:5: in main chunk
        [C]: in ?
```

With a specific `return` statement added to the loop, however, we no longer get a stack overflow:

```
local function recur()
  return recur()
end

recur()
```

This will run until you manually kill the application process. Killing it returns a somewhat mysterious stack track:

```
lua: test4.lua:2: interrupted!
stack traceback:
        test4.lua:2: in function <test4.lua:1>
        (...tail calls...)
        test4.lua:2: in function <test4.lua:1>
        (...tail calls...)
        test4.lua:5: in main chunk
        [C]: in ?
```

So how did our modification save us from overflowing our stack?

Tail call optimization

Inside a function when you return another function call, the interpreter has the ability to re-use the same layer of the stack instead of creating another layer. This works with direct recursion (function calling itself) and indirect (mutual) recursion such as two functions calling each other:

```
local one
local two

one = function()
  return two()
end

two = function()
  return one()
end

one()
```

Programming in Lua goes into greater detail on when a recursion will or won't be optimized, but the simple thing to remember is that the function(s) must return the value of invoking a function for this to work. The following will be tail-call optimized:

```
local one
local two

one = function(n)
  print(n)
  return two(n + 1)
end

two = function(n)
  print(n)
  return one(n + 1)
end

-- Count until we run out of numbers
one(1)
```

But the following won't, since it returns an operation including the function call instead of just the function call itself:

```
local one
local two

one = function(n)
  print(n)
  return 1 + two(n)
end

two = function(n)
  print(n)
  return 1 + one(n)
end

-- This won't work!
one(1)
```

The case for recursive loops

So why would we want to do recursion? It seems trickier than a `for` loop and perhaps just as easy to mess up as a `while` loop.

It's not necessarily a replacement for the `for` loop, but allows you to do certain things you can't easily do without recursion. Take this example from Rosetta Code which will flatten a list of lists into a single, flat list. It uses a `for` loop and a recursive loop in conjunction with each other:

```
local function flatten(list)
  if type(list) ~= "table" then return {list} end
  local flat_list = {}
  for _, elem in ipairs(list) do
    for _, val in ipairs(flatten(elem)) do
      flat_list[#flat_list + 1] = val
    end
  end
  return flat_list
end

local test_list = {
  {1},
  2,
  {{3,4}, 5},
  {{{}}},
  {{{6}}},
  7,
  8,
  {}
}

print(table.concat(flatten(test_list), ","))
```

Which prints:

```
1,2,3,4,5,6,7,8
```

This function isn't tail-call optimized, but it probably won't be passed a nested list deep enough to cause a stack overflow.

Here's just a few of the many situations where recursion is usually the best tool for the job:

- Sorting data
- Searching trees (nested data) in a database or nested folders in a filesystem.
- Finding the shortest path between two points
- Loops that increment or decrement in irregular patterns
- Evaluating a finite set of moves in a game like chess

The point isn't to replace the `for` loop, although you can. Take the following example, which returns the factorial of the given number (5):

```
local fact = function(n)
  local acc = 1
  for iteration = n, 1, -1 do
    acc = acc * iteration
  end
  return acc
end

print(fact(5))
```

The same functionality written with a recursive loop would look very different:

```
local function fact(n, acc)
  acc = acc or 1
  if n == 0 then
    return acc
  end
  return fact(n-1, n*acc)
end
```

...but one method wouldn't offer an advantage over the other here. Depending on the language you are working in, one method may be easier to read than the other. Maybe the language supports one type of loop and not the other. These are the factors that will often do the deciding for you.

Reduce (fold)

In previous sections we discussed many methods for iterating over data and transforming it. In this section we'll discuss another higher order function that is arguably one of the most powerful. It is a concept recognized across enough programming languages to get its own wikipedia article. Most popular languages call it **reduce**, although some languages will call it *fold* or *inject*. Here's the parameters it takes, and although the order of the parameters may be different in other languages the functionality and output will be the same.

```
reduce(list, fn, starting_value)
```

Like with `map()` and `filter()`, it takes a list you want to transform and a function (`fn`) to do the transformation. The transformation function behaves like a recursive loop like seen in the last section. Here's a function that takes a list of numbers and gives you the total sum of those numbers.

```
local list = {23, 63, 12, 48, 3}

local sum_fn = function(accumulator, current_number)
  return accumulator + current_number
end


local total_sum = reduce(list, sum_fn, 0)
```

We pass reduce a starting number of `0`. What happens is `sum_fn` is invoked with the first parameter, the `accumulator` being the starting number 0 and `current_number` being the first number in the list. Whatever value the function returns becomes the new value for `accumulator` next loop around.

Lua doesn't have a reduce function built in so we'll implement our own here with a detailed description of all the parameters. Try not to get too hung up on the actual reduce function's implementation at the top, but rather focus below that on how it works. There will be several more examples. Once you understand how to use it, go back to the top and look at the actual `reduce` function's implementation. Copy all this code into the text editor window on the REPL and run it:

```lua
-- Applies fn on two arguments cumulative to the items of the array t,
-- from left to right, so as to reduce the array to a single value. If
-- a first value is specified the accumulator is initialized to this,
-- otherwise the first value in the array is used.
-- @param {table} t - a table to reduce
-- @param {function} fn - the reducer for comparing the two values
--   @param {*} acc - The accumulator accumulates the callback's return
--     values; It is the accumulated value previously returned in the
--     last invocation of the callback, or `first_value`, if supplied.
--   @param {*} current_value - The current element being processed in the list
--   @param {number} current_index - The index of the current element
--     being processed in the list, starting at 1.
-- @param {*} first_value - The initial value of the accumulation. If the array
--   empty, the first_value will also be the returned value. If the array is em
--   and no first value is specified an error is raised.
-- @example
--   -- returns 'zxy'
--   reduce(
--     { 'x', 'y' },
--     function(a, b) return a + b end,
--     'z'
--    )
local function reduce(t, fn, first)
  local acc = first
  local starting_value = first ~= nil
  for i, v in ipairs(t) do
    -- No starting value, start on
    -- the first element in the list
    if starting_value then
      acc = fn(acc, v, i, t)
    else
      acc = v
      starting_value = true
    end
  end
  assert(
    starting_value,
    'Attempted to reduce an empty table with no first value.'
  )
  return acc
end

local list = {23, 63, 12, 48, 3}
local sum_fn = function(accumulator, current_number)
  print(accumulator)
  return accumulator + current_number
end

local total_sum = reduce(list, sum_fn, 0)
print('The total sum is:', total_sum)
```

Following the `print` statement inside of `sum_fn`, we can see that the `accumulator` starts out with the 0 we pass in. We add `current_number` to `accumulator` and it begins to *accumulate* all the values as it goes.

```
0
23
86
98
146
The total sum is:   149
```

If we don't pass in a starting number, the accumulator will begin right away with the first number in the list:

```
local sum_fn = function(accumulator, current_number)
  print(accumulator)
  return accumulator + current_number
end

local total_sum = reduce(list, sum_fn)
```

```
23
86
98
146
The total sum is:   149
```

If you've used javascript, you may be starting to see the uncanny resemblance it bears to javascript's reduce function. Both languages are very similar syntactically, and given the ubiquity of javascript this Lua implementation follows much of the same behavior.

Let's look at some more examples to better understand how to reduce and what situations doing so could prove useful. The reduce function is omitted in the following examples, but you can copy and paste the function in the REPL alongside the examples to run the code yourself.

```
-- Concatenate a list of words
local list = {'this', 'is', 'a', 'sentence'}

local sentence = reduce(list, function(acc, word, index, list)
  -- Add a period if this is the last word
  if index == #list then
    word = word .. '.'
  end
  -- Otherwise add a space between the words
  return acc .. ' ' .. word
end)

print(sentence)
```

```
this is a sentence.
```

```
-- Only keep odd numbers
local list = {23, 63, 12, 48, 3}
local odd_numbers = reduce(list, function(acc, current_number)
  if current_number % 2 == 0 then
    return acc
  end
  acc[#acc + 1] = current_number
  return acc
end, {})

for key, value in ipairs(odd_numbers) do
  print(value)
end
```

```
23
63
3
```

This looks similar to what we might do with the `filter` function previously covered in 3.3 - Map and filter. In fact, we can compose `filter` and `map` from `reduce`. Take a look at the same code refactored out:

```
local filter = function(list, predicate_fn)
  return reduce(list, function(acc, val, i, t)
    if predicate_fn(val, i, t) then
      acc[#acc + 1] = val
      return acc
    end
    return acc
  end, {})
end


-- Only keep odd numbers
local list = {23, 63, 12, 48, 3}
local odd_numbers = filter(list, function(current_number)
  return current_number % 2 ~= 0
end)

for key, value in ipairs(odd_numbers) do
  print(value)
end
```

An example of wrapping `reduce` with a new `map` function won't be explained here, but rather left up to the reader as an exercise at the end of this section.

Here's one more example that is a bit more complex, a function called `compose` that creates a pipeline for passing data through. It accomplishes this by passing any functions you give it through to `reduce` as a list:

```
-- Function that allows you to compose other functions
-- together to form a pipeline. The resulting pipeline
-- is a function that you can pass your intended data through.
local compose = function(...)
  -- "..." and "arg" are special keywords in Lua.
  -- See: https://www.lua.org/pil/5.2.html
  local fns = arg
  return function(x)
    return reduce(fns, function(acc, v)
      return v(acc)
    end, x)
  end
end

-- Some example composable functions
local add = function(x)
  return function(y)
    return y + x
  end
end
local multiply = function(x)
  return function(y)
    return y * x
  end
end
local subtract = function(x)
  return function(y)
    return y - x
  end
end


local number_pipeline = compose(add(12), multiply(2), subtract(9))
print(number_pipeline(3))
print(number_pipeline(2))
```

Alternative reduce implementations

Iterating tables

Let's go back to the implementation of reduce for a moment. Take a look at the implementation of it given above. Notice the iteration inside is using `ipairs` which expects an array/list-type table. If we wanted to reduce a non-list table we could modify `reduce` to first check if the table is an array and do appropriate iteration over the table whether or not it is. Let's test that:

```
local function reduce(t, fn, first)
  local get_iterator = function(t)
    if type(t) == 'table' then
      -- If property of 1 is empty then
      -- iterate as a regular keyed table
      if t[1] == nil then
        return pairs(t)
      end
      return ipairs(t)
    end
    error('Expected table, got ' .. tostring(t))
  end
  local acc = first
  local starting_value = first ~= nil
  -- Whether we do ipairs or pairs is conditional
  for i, v in get_iterator(t) do
    -- No starting value, start on
    -- the first element in the list
    if starting_value then
      acc = fn(acc, v, i, t)
    else
      acc = v
      starting_value = true
    end
  end
  assert(
    starting_value,
    'Attempted to reduce an empty table with no first value.'
  )
  return acc
end


local list = {
  monday = 23,
  tuesday = 63,
  wednesday = 12,
  thursday = 48,
  friday = 3
}
local total_sum = reduce(list, function(acc, current_number, key)
  print(key .. ': ' .. current_number)
  return acc + current_number
end)

print('total sum: ' .. total_sum)
```

This should print something like this:

```
wednesday: 12
friday: 3
thursday: 48
monday: 23
total sum: 149
```

Note that the order the keys are iterated in are not guaranteed. Also "tuesday" wasn't printed out because it was the starting number, but it was still included in the total. Passing an extra argument of `0` to `reduce` would have caused all the days to be passed through our reducer function and printed out.

Break early

Ok, here's another example that seems tricky at first glance; Let's say you implemented some search functionality on top of reduce like this:

```
local list = {23, 63, 12, 48, 3}

local find = function(list, predicate_fn)
  return reduce(list, function(acc, v, i, t)
    if predicate_fn(v, a, t) then
      return v
    end
    return acc
  end)
end

print(find(list, function(val)
  return val > 50
end))
print(find(list, function(val)
  return val % 8 == 0
end))
```

Which prints out the expected results:

```
63
48
```

But do you see what's problematic about this? If we find the results we want, the reduce function will keep running through the entire list unnecessarily. Typically when doing a search you only want the first item you find anyway, but the above implementation will return the last item found if more than one match is made. Do you remember how the reduce function passes in the table as the last argument to the reducer function? We can take control of iterator via the table and kill the iteration prematurely. This involved mutating the table:

```
local list = {23, 63, 12, 48, 3}

local find = function(list, predicate_fn)
  return reduce(list, function(acc, v, i, t)
    if predicate_fn(v, a, t) then
      -- If a result was found, destroy the next item in the list
      -- to prevent the iteration from going any further.
      t[i + 1] = nil
      return v
    end
    return acc
  end)
end

print(find(list, function(val)
  return val > 1
end))
```

This returns the correct result:

```
23
```

But if we loop over the table afterwards we can see we've messed with the original data which can lead to unexpected consequences in a real application. If your data is coming from an immutable source, meaning something is generating a new copy each time you use it then this wouldn't be a problem:

```
local generate_list = function()
  return {23, 63, 12, 48, 3}
end

reduce(generate_list(), function()
  ...
  ...
```

However we could fix all of this if we are willing to add another parameter to our reduce implementation.

```
local function reduce(t, fn, first)
  local get_iterator = function(t)
    if type(t) == 'table' then
      -- If property of 1 is empty then
      -- iterate as a regular keyed table
      if t[1] == nil then
        return pairs(t)
      end
      return ipairs(t)
    end
    error('Expected table, got ' .. tostring(t))
  end
  local acc = first
  local starting_value = first ~= nil
  for i, v in get_iterator(t) do
    -- Exit the loop when true
    local should_break = false
    -- No starting value, start on
    -- the first element in the list
    if starting_value then
      acc, should_break = fn(acc, v, i, t)
      if should_break then
        break
      end
    else
      acc = v
      starting_value = true
    end
  end
  assert(
    starting_value,
    'Attempted to reduce an empty table with no first value.'
  )
  return acc
end
```

Now if we pass `true` as a second return parameter then we will get the first number we are looking for instead of the last. Loop through and print out the list afterward to make sure we haven't mutated it unexpectedly.

```
local list = {23, 63, 12, 48, 3}

local find = function(list, predicate_fn)
  return reduce(list, function(acc, v, i, t)
    if predicate_fn(v, a, t) then
      return v, true
    end
    return acc
  end, false)
end

print(find(list, function(val)
  return val > 1
end))

for idx, val in ipairs(list) do
  print(idx, val)
end
```

reduce_right

Another possible change you would want to make is to replace the iterator with a custom-made one to transform data in a specific order or pattern. Taken from lua-users.org's Iteration Tutorial is this reverse-ipairs (`ripairs`) implementation that allows you to iterate over a table from right to left. This modified version of `reduce` is typically called `reduce_right` .

```
local function reduce_right(t, fn, first)
  local ripairs = function(t)
    local max = 1
    while t[max] ~= nil do
      max = max + 1
    end
    local function ripairs_it(t, i)
      i = i-1
      local v = t[i]
      if v ~= nil then
        return i,v
      else
        return nil
      end
    end
    return ripairs_it, t, max
  end
  local acc = first
  local starting_value = first ~= nil
  for i, v in ripairs(t) do
    -- Exit the loop when true
    local should_break = false
    -- No starting value, start on
    -- the first element in the list
    if starting_value then
      acc, should_break = fn(acc, v, i, t)
      if should_break then
        break
      end
    else
      acc = v
      starting_value = true
    end
  end
  assert(
    starting_value,
    'Attempted to reduce an empty table with no first value.'
  )
  return acc
end
```

Then swap out `reduce` for `reduce_right` in the places you want to use it:

```
local list = {23, 63, 12, 48, 3}

local find = function(list, predicate_fn)
  return reduce_right(list, function(acc, v, i, t)
    if predicate_fn(v, a, t) then
      return v, true
    end
    return acc
  end, false)
end

print(find(list, function(val)
  return val > 1
end))
```

Recursive

Since we talked about recursion in the last section, let's try a recursive implementation of `reduce`. Although with Lua there's no practical reason to choose a recursive implementation over a for-loop or while-loop implementation, doing recursion is fun.

```
local function reduce(t, fn, acc, key)
  -- Check for starting value
  if key == nil and acc == nil then
    key = next(t, key)
    acc = t[key]
  end
  -- Begin next iteration. Next is a Lua built-in function
  -- that fetches the next key in a table after the given key.
  -- See: https://www.lua.org/pil/7.3.html
  key = next(t, key)
  -- Return acc if we've iterated all keys
  if key == nil then
    return acc
  end
  local break_early = false
  -- Collect new accumulator from predicate function
  acc, break_early = fn(acc, t[key], key, t)
  -- Check to see if the predicate wants to end early
  if break_early then
    return acc
  end
  -- Recur
  return reduce(t, fn, acc, key, acc)
end


-- Test it by getting the total sum from a table like before
local list = {
  monday = 23,
  tuesday = 63,
  wednesday = 12,
  thursday = 48,
  friday = 3
}
local total_sum = reduce(list, function(acc, current_number, key)
  print(key .. ': ' .. current_number)
  return acc + current_number
end, 0)

print('total sum: ' .. total_sum)
```

This supports breaking early like the two previous implementations.

Exercises

- Create a `count` function that counts up the number of items in a list that match the predicate and returns the total. It should work like this:

```
local count = function(list, predicate_fn)
  ????
end

local list = {23, 63, 12, 48, 3}
-- Print number of items evenly divisible by 3 (should return 4)
print(count(list, function(v)
  return v % 3 == 0
end))
```

- Go back to the map section in 3.3 and see if you can reimplement the `map` function on top of `reduce`.